CDF

C# Reference Manual

Version 3.6.3, October 20, 2016

Space Physics Data Facility
NASA / Goddard Space Flight Center

Internet – gsfc-cdf-support@lists.nasa.gov

Contents

Chapter 1

1 Compiling

C#-CDF distribution is packaged in a self-extracting installer. Once the installer is downloaded and run, all distributed files, i.e., APIs, test programs, batch files, help information and the document, will be placed into a directory of choice, and environment variables, **PATH** and **CsharpCDFDir**, are automatically set. If an older version already exists in the host machine, the installer will try to remove it before the new one is installed.

To C#, CDF library is unmanaged code distributed in the native DLL. The distributed .DLLs were built from a 32-bit (x86) Windows and they can be run on a 32-bit Windows via the x86-compatible Common Language Runtime (CLR), as well as a 64-bit Windows under WOW64.

1.1 Namespaces

Several classes are created for C# applications that facilitate the calls to the native **CDF .DLL**. The **CDF namespace** has been set up to include these CDF related classes: **CDFConstants**, **CDFException, CDFAPIs**. and **CDFUtils**. CDFConstants provides commonly used constants that mimic to those defined in the .DLL CDFException provides the exception handling when a failed CDF operation is detected. CDFAPIs provide all (static) public (and private) methods that C# applications can call to interact with the similar, underlining functions provided by the CDF Standard Interface in the .DLL. CDFUtils provides several small utility tools. These classes are distributed in the form of **signed assemblies** , as **.DLLs**. To facilitate the access to functions in DLL, each C# application must use the "**cdf**" namespace in order to call the C#-CDF APIs. The following namespaces should be included by C# applications that call CDF APIs:

```
using System;
using System.Runtime.InteropServices;
using CDF;
```

1.2 Base Classes

CDFAPIs is the main class that provides the C#-CDF APIs. Class CDFAPIs inherits from CDFConstants class, which defines all constants referenced by the CDF. A C# application, if inheriting from the CDFAPIs class, can call all

CDFAPIs methods and refer CDFConstants' constants directly, without specifying their class names. CDFException class inherits from C#'s Exception class and CDFUtils class inherits from CDFConstants class as well, .

1.3 Compiling with Compiler Options

If a test application, e.g., TestCDF.cs, resides in the same directory as all distributed **.dll** files, the following command can be used to create an executable

```
csc /unsafe /platform:x86 /r:CDFAPIs.dll,CDFException.dll,
                          CDFConstants.dll,CDFUtils.dll   TestCDF.cs
```

csc.exe, the C# compiler, can be called automatically from an IDE such as Visual Studio .NET, or run from the command line if the PATH environment variable is set properly. csc.exe can be found in the **Windows's .NET Framework** directory, <windows>\Microsoft.NET\Framework\v#.# (v#.# as v3.5 or in the latest release version).

/unsafe option is required as pointers are used by C# applications to communicate with the CDF APIs and .DLL.
/platform:x86 option is required for the Windows running 64-bit OS as C#-CDF is built on an **x86** (32-bit) platform.

When the C#-CDF package is installed, the **PATH** environment variable is automatically modified to include the installation directory so the native CDF .DLL, **dllcdfcsharp.dll** , becomes available when a C# application calls CDF functions. Once the executable, TestCDF.exe, is created, it can be run from any directory.

If the C# applications that call CDF APIs reside in the directories other than the C#-CDF installation directory, the following compilation command can be used to create an executable (.exe):

```
csc /unsafe /platform:x86
    /lib:%CsharpCDFDir%
    /r:cdfapis.dll,cdfconstants.dll,cdfexception.dll,cdfutils.dll
    TestCDF.cs
```

where environment variable CsharpCDFDir, the installation directory for C#-CDF package, .is set when the installer is run.

When the executable is run, an exception of "**FileNotFoundException**" will be encountered as CDFAPIs could not be loaded. It's because the distributed CDF assemblies are considered **private** in the .NET environment. The .NET Framework's runtime, **Common Language Runtime** (**CLR**), will not be able to locate the files if the application resides in a different directory from the called assemblies. To make these assemblies **global** so CLR can locate, they need to be placed in the **Global Assembly Cache** (**GAC**) repository. Use the following steps to do so:

```
gacutil /i CDFConstants.dll
gacutil /i CDFException.dll
gacutil /i CDFAPIs.dll
gacutil /i CDFUtils.dll
```

gacutil.exe (Global Assembly Cache utility) is a **Microsoft Software Development Kits** (SDKs) utility that can insert, list and remove the assemblies to and from GAC. Gacutil.exe usually can be found at <Program Files>\Microsoft SDKs\Windows\v#.#\bin (v#.# as v6.0A or in the latest release version). Use "gacutil /u" to remove assemblies of older versions form GAC.

ildasm.exe is another SDKs utility that can be used to browse the assemblies for information as versions, keys, etc..

1.4 Sample programs

Several sample programs are included for distribution. **Qst2cs.cs** and **Qst2cs2.cs**, two quick test programs for C#, are similar and call the same APIs. The only significant difference between them is they use different overloaded methods when calling the same APIs: Qst2cs uses **pointers arguments**, if possible, while Qst2cs2 passes the base class **objects** for arguments, instead. Tests show that Qst2cs runs faster than Qst2cs2. **Qts2cEpoch.cs** and **Qst2cEpoch16.cs** are two sample programs that show how EPOCH-related functions are used. A batch file, **tocompile.bat**, is distributed along with the sample programs. Execute it from a Command Prompt window to compile the programs into executables (**.exe**). Run **totest.bat** to test the executables to make sure they all work fine.

Chapter 2

2 Programming Interface

2.1 Item Referencing

The following sections describe various aspects of the programming interface for C# applications.

For C# applications, all item numbers 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 Compatible Types

As C# and CDF .DLL may have different sizes of the same data types, e.g. long, the size compatibility must be enforced when passing the data between the two. On 32-bit Windows, **4-byte long** has been used all over in the CDF .DLL. However, long in C# is defined as **8-byte**. So, to make the size compatible, 4-byte **int** is used, instead, in C# for each long type variable in the .DLL. For CDF data of CDF_CHAR, or CDF_UCHAR type, it is represented by a string in C#. They are not size compatible, so conversion, performed in the APIs, is needed between a character array in .DLL and string in C#.

The C#-CDF operations normally involve two variables: the operation status, status, and the CDF identifier, id:

status — All C#-CDF functions, except CDFvarNum, CDFgetVarNum, CDFattrNum and CDFgetAttrNum, return an operation status. This status is defined as an **int** in .DLL and C#. The CDFerror method can be used to inquire the meaning of any status code. Appendix A lists the possible status codes along with their explanations. Chapter 5 describes how to interpret status codes.

id — An identifier (or handle) for a CDF that must be used when referring to a CDF. This identifier has a type of **void*** in .DLL and C#. A new void* is established whenever a CDF is created or opened, establishing a connection to that CDF on disk. The void* is used in all subsequent operations on a particular CDF. The void* must not be altered by an application. With this pointer, C# applications are always declared as "**unsafe**".

2.3 CDFConstants

CDF defines a set of constants that are used all over the .DLL. These constants are mimicked in CDFConstants class with compatible data types.

2.4 CDF status

These constants are of same type as the operation status, mentioned in 2.2.

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.

Status less than CDF_OK normally indicate an error. For most cases, an exception will be thrown.

2.5 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.

2.6 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.

The following table depicts the equivalent data type between the CDF and C#:

CDF Data Type	C# Data Type
CDF_BYTE	sbyte
CDF_INT1	sbyte
CDF_UINT1	byte
CDF_INT2	short
CDF_UINT2	ushort
CDF_INT4	int
CDF_UINT4	uint
CDF_INT8	long
CDF_REAL4	float
CDF_FLOAT	float
CDF_REAL8	double
CDF_DOUBLE	double
CDF_EPOCH	double
CDF_EPOCH16	double[][1]
CDF_TIME_TT2000	long
CDF_CHAR	string
CDF_UCHAR	string

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 (representing the length of the string, where each element is a character).

NOTE: Keep in mind that an long is 8 bytes and that an int is 4 bytes. Use int for CDF data types CDF_INT4 and CDF_UINT4, rather than long. Use long for CDF_INT8 and CDF_TIME_TT2000 data types.

[1] CDF_EPOCH16 has two doubles, which corresponds to an array as double[] in C#.

2.7 Data Encodings

A CDF's data encoding affects how its attribute entry and variable data values are stored (on disk). Attribute entry and variable values passed into the CDF library (to be written to a CDF) should always be in the host machine's native encoding. Attribute entry and variable values read from a CDF by the CDF library and passed out to an application will be in the currently selected decoding for that CDF (see the Concepts chapter in the CDF User's Guide).

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

When creating a CDF (via CDFcreate) or respecifying a CDF's encoding (via CDFsetEncoding), 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.)

2.8 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).
PC_DECODING	Indicates PC data representation.
NeXT_DECODING	Indicates NeXT data representation.
MAC_DECODING	Indicates Macintosh data representation.

The default decoding is HOST_DECODING. The other decodings may be selected via the CDFsetDecoding method. The Concepts chapter in the CDF User's Guide describes those situations in which a decoding other than HOST_DECODING may be desired.

2.9 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.

2.10 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.)

2.11 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. Amont the available 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.[2] 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.

2.12 Sparseness

2.12.1 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.

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

2.12.2 Sparse Arrays

The following types of sparse arrays for variables are supported by VFG.[3]

NO_SPARSEARRAYS	No sparse arrays.

Note: sparse array is not supported and will not be implmeneted.

2.13 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.)

2.14 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 CDFsetReadOnlyMode method. 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.

2.15 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 via CDFsetzMode method.

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

[3] The sparse arrays are not (and will not be) supported.

2.16 -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 CDFsetNegtoPosfp0Mode method.

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.

2.17 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.

2.18 Limits of Names and Other Character Strings

CDF_PATHNAME_LEN	Maximum length of a CDF file name. 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.
CDF_ATTR_NAME_LEN256	Maximum length of an attribute name.
CDF_COPYRIGHT_LEN	Maximum length of the CDF Copyright text.
CDF_STATUSTEXT_LEN	Maximum length of the explanation text for a status code.

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

The following example creates two CDF files: "MY_TEST1.cdf" is a V3.* file while "MY_TEST2.cdf" a V2.7 file.

```
.
.
void*          id1, id2;                                   /*  CDF  identifier.  */
int    status;                                             /*  Returned status code.  */

try {
   status =  CDFcreateCDF("MY_TEST1", &id1);

..
  CDFsetFileBackward(BACKWARDFILEon);
  status =  CDFCreateCDF("MY_TEST2", &id2);

} catch (CDFException ex) {
 }.
.
```

Another method is through an environment variable and no method call is needed (and thus no code change involved in any existing applications). The environment variable, **CDF_FILEBACKWARD** on Windows, 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 method call through CDFsetFileBackward will take the precedence over the environment variable.

You can use the **CDFgetFileBackward** method 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.

```
.
.
int    flag;                                               /*  Returned status code.  */
.
flag = CDFgetFileBackward();
```

2.20 Checksum

To ensure the data integrity while transferring CDF files from/to different platforms at different locations, the checksum feature was added in CDF V3.2 as an option for the single-file format CDF files (not for the multi-file format). By default, the checksum feature is not turned on for new files. Once the checksum bit is turned on for a

particular file, the data integrity check of the file is performed every time it is open; and a new checksum is computed and stored when it is closed. This overhead (performance hit) may be noticeable for large files. Therefore, it is strongly encouraged to turn off the checksum bit once the file integrity is confirmed or verified.

If the checksum bit is turned on, a 16-byte signature message (a.k.a. message digest) is computed from the entire file and appended to the end of the file when the file is closed (after any create/write/update activities). Every time such file is open, other than the normal steps for opening a CDF file, this signature, serving as the authentic checksum, is used for file integrity check by comparing it to the re-computed checksum from the current file. If the checksums match, the file's data integrity is verified. Otherwise, an error message is issued. Currently, the valid checksum modes are: **NO_CHECKSUM** and **MD5_CHECKSUM**, both defined in CDFConstants class. 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 method call 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.

The environment variable **CDF_CHECKSUM** on Windows 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 set a new CDF file with the MD5 checksum and set another existing file's checksum to none.

```
.
.
.
void*           id1, id2;                    /*  CDF  identifier.  */
int    status;                               /*  Returned status code.  */
int    checksum;                             /*  Checksum code.  */
.
.
status =  CDFCreateCDF("MY_TEST1", &id1);
.
status =  CDFsetChecksum(id1, MD5_CHECKSUM);
.
status =  CDFclose(id1);
.
status =  CDFopen("MY_TEST2", &id2);
.
status =  CDFsetChecksum(id2, NO_CHECKSUM);
.
status =  CDFclose(id2);
.
.
```

2.21 Data Validation

To ensure the data integrity of CDF files and secure operation of CDF-based applications, a data validation feature has been added to the CDF opening logic. This process, as the default, performs sanity checks on the data fields in the CDF's internal data structures to make sure that the values are within valid ranges and consistent with the defined values/types/entries. It also ensures that the variable and attribute associations within the file are valid. Any

compromised CDF files, if not validated properly, could cause applications to function unexpectedly, e.g., segmentation fault due to a buffer overflow. The main purpose of this feature is to safeguard the CDF operations, catch any bad data in the file and end the application gracefully if any bad data is identified. Using this feature, in most cases, will slow down the file opening process especially for large or very fragmented files. Therefore, it is recommended 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 for further information. [4]

This validation feature is controlled by setting/unsetting the environment variable **CDF_VALIDATE** on Windows is not set or set to "**yes**", all CDF files are subjected to the data validation process. If the environment variable is set to "**no**", then no validation is performed. The environment variable can be set at logon or through the command line, which goes into effect during a terminal session, or within an application, which is good only while the application is running. Setting the environment variable, using C method **CDFsetValidate**, at application level will overwrite the setup from the command line. The validation is set to be on when **VALIDATEFILEon** is passed in as an argument. **VALIDATEFILEoff** will turn off the validation. The function, **CDFgetValidate**,will return the validation mode, **1** (one) means data being validated, **0** (zero) otherwise. If the environment variable is not set, the default is to validate the CDF file upon opening.

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

```
.

.
void*          id                                  /*  CDF  identifier.  */
int    status                                      /*  Returned status code.  */
.
.
CDFsetValidate (VALIDATEFILEon)
status =  CDFopen("TEST", &id);
.
.
```

The following example turns off the data validation when the CDF file, "TEST" is open.

```
.
.
.
void*          id                      /*  CDF  identifier.  */
int            status                  /*  Returned status code.  */
.
.
CDFsetValidate (VALIDATEFILEoff)
status =  CDFopen("TEST", &id);

.
```

2.22 8-Byte Integer

Both data types of CDF_INT8 and CDF_TIME_TT2000 use 8-byes signed integer. C#'s "long" type is the one that matches to these two types.

[4] 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.

2.23 Leap Seconds

CDF's **CDF_TIME_TT2000** is the epoch value in nanoseconds since **J2000** (2000-01-01T12:00:00.000000000) with leap seconds included. The CDF uses an external or internal table for computing the leap seconds. The external table, if present and properly pointed to by a predefined environment variable, will be used over the internal one. When the C# package is installed, the external table and environment variables are set so it can be used. If the external table is deleted or no longer pointed by the environment variable, the internal, hard-coded table in the library is used. When a new leap second is added, if the external table is updated accordingly, then the software does not need to be upgraded. Refer to CDF User's Guide for leap seconds.

A tool program, **CDFleapsecondsInfo** distributed with the CDFpackage, will show how the table is referred and when the last leap second was added. Optionally, it can dump the table contents.

Chapter 3

3 Understanding the Application Interface

This chapter provides some basic information about the C#'s Application Interfaces (APIs) to CDF, and the native CDF .DLL The following chapter will describe each API in detail.

3.1 Unsafe Code and Pointers

To manipulate CDFs, pointers are used extensively in CDF APIs to interact with the CDF native .DLL. The CDF file structure object itself is presented as a pointer of void (**void***) once a CDF is opened or created. Data from CDF is read or written through the API calls via pointers directly or indirectly. All APIs for CDF operations and Epoch utilities are built as **unsafe** code. The C# applications, while calling CDF APIs, have to be declared as unsafe as well. Compilation of the applications needs to specify the "**/unsafe**" option.

3.2 Overloaded Methods and Arguments Passing

Each CDF API has a sequence of parameters, which define the set of arguments that must be provided for that method in C# applications. Being a strongly typed language, C#'s APIs to CDF follow the same rules for the parameters. APIs that perform CDF data get, put or inquire operations are overloaded as different signatures for the output parameters would make method calls more flexible.

The input parameters in APIs such as the CDF identifier, variable number, attribute number, entry number, record number, record counts and record indices, etc, are always of fixed types as they must be a scalar of type **void*** for CDF identifier, **int** for variable/attribute/entry number and record number/count, or an array of ints, or an array of ints, **int[]**, for record indices or indices counts. Compilation error will occur if any one of the such arguments from the applications does not match to that defined in the API. The output parameters can be in one of the allowable forms defined by the overloaded API methods. For example, for a returned data, var, of type int, the passing argument in the calling application can be either "out var" or &var, a pointer to the declared int variable.

For example, **CDFsetEncoding** and **CDFgetEncoding** are used to set and get the data encoding of a CDF. Both APIs take two parameters, the CDF identifier, always a pointer as void*, and the encoding, an int. CDFsetEncoding take both parameters from applications for input, while CDFgetEncoding has the CDF identifier as input and the encoding for output. The following code shows how these methods can be used.

To set a CDF's encoding,

```
int status;
void* id;
int encoding;
…
encoding = IBMPC_ENCODING;
status = CDFsetEncoding(id, encoding);
```

The CDF identifier, id, is set when a CDF is open or created. The encoding is set to PC encoding, defined in CDFConstants class.

Similarly, to get the CDF's encoding:

```
status = CDFgetEncoding(id, out encoding);
```

The "out" argument modifier is specified in the call as it will get the encoding value from the API.

This data inquiry method is overloaded. Alternatively, it can be called by passing in the pointer of the encoding:

```
status = CDFgetEncoding(id, &encoding);
```

APIs that read or write CDF data, either variable's data (and their pad value) or metadata, are overloaded as they need to be more flexible when dealing with data of different pre-defined CDF types, e.g., CDF_INT1, CDF_UINT1, CDF_FLOAT, CDF_CHAR, CDF_EPOCH, etc. To pass the data value(s) to the APIs, one of the following forms can be used, depending on the data type: **pointer, string, an array of strings** or **object**. To use data pointers in the API calls, the data has to be of a **numeric type** ,and **pre-allocated** if the parameter is a dimensional object. String or an array of strings involves data of **CDF_CHAR** or **CDF_UCHAR** type. As C#'s character/string has a different characteristic from the ASCII-based code in the CDF native DLL library, some manipulations are needed in APIs when dealing with such data. C# objects can be used, as a general form for all data value(s), when reading/writing data from CDF. The called APIs will handle the passed object and map it to its corresponding CDF data type. ***Boxing/unboxing*** the objects presented by the APIs and/or applications may be needed.

For example, overloaded methods: **CDFputzVarData** and **CDFgetzVarData** are used to write and read a **single data value** for an zVariable in a CDF. Both take five parameters: the first four, for input, are the CDF identifier, variable number, record number and indices with fixed types of: **void***, **int**, **int** and an **array of ints** (**int[]**), respectively, and the last parameter is for data value, as an input for CDFputzVarData or an output for CDFgetzVarData. To call CDFputzVarData, the data value can be passed in as is or use data value's pointer if the data is a numeric. To retrieve the data by CDFgetzVarData, either specify the "out" modifier for the value argument, or pass in the data value's pointer if the data is a numeric.

The following samples show how these arguments are set up to write a data value to record 1, indices [1,1] for zVariable, "zVar1", a 2-dimentional of CDF_INT2. The first one uses the data value as is:

```
int status;
void* id;
int varNum;
int recNum = 1;
int[] indices = new int[] {1,1};
short value = (short) 100;
…
varNum = CDFvarNum(id, "zVar1");
status = CDFputzVarData(id, varNum, recNum, indices, value);
```

Alternatively, the **pointer** to the value argument can be used:

```
…
status = CDFputzVarData(id, varNum, recNum, indices, &value);
```

To read the data value the same variable at the same record and indices: a **pointer** or "out" modifier can be used for the value argument:

```
short value;
…
status = CDFgetzVarData(id, varNum, recNum, indices, out value);
```

Alternatively, the **pointer** to the value argument can be used:

```
…
status = CDFgetzVarData(id, varNum, recNum, indices, &value);
```

If value is declared as an object, instead of its specific type short, use the proper **type casting** to unbox the object after the data is returned.

```
…
object value;
status = CDFgetzVarData(id, varNum, recNum, indices, out value);
short value1 = (short) value;
```

Overloaded APIs that handle the reads and writes of **multiple data values**, have the similar ways of passing arguments. **CDFputzVarRecordData** and **CDFgetzVarRecordData** are two APIs that are used to write and read a full data record, either a scalar or dimensional data, for an zVariable. They both take four parameters: the first three, as input, are the CDF identifier, variable number, record number of the fixed types: **void***, **int** and **int**, respectively, and the last one is data values as an input for CDFputzVarRecordData or output for CDFgetzVarRecordData. Calling CDFputzVarRecordData, the data values object can be passed in as is or use object's pointer if the data are numeric. The values object parameter in CDFgetzVarRecordData can be declared with its data type and dimensionality, and is passed in with the "**out**" modifier as **out string**, **array of strings**, as **out string[]**, or **out object**. For numeric type data values, the argument can be passed in as a pointer, as **void***.

The following samples show how the arguments are set in CDFputzVarRecordData to write the full record 1 for zVariable, "zVar1", a 2-dim [2,3] of type short. The first one passes the data value object as is, while the second one uses a pointer to the data values.

```
int status;
void* id;
int varNum;
int recNum = 1;
short[,] values = new short[2,3] {{1,2,3},{11,12,13}};
…
varNum = CDFvarNum(id, "zVar1");
status = CDFputzVarRecordData(id, varNum, recNum, values);
```

Alternatively, a pointer to the array of numeric values can be used:

```
fixed (void* pvalues = values) {
  status = CDFputzVarRecordData(id, varNum, recNum, pvalues);
}
```

To use a pointer for the data values, its object needs to be pre-allocated. A **fixed** statement is required to pin the numeric data value array object to avoid garbage collector moving the object around during the call.

For CDFgetzVarRecordData to read back the same variable's record data, one can use "out" modifier for the data values, or pass in a pointer to the pre-allocated data array.

```
void* id;
int varNum;
int recNum = 1;
short[,] values ;
…
varNum = CDFvarNum(id, "zVar1");
status = CDFgetzVarRecordData(id, varNum, recNum, out values);


Console.WriteLine(values[0.0]+","+ values[0.1]+","+ values[0.2]+"\n"+
                  values[1.0]+","+ values[1.1]+","+ values[1.2]);
```

Alternatively, allocate the array with its type and dimensionality, and pass in the pointer:

```
short[,] values = new short[2,3];
…
fixed (void* pvalues = values) {
  status = CDFgetzVarRecordData(id, varNum, recNum, pvalues);
}

Console.WriteLine(values[0.0]+","+ values[0.1]+","+ values[0.2]+"\n"+
                  values[1.0]+","+ values[1.1]+","+ values[1.2]);
```

If the data values, in this case, is declared as a base class object, instead of a two-dimensional short, the API will determine the variable's data type and dimensionality and create the data object accordingly before the data values are read. Unboxing the object with a proper type and dimensionality casting is needed after the object is returned.

```
object values;
status = CDFgetzVarRecordData(id, varNum, recNum, out values);


Console.WriteLine(((short[,])values)[0.0]+","+ ((short[,])values)[0.1]+","+ ((short[,])values)[0.2]+"\n"+
                  ((short[,])values)[1.0]+","+ ((short[,])values)[1.1]+","+ ((short[,])values)[1.2]);
```

NOTE: Using the pointers to pass in the pre-allocated data objects is limited to numeric types. For data read/write operations, it should have a better performance than passing base class objects as APIs with pointers can directly interact with the CDF native .DLL If a method returns multiple values, either use pointers for all those output arguments, or use "out" modifier for all of them. Mixing pointers and "out" modifier arguments in a call is not supported.,

Pointers can not be used for string data, being a CDF_CHAR or CDF_UCHAR type in a CDF, as different characteristics between the C# characters and the CDF ASCII based library. The data needs to be declared as an string or array of strings, or simply a base class object, if an output operation involve such type. Use base class object for read operations that involve string array data of more than one dimension.

For example, if variable zVar1 is a one dimensional of two elements, of type CDF_CHAR. This sample code will retrieve both strings from the second record from the variable:

```
int status;
void* id;
int varNum;
```

```
int recNum = 1;
string[] strings;
…
…
varNum = CDFvarNum(id, “zVar1”);
status = CDFgetzVarRecordData(id, varNum, recNum, out strings);

Console.WriteLine(strings[0]+”,”+ strings[1]);
```

If value strings is declared as an object, instead of an array of strings, then unboxing the object is needed when it is returned.

```
object strings;
status = CDFgetzVarRecordData(id, varNum, recNum, out strings);

Console.WriteLine(((string[])strings)[0]+”,”+ ((string[])strings)[1]);
```

3.3 Multi-Dimensional Arrays

For data involved multidimensional arrays, CDF’s native .DLL data structure is equivalent to the **rectangular array** in C#. Multidimensional arrays of jagged type are not supported by APIs. An extra dimension is added to the retrieved data if the operations involve multiple records. For example, to read two full records from a variable of two-dimensions, 3-by-4 by the hyper get method, the returned will be a three-dimensional, 2-by-3-by-4, object. Conversely, if the hyper read skips certain dimension(s) from an operation, the returned object’s dimensionality will be reduced accordingly. For example, to read a row or column from a variable’s two-dimensional record, the returned will be a single array of either column or row count.

3.4 Data Type Equivalent

The following list shows the data types used by CDF and their corresponding types in C#:

- CDF_INT1 sbyte
- CDF_INT2 short
- CDF_INT4 int
- CDF_INT8 long
- CDF_UINT1 byte
- CDF_UINT2 ushort
- CDF_UINT4 uint
- CDF_BYTE sbyte
- CDF_REAL float
- CDF_FLOAT float
- CDF_DOUBLE double
- CDF_REAL8 double
- CDF_EPOCH double
- CDF_EPOCH16 double[2]
- CDF_TIME_TT2000 long
- CDF_CHAR string (with manipulation)
- CDF_UCHAR string (with manipulation)

3.5 Fixed Statement

Fixed statement is required to pin C# managed data objects, mainly arrays of numeric data, so that pointers of the objects can be safely used and passed to the CDF APIs. By doing so, the objects' addresses in the heap won't be moved around by the garbage collector during the operation.

For example, CDFhyperGetzVarData method can be called to retrieve a number of data values for a zVariable. For instance, the following application code can be used to read the first four (4) records from a zVariable of 2-dim [2,3] of type CDF_INT4. The declared data buffer, a 3-dimensional of int, is blocked in the fixed statement when the call is made.

```
void*     id;
void      status;
int       varNum;
int       recNum = 0, recCount = 4, recInterval = 1;
int[,]    indices = new int[,] {0, 0};
int[,]    counts = new int[,] {2, 3};
int[,]    intervals = new int[,] {1,1};
int [,,]  data = new int[4,2,3];                    /* Dimension: record number, row, column */
…
…
fixed (void* pdata = data) {
  status = CDFhyperGetzVarData (id, varNum, recNum, recCount, recInterval, indices, counts, intervals, pdata);
}
…
.
```

3.6 Exception Handling

Except a few APIs, each call to a CDF method will return an operation status. If the status is abnormal, less than CDF_OK, an exception might be thrown. It is recommended that the code for the CDF-based application be surrounded by a try-catch block so an exception can be caught and handled. The methods to check the existence of a CDF entity, e.g., entry, attribute, variable, will not throw exception if that entity is not in the CDF. The returned, informational status will reflect so. Once an exception is thrown, the thrown object, if initiated from the CDF APIs, is a CDFException class object. There are a couple of class methods, **GetCurrentStatus** and **GetStatusMsg** ,which can be used to acquire the status when an exception is thrown and the descriptive information about that exception.

```
void*     id;
int       status;
int       encoding;
try {
  status = CDFopen("TEST", &id);
  …
  status = CDFgetEncoding(id, &encoding);
```

```
  …..
status = CDFclose(id);
} catch (CDFException ex) {
   Console.WriteLine("Exception: "+ex);
  Or,
    int status1 = ex.GetCurrentStatus();
    Console.WriteLine("Exception: "+ex.GetStatusMsg(status1));
 }
```

3.7 Dimensional Limitations

The C# to CDF APIs follow the same dimensional restriction as in the CDF native DLL: a limit of **ten** (10) dimensions a CDF variable's numeric typed data record can have. For **string** typed data, represented in a CDF file with CDF_CHAR or CDF_UCHAR type, a limit of four (4) dimensions is applied.

Chapter 4

4 Application Interface

This chapter covers all Application Interfaces (APIs) that C# programs can interact with CDF. These APIs are based on the CDF native extended Standard Interface functions: they bear the same names with the similar calling arguments. They are easier to develop and use, and require a much shorter learning curve. All these APIs are defined as static methods in **CDFAPIs** class.

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 description for each API will detail its parameters: their types, for input or output; and what the method returns. APIs that are **overloaded** use a special indicator, **TYPE** (and **TYPE2**, **TYPE3**, etc, if necessary), to specify the parameters that can have different signatures. The acceptable data types for each overloaded method are specified. For example, **CDFgetEncoding** method, returning a CDF's encoding, is described as:

```
int CDFgetEncoding (                    /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
TYPE  encoding);                        /*  out --  CDF encoding.  */
                                        /*  TYPE  --  int*  or "out int"  */
```

TYPE can be a pointer to an int, ***int**, or an argument modifier "out" for an int object as "**out int**". The following application shows how the API is used:

```
    int status;
    int encoding;
    void* id;
    ….
     status = CDFgetEncoding(id,&encoding);
  Or,
     status = CDFgetEncoding(id, out encoding);
```

APIs are grouped based on the CDF entity they operate on. These groups are for general library information, CDF as a whole, variable and attribute/entry.

4.1 Library Information

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

4.1.1 CDFgetDataTypeSize

```
int CDFgetDataTypeSize (                    /*  out --  Completion status code.  */
int  dataType,                              /*  in --  CDF  data type.  */
TYPE  numBytes);                            /*  out --  # of bytes for the given type.  */
                                            /*  TYPE -- int* or out int  */
```

CDFgetDataTypeSize returns the size (in bytes) of the specified CDF data type.

The arguments to CDFgetDataTypeSize are defined as follows:

dataType	The CDF supported data type.
numBytes	The size of dataType.

4.1.1.1. Example(s)

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

```
.
.
.
int    status;                              /*  Returned status code.  */
int    numBytes;                            /*  Number of bytes.  */
.
.
try {
  ….
  status = CDFgetDataTypeSize(CDF_INT4, &numBytes);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.1.2 CDFgetLibraryCopyright

```
int CDFgetLibraryCopyright (                /*  out --  Completion status code.  */
out string  copyright);                     /*  out -- Library copyright.  */
```

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

The arguments to CDFgetLibraryCopyright are defined as follows:

Copyright	The Copyright notice.

4.1.2.1. Example(s)

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

```
.
.
.
int    status;                                          /*  Returned status code. */
string          copyright;                              /*  CDF library copyright.  */
.
.
try {
  ….
  status = CDFgetLibraryCopyright(out copyright);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.1.3 CDFgetLibraryVersion

```
int CDFgetLibraryVersion (                              /*  out --  Completion status code.  */
TYPE  version,                                          /*  out -- Library version.  */
TYPE  release,                                          /*  out -- Library release.  */
TYPE  increment,                                        /*  out -- Library increment.  */
out string  subIncrement);                              /*  out -- Library sub-increment.  */
                                                        /*  TYPE  -- int* or “out int”  */
```

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

The arguments to CDFgetLibraryVersion are defined as follows:

version	The library version number.
release	The library release number.
increment	The library incremental number.
subIncrement	The library sub-incremental string, a single character.

4.1.3.1. Example(s)

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

```
.
.
.
int     status;                                  /*  Returned status code. */
int     version;                                 /*  CDF library version number.  */
int     release;                                 /*  CDF library release number.  */
int     increment;                               /*  CDF library incremental number.  */
string          subIncrement;                    /*  CDF library sub-incremental character.  */
.
.
try {
  ….
  status = CDFgetLibraryVersion(&version, &release, &increment, out subIncrement);
..or,
 status = CDFgetLibraryVersion(out version, out release, out increment, out subIncrement);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.1.4 CDFgetStatusText

```
int CDFgetStatusText(                    /*  out --  Completion status code.  */
int status,                              /*  in -- The status code.  */
out string message);                     /*  out -- The status text description.  */
```

CDFgetStatusText is identical to CDFerror, a legacy CDF function, (see section 4.2.8), and the use of this method is strongly encouraged over CDFerror as it might not be supported in the future. This method is used to inquire the text explanation of a given status code. Chapter 5 explains how to interpret status codes and Appendix A lists all of the possible status codes.

The arguments to CDFgetStatusText are defined as follows:

status	The status code to check.
message	The explanation of the status code.

4.1.4.1. Example(s)

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

```
.
.
.
void*           id;                      /*  CDF  identifier.  */
int     status;                          /*  Returned status code.  */
string          text;                    /*  Explanation text.  */
```

```
.
.
try {
  ….
  status =  CDFopenCDF  ("giss_wetl",  &id);
  …
  status = CDFclose(id);
.
} catch (CDFException ex) {
   text = CDFgetStatusMsg(ex.CDFgetCurrentStatus());…
}.
```

4.2 CDF

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

4.2.1 CDFclose

```
int CDFclose(                                   /*  out --  Completion status code.  */
void*  id);                                     /*  in --  CDF  identifier.  */
```

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

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

The arguments to CDFclose are defined as follows:

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

4.2.1.1. Example(s)

The following example will close an open CDF.

```
.
.
.
void*          id;                              /*  CDF  identifier.  */
int    status;                                  /*  Returned status code.  */
.
.
try {
   ….
```

```
    status = CDFopen("…", &id);
    status =  CDFclose  (id);
} catch (CDFException ex) {
   …
}.
```

4.2.2 CDFcloseCDF

```
int CDFcloseCDF (                                   /*  out --  Completion status code.  */
void*  id);                                         /*  in --  CDF  identifier.  */
```

CDFcloseCDF closes the specified CDF. This method is identical to CDFclose, a legacy CDF function. The use of this method is strongly encouraged over CDFclose as it might not be supported in the future. The CDF's cache buffers are flushed; the CDF's open file is closed (or files in the case of a multi-file CDF); and the CDF identifier is made available for reuse.

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

The arguments to CDFcloseCDF are defined as follows:

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

4.2.2.1. Example(s)

The following example will close an open CDF.

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
int    status;                                      /*  Returned status code.  */
.
.
try {
  ….
  status =  CDFopenCDF  ("giss_wetl",  &id);
  ...
  status =  CDFcloseCDF  (id);
} catch (CDFException ex) {
   …
}.
```

4.2.3 CDFcreate

```
int CDFcreate(                  /* out -- Completion status code. */
string  CDFname,                /* in -- CDF file name. */
int  numDims,                   /* in -- Number of dimensions, rVariables. */
int[]  dimSizes,                /* in -- Dimension sizes, rVariables. */
int encoding,                   /* in -- Data encoding. */
int  majority,                  /* in -- Variable majority. */
void*  *id);                    /* out -- CDF identifier. */
```

CDFcreate, a legacy CDF function, 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 CDFopenCDF, delete it with CDFdeleteCDF, 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. 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 2.7.
majority	The majority for variable data. Specify one of the majorities described in Section 2.9.
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.

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

4.2.3.1. Example(s)

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

```
.
.
.
```

```
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */
int    numDims  =  3;                                 /*  Number of dimensions, rVariables.  */
int[]  dimSizes = new int[]  {180,360,10};            /*  Dimension sizes, rVariables.  */
int    majority =  ROW_MAJOR;                         /*  Variable majority.  */
.
.
try {
   status =  CDFcreate  ("test1",  numDims,  dimSizes,  NETWORK_ENCODING, majority,  &id);
.
} catch (CDFException ex) {
  …
}.
```

4.2.4 CDFcreateCDF

```
int CDFcreateCDF(                                     /*  out --  Completion status code.  */
string  cdfName,                                      /*  in --  CDF file name.  */
void*  *id);                                          /*  out --  CDF  identifier.  */
```

CDFcreateCDF creates a CDF file. This method is a simple form of CDFcreate without the number of dimensions, dimensional sizes, encoding and majority arguments. It is the better method if only zVariables are to be created in the CDF. The created CDF will use the default encoding (HOST_ENCODING) and majority (ROW_MAJOR). A CDF cannot be created if it already exists. (The existing CDF will not be overwritten.) If you want to overwrite an existing CDF, you can either manually delete the file or open it with CDFopenCDF ,delete it with CDFdeleteCDF, and then recreate it with CDFcreateCDF. If the existing CDF is corrupted, the call to CDFopenCDF will fail. (An error code will be returned.) In this case you must delete the CDF at the command line. Delete the dotCDF file (having an extension of .cdf), and if the CDF has the multi-file format, delete all of the variable files (having extensions of .v0,.v1,. . . and .z0,.z1,.. .).

Note that a CDF file created with CDFcreateCDF can only accept zVariables, not rVariables. But this is fine since zVariables are more flexible than rVariables. See the third paragraph of Chapter 3 for the differences between rVariables and zVariables.

The arguments to CDFcreateCDF are defined as follows:

CDFname	The file name of the CDF to create. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters. A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on OpenVMS systems and environment variables on UNIX systems). **UNIX:** File names are case-sensitive.
id	The identifier for the created CDF. This identifier must be used in all subsequent operations on the CDF.

When a CDF is created, both read and write access are allowed. The default format for a CDF created with CDFcreateCDF is specified in the configuration file of your CDF distribution. Consult your system manager for this default.

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

4.2.4.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */
.
.
try {
  ….
  status =  CDFcreateCDF  ("test1", &id);
  ...
  ...
  status = CDFclose (id);
} catch (CDFException ex) {
   …
}
```

4.2.5 CDFdelete

```
int CDFdelete(                                        /*  out --  Completion status code.  */
void*  id);                                           /*  in --  CDF  identifier.  */
```

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

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

The arguments to CDFdelete are defined as follows:

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

4.2.5.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */
```

```
.
.
try {
  ….
  status =  CDFopen  ("test2",  &id);
  status =  CDFdelete  (id);
.
} catch (CDFException ex) {
   …
}.
```

4.2.6 CDFdeleteCDF

```
int CDFdeleteCDF(                                   /*  out --  Completion status code.  */
void*  id);                                         /*  in --  CDF  identifier.  */
```

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

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

The arguments to CDFdeleteCDF are defined as follows:

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

4.2.6.1. Example(s)

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

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
int    status;                                /*  Returned status code.  */
.
.
try {
  ….
  status =  CDFopenCDF  ("test2",  &id);
  …
  status = CDFdeleteCDF(id);
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.7 CDFdoc

```
int CDFdoc(                                 /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
TYPE  version,                              /*  out --  Version number.  */
TYPE  release,                              /*  out --  Release number.  */
out string  copyright);                     /*  out --  copyright.  */
                                            /*  TYPE  --  int* or "out int"  */
```

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

The arguments to CDFdoc are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
version	The version number of the CDF library that created the CDF.
release	The release number of the CDF library that created the CDF.
copyright	The Copyright notice of the CDF library that created the CDF. This string will contain a newline character after each line of the Copyright notice.

4.2.7.1. Example(s)

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

```
.
.
void*          id;                          /*  CDF  identifier.  */
int    status;                              /*  Returned status code.  */
int    version;                             /*  CDF  version number.  */
int    release;                             /*  CDF  release number.  */
string         copyright;                   /*  Copyright  notice.  */
.
.
try {
   ….
   status =  CDFdoc  (id,  &version,  &release,  out copyright);
  Or,
   status =  CDFdoc  (id,  out version,  out release,  out copyright);
.
} catch (CDFException ex) {
   …
}.
```

4.2.8 CDFerror[5]

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

CDFerror, a legacy CDF function, 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.

4.2.8.1. Example(s)

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

```
.
.
.
void*          id;                              /*  CDF  identifier.  */
int            status;                          /*  Returned status code.  */
string         text;                            /*  Explanation text.  */
.
.
try {
  ….
  status =  CDFopen  ("giss_wetl",  &id);
.
} catch (CDFException ex) {
   int status1 =  CDFerror(ex.GetCurrentStatus(), out text); …
}.
```

4.2.9 CDFgetCacheSize

```
int CDFgetCacheSize (                           /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
TYPE  numBuffers);                              /*  out --  CDF's cache buffers.  */
                                                /*  TYPE --  int* or "out int"  */
```

CDFgetCacheSize returns the number of cache buffers being used for the dotCDF file when a CDF is open. Refer to the CDF User's Guide for description of caching scheme used by the CDF library.

The arguments to CDFgetCacheSize are defined as follows:

[5] A legacy CDF function. While it is still available in V3.1, CDFgetStatusText is the preferred function for it.

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

4.2.9.1. Example(s)

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

```
.
.
.
void*          id;                    /* CDF identifier. */
int    status;                        /* Returned status code. */
int    numBuffers;                    /* CDF's cache buffers. */
.
.
try {
    ....
    status = CDFgetCacheSize (id, &numBuffers);
  Or,
    status = CDFgetCacheSize (id, out numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.10 CDFgetChecksum

```
int CDFgetChecksum (                  /* out -- Completion status code. */
void* id,                             /* in -- CDF identifier. */
TYPE checksum);                       /* out -- CDF's checksum mode. */
                                      /* TYPE -- int* or "out int" */
```

CDFgetChecksum returns the checksum mode of a CDF. The CDF checksum mode is described in Section 2.20.

The arguments to CDFgetChecksum are defined as follows:

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

4.2.10.1. Example(s)

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

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
int    checksum;                    /* CDF's checksum. */
.
.
try {
    ….
    status = CDFgetChecksum (id, &checksum);
  Or,
    status = CDFgetChecksum (id, out checksum);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.11 CDFgetCompression

```
int CDFgetCompression (                 /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
TYPE compressionType,                           /* out -- CDF's compression type. */
TYPE2 compressionParms,                 /* out -- CDF's compression parameters. */
TYPE compressionPercentage);            /* out -- CDF's compressed percentage. */
                                        /* TYPE -- int* or "out int" */
                                        /* TYPE2 -- int* or "out int[]" */
```

CDFgetCompression gets the compression information of the CDF. It returns the compression type (method) and, if compressed, the compression parameters and compression rate. CDF compression types/parameters are described in Section 2.11. The compression percentage is the result of the compressed file size divided by its original, uncompressed file size.[6]

The arguments to CDFgetCompression are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
compressionType	The type of the compression.
compressionParms	The parameters of the compression.
compressionPercentage	The compression rate.

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

4.2.11.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */
int    compressionType;                               /*  CDF's compression type.  */
int[]  compressionParms                               /*  CDF's compression parameters.  */
int    compressionPercentage;                         /*  CDF's compression rate.  */
.
.
try {
    ….
    compressionParms = new int[1];
    fixed (int*pcompresionParms = compressionParms) {
      status = CDFgetCompression (id, &compression, pcompressionParms, &compressionPercentage);
    }
  Or,
    status = CDFgetCompression (id, out compression, out compressionParms, out compressionPercentage);
  …
  …
} catch (CDFException ex) {
   …
}.
```

4.2.12 CDFgetCompressionCacheSize

```
int CDFgetCompressionCacheSize (                      /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
TYPE  numBuffers);                                    /*  out --  CDF's compressed cache buffers.  */
                                                      /*  TYPE  --  int* or "out int"  */
```

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

The arguments to CDFgetCompressionCacheSize are defined as follows:

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

4.2.12.1. Example(s)

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

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
int    numBuffers;                  /* CDF's compression cache buffers. */
.
.
try {
    ….
    status = CDFgetCompressionCacheSize (id, &numBuffers);
  Or,
    Status = CDFgetCompressionCacheSize (id, out numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.13 CDFgetCompressionInfo

```
int CDFgetCompressionInfo (                 /* out -- Completion status code. */
string  CDFname,                            /* in -- CDF name. */
TYPE  cType,                                /* out -- CDF compression type. */
TYPE2  cParms.                              /* out -- CDF compression
parameters. */
TYPE3 cSize.                                /* out -- CDF compressed size. */
TYPE3  uSize);                              /* out -- CDF decompressed size. */
                                            /* TYPE -- int* or "out int" */
                                            /* TYPE2 -- int* or "out int[]" */
                                            /* TYPE3 -- long* or "out long" */
```

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

The arguments to CDFgetCompressionInfo are defined as follows:

CDFname	The pathname of a CDF file without the .cdf file extension.
cType	The CDF compression type.
cParms	The CDF compression parameters.
cSize	The compressed CDF file size.
uSize	The size of CDF when decompress the originally compressed CDF.

4.2.13.1. Example(s)

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

```
.
.
.
int     status;                                          /*  Returned status code.  */
int     cType;                                           /*  Compression type.  */
int[]  cParms;                                           /*  Compression parameters.  */
long  cSize;                                             /*  Compressed file size.  */
long  uSize;                                             /*  Decompressed file size.  */
.
.
try {
    ….
    cParms = new int[1];
    fixed (int* pcParms = cParms) {
      status = CDFgetCompressionInfo(“MY_TEST”, &cType, pcParms, &cSize, &uSize);
    }
  Or,
    status = CDFgetCompressionInfo(“MY_TEST”, out cType, out cParms, out cSize, out uSize);
    …
    ...
} catch (CDFException ex) {
   …
}.
```

4.2.14 CDFgetCopyright

```
int CDFgetCopyright (                                    /*  out --  Completion status code.  */
void*  id,                                               /*  in --  CDF  identifier.  */
out string  copyright);                                  /*  out --  Copyright notice.  */
```

CDFgetCopyright gets the Copyright notice in a CDF.

The arguments to CDFgetCopyright are defined as follows:

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

4.2.14.1. Example(s)

The following example returns the Copyright in a CDF.

```
.
.
.
void*          id;                                       /*  CDF  identifier.  */
int     status;                                          /*  Returned status code.  */
```

```
string          copyright;                          /* CDF's copyright. */
.
.
try {
  ….
  status = CDFgetCopyright (id, out copyright);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.15 CDFgetDecoding

```
int CDFgetDecoding (                                /* out -- Completion status code. */
void* id,                                           /* in -- CDF identifier. */
TYPE decoding);                                     /* out -- CDF decoding. */
                                                    /* TYPE -- int* or "out int" */
```

CDFgetDecoding returns the decoding code for the data in a CDF. The decodings are described in Section 2.8.

The arguments to CDFgetDecoding are defined as follows:

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

4.2.15.1. Example(s)

The following example returns the decoding for the CDF.

```
.
.
.
void*           id;                                 /* CDF identifier. */
int    status;                                      /* Returned status code. */
int    decoding;                                    /* Decoding. */
.
.
try {
    ….
    status = CDFgetDecoding(id, &decoding);
  Or,
   status = CDFgetDecoding(id, out decoding);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.16 CDFgetEncoding

```
int CDFgetEncoding (                    /* out -- Completion status code.  */
void* id,                               /* in -- CDF identifier.  */
TYPE encoding);                         /* out -- CDF encoding.  */
                                        /* TYPE -- int* or "out int"  */
```

CDFgetEncoding returns the data encoding used in a CDF. The encodings are described in Section 2.7.

The arguments to CDFgetEncoding are defined as follows:

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

4.2.16.1. Example(s)

The following example returns the data encoding used for the given CDF.

```
.
.
.
void*           id;                     /* CDF identifier.  */
int    status;                          /* Returned status code.  */
int    encoding;                        /* Encoding.  */
.
.
try {
    ….
    status = CDFgetEncoding(id, &encoding);
  Or,
    status = CDFgetEncoding(id, out encoding);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.17 CDFgetFileBackward

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

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

The arguments to CDFgetFileBackward are defined as follows:

N/A

4.2.17.1. Example(s)

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

```
.
.
.
.
void*           id;                          /* CDF  identifier.  */
int    status;                               /* Returned status code.  */
int    mode;                                 /* Backward mode.  */
.
.
try {
  ….
  mode =  CDFgetFileBackward  ();
  if (mode ==  BACKWARDFILEon)  {
 .
  }

} catch (CDFException ex) {
   …
}.
```

4.2.18 CDFgetFormat

```
int CDFgetFormat (                             /* out --  Completion status code.  */
void*  id,                                     /* in --  CDF  identifier.  */
TYPE  format);                                 /* out --  CDF format.  */
                                               /* TYPE  --  int* or "out int"  */
```

CDFgetFormat returns the file format, single or multi-file, of the CDF. The formats are described in Section 2.5.

The arguments to CDFgetFormat are defined as follows:

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

4.2.18.1. Example(s)

The following example returns the file format of the CDF.

```
.
.
.
void*           id;                 /* CDF identifier.  */
int    status;                      /* Returned status code.  */
int    format;                      /* Format.  */
.
.
try {
   status = CDFgetFormat(id, &format);
  Or,
   status = CDFgetFormat(id, &format);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.19 CDFgetLeapSecondLastUpdated

```
int CDFgetLeapSecondLastUpdated (      /* out --  Completion status code.  */
void*  id,                             /* in --  CDF  identifier.  */
TYPE  lastUpdated);                    /* out --  Leap second last updated date.  */
                                       /* TYPE  --  int* or "out int"  */
```

CDFgetLeapSecondLastUpdated returns the leap second last updated from the CDF. This value indicates what/if the leap second table this CDF is based on. It is of YYYYMMDD form. The value can also be negative 1 (-1), the field not set (for older CDFs), or zero (0) if the leap second table is not being accessed. This field is only relevant to TT2000 data in the CDF.

The arguments to CDFgetLeapSecondLastUpdated are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
lastUpdated	The date that the latest leap second was added to the leap second table.

4.2.19.1. Example(s)

The following example returns the date that the last leap second was added to the leap second table from the CDF.

```
.
.
.
void*           id;                 /* CDF identifier.  */
int    status;                      /* Returned status code.  */
int    lastUpdated;                 /* Date of the last updated.  */
.
```

```
.
try {
   status = CDFgetLeapSecondLastUpdated (id, &lastUpdated);
  Or,
   status = CDFgetLeapSecondLastUpdated (id, out lastUpdated);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.20 CDFgetLibraryCopyright

```
int CDFgetLibraryCopyright (                    /*  out --  Completion status code.  */
out String  copyRight);                         /*  out --  The copyright note.  */
```

CDFgetLibraryCopyright returns the CDF copy right statements.

The arguments to CDFgetLeapSecondLastUpdated are defined as follows:

copyRight	The copy right statements.

4.2.20.1. Example(s)

The following example returns the copy right note from the CDF library.

```
.
.
.
void*          id;                              /*  CDF identifier.  */
int    status;                                  /*  Returned status code.  */
String         copyRight;                       /*  Copy right note.  */
.
.
try {
   status = CDFgetLibraryCopyright (out copyRight);
  …
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.21 CDFgetLibraryVersion

```
int CDFgetLibraryVersion (                      /*  out --  Completion status code.  */
```

```
TYPE  version;                                  /*  out --  The version of the library.  */
TYPE  release;                                  /*  out --  The release of the library.  */
TYPE  increment;                                /*  out --  The increment of the library.  */
                                                /*  TYPE  --  int* or "out int"  */
out string  subIncrement);                      /*  out --  The subincrement of the library.  */
```

CDFgetLibraryVersion returns the CDF library version/release/increment/subincrement information.

The arguments to CDFgetLeapSecondLastUpdated are defined as follows:

version	The version
release	The release
increment	The increment
subincrement	The subincrement.

4.2.21.1. Example(s)

The following example returns the library version information from the CDF library.

```
.
.
.
void*       id;                                 /*  CDF identifier.  */
int         status;                             /*  Returned status code.  */
int         version, release, increment;        /*  version/release/increment.  */
string      subincrement;                       /*  Copy right note.  */
.
.
try {
  status = CDFgetLibraryVersion (&version, &release, &increment, out subincrement);
  or,
  status = CDFgetLibraryVersion (out version, out release, out increment, out subincrement);
 …
 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.2.22 CDFgetMajority

```
int CDFgetMajority (                            /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
TYPE  majority);                                        /*  out --  Variable majority.  */
                                                /*  TYPE  --  int* or "out int"  */
```

CDFgetMajority returns the variable majority, row or column-major, of the CDF. The majorities are described in Section 2.9.

The arguments to CDFgetMajority are defined as follows:

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

4.2.22.1. Example(s)

The following example returns the majority of the CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
int    status;                          /* Returned status code. */
int    majority;                        /* Majority. */
.
.
try {
   status = CDFgetMajority (id, &majority);
  Or,
   status = CDFgetMajority (id, out majority);

  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.23 CDFgetName

```
int CDFgetName (                        /* out -- Completion status code. */
void*  id,                              /* in -- CDF identifier. */
out string  name);                      /* out -- CDF name. */
```

CDFgetName returns the file name of the specified CDF.

The arguments to CDFgetName are defined as follows:

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

4.2.23.1. Example(s)

The following example returns the name of the CDF.

```
.
.
.
void*           id;                     /* CDF identifier.  */
int    status;                          /* Returned status code.  */
string          name;                   /* Name of the CDF.  */
.
.
try {
  ….
  status = CDFgetName (id, out name);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.24 CDFgetNegtoPosfp0Mode

```
int CDFgetNegtoPosfp0Mode (             /* out --  Completion status code.  */
void*  id,                              /* in --  CDF  identifier.  */
TYPE  negtoPosfp0);                     /* out --  -0.0 to 0.0 mode.  */
                                        /* TYPE  --  int* or "out int"  */
```

CDFgetNegtoPosfp0Mode returns the –0.0 to 0.0 mode of the CDF. You can use CDFsetNegtoPosfp0 method to set the mode. The –0.0 to 0.0 modes are described in Section 2.16.

The arguments to CDFgetNegtoPosfp0Mode are defined as follows:

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

4.2.24.1. Example(s)

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

```
.
.
.
void*           id;                     /* CDF  identifier.  */
int    status;                          /* Returned status code.  */
int    negtoPosfp0;                     /* -0.0 to 0.0 mode.  */
.
.
try {
   ….
   status = CDFgetNegtoPosfp0Mode (id, &negtoPosfp0);
```

```
  Or,
    status = CDFgetNegtoPosfp0Mode (id, out negtoPosfp0);

  …
  ….
} catch (CDFException ex) {
  …
}.
```

4.2.25 CDFgetReadOnlyMode

```
int CDFgetReadOnlyMode(                  /*  out --  Completion status code.  */
void*  id,                               /*  in --  CDF  identifier.  */
TYPE  readOnlyMode);                     /*  out --  CDF  read-only mode.  */
                                         /*  TYPE  --  int* or "out int"  */
```

CDFgetReadOnlyMode returns the read-only mode for a CDF. You can use CDFsetReadOnlyMode to set the mode of readOnlyMode. The read-only modes are described in Section 2.14.

The arguments to CDFgetReadOnlyMode are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
readOnlyMode	The read-only mode (READONLYon or READONLYoff).

4.2.25.1. Example(s)

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

```
.
.
.
void*          id;                       /*  CDF identifier. */
int    readMode;                         /*  CDF read-only mode.  */
.
.
try {
    ….
    status = CDFgetReadOnlyMode (id, &readMode);
  Or,
    status = CDFgetReadOnlyMode (id, out readMode);

  …
  ...
} catch (CDFException ex) {
  …
}.
```

4.2.26 CDFgetStageCacheSize

```
int CDFgetStageCacheSize(                  /* out -- Completion status code.  */
void*  id,                                 /* in -- CDF  identifier.  */
TYPE  numBuffers);                         /* out -- The stage cache size.  */
                                           /* TYPE  --  int* or "out int"  */
```

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

The arguments to CDFgetStageCacheSize are defined as follows:

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

4.2.26.1. Example(s)

The following example returns the number of cache buffers used in a CDF.

```
.
.
.
void*            id;                       /* CDF identifier. */
int    numBufffers;                        /* The number of cache buffers. */
.
.
try {
   ….
   status = CDFgetStageCacheSize (id,  &numBuffers);
  Or,
   status = CDFgetStageCacheSize (id,  out numBuffers);

  …
  …
.
} catch (CDFException ex) {
   …
}.
```

4.2.27 CDFgetValidate

```
int  CDFgetValidate();                     /* out – CDF validation mode.  */
```

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

The arguments to CDFgetVersion are defined as follows:

N/A

4.2.27.1. Example(s)

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

```
.
.
.
void*           id;                 /*  CDF  identifier.  */
int    status;                      /*  Returned status code.  */
int    validate;                    /*  Data validation flag.  */
.
.
try {
  ….
  validate =  CDFgetValidate  ();
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.28 CDFgetVersion

```
int  CDFgetVersion(                 /*  out --  Completion status code.  */
void*  id,                          /*  in --  CDF  identifier.  */
TYPE  version,                      /*  out --  CDF version.  */
TYPE  release,                      /*  out --  CDF release.  */
TYPE  increment);                   /*  out --  CDF increment.  */
                                    /*  TYPE  --  int* or “out int”  */
```

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

The arguments to CDFgetVersion are defined as follows:

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

4.2.28.1. Example(s)

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

```
.
.
.
void*           id;                                   /* CDF  identifier.  */
int    status;                                        /* Returned status code.  */
int    version;                                       /* CDF version.  */
int    release;                                       /* CDF release  */
int    increment;                                     /* CDF increment.  */
.
.
try {
    ….
    status =  CDFgetVersion  (id,  &version, &release, &increment);
   Or,
    status =  CDFgetVersion  (id,  out version, out release, out increment);
 …
  ....
} catch (CDFException ex) {
   …
}.
```

4.2.29 CDFgetzMode

```
int  CDFgetzMode(                                     /* out --  Completion status code.  */
void*  id,                                            /* in --  CDF  identifier.  */
TYPE  zMode);                                         /* out --  CDF zMode.  */
                                                      /* TYPE  --  int* or "out int"  */
```

CDFgetzMode returns the zMode for a CDF file. The zModes are described in Section 2.15.

The arguments to CDFgetzMode are defined as follows:

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

4.2.29.1. Example(s)

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

```
.
```

```
.
.
void*           id;                     /* CDF identifier. */
int    status;                          /* Returned status code. */
int    zMode;                           /* CDF zMode. */
.
.
try {
   ….
   status = CDFgetzMode (id, &zMode);
 Or,
   status = CDFgetzMode (id, out zMode);

 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.2.30 CDFinquire

```
int CDFinquire(                     /* out -- Completion status code. */
void* id,                           /* in -- CDF identifier */
TYPE numDims,                               /* out -- Number of dimensions, rVariables. */
TYPE2 dimSizes,                     /* out -- Dimension sizes, rVariables. */
TYPE encoding,                              /* out -- Data encoding. */
TYPE majority,                      /* out -- Variable majority. */
TYPE maxRec,                        /* out -- Maximum record number in the CDF, rVariables. */
TYPE numVars,                               /* out -- Number of rVariables in the CDF. */
TYPE numAttrs);                     /* out -- Number of attributes in the CDF. */
                                    /* TYPE -- int* or "out int" */
                                    /* TYPE2 -- int* or "out int[]" */
```

CDFinquire returns the basic characteristics of a CDF. An application needs to know the number of rVariable dimensions and their sizes before it can access rVariable data (since all rVariables' dimension and dimension size are the same). Knowing the variable majority can be used to optimize performance and is necessary to properly use the variable hyper functions (for both rVariables and zVariables).

The arguments to CDFinquire are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
numDims	The number of dimensions for the rVariables in the CDF.
dimSizes	The dimension sizes of the rVariables in the CDF. dimSizes is a 1-dimensional array containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional rVariables this argument is ignored (but must be present).
encoding	The encoding of the variable data and attribute entry data. The encodings are defined in Section 2.7.

majority	The majority of the variable data. The majorities are defined in Section 2.9.
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.

4.2.30.1. Example(s)

The following example returns the basic information about a CDF.

```
.
.
.
void*           id;                    /*  CDF  identifier.  */
int    status;                         /*  Returned status code.  */
int    numDims;                        /*  Number of dimensions,  rVariables.  */
int[]  dimSizes;                       /*  Dimension sizes,  rVariables (allocate to
allow the
                                            allow the maximum # of dimensions).  */
int    encoding;                       /*  Data encoding.  */
int    majority;                       /*  Variable majority.  */
int    maxRec;                         /*  Maximum record number,  rVariables.  */
int    numVars;                        /*  Number of rVariables in CDF.  */
int    numAttrs;                       /*  Number of attributes in CDF.  */
.
.
try {
  ….
  dimSizes = new int[CDF_MAX_DIMS];
  fixed (int* pdimSizes = dimSizes) {
     status =  CDFinquire  (id,  &numDims,  pdimSizes,  &encoding,  &majority,
                            &maxRec,  &numVars,  &numAttrs);
   }
  Or,
    status =  CDFinquire  (id,  out numDims,  out dimSizes,  out encoding,  out majority,
                           out maxRec,  out numVars,  out numAttrs);

.
} catch (CDFException ex) {
   …
}.
```

4.2.31 CDFinquireCDF

```
int CDFinquireCDF(                  /* out -- Completion status code. */
void* id,                           /* in -- CDF identifier */
TYPE numDims,                       /* out -- Number of dimensions for rVariables. */
TYPE2 dimSizes,                     /* out -- Dimension sizes for rVariables. */
TYPE encoding,                      /* out -- Data encoding. */
TYPE majority,                      /* out -- Variable majority. */
TYPE maxrRec,                       /* out -- Maximum record number among rVariables in the
CDF. */
TYPE numrVars,                      /* out -- Number of rVariables in the CDF. */
TYPE maxzRec,                       /* out -- Maximum record number among zVariables in the
CDF. */
TYPE numzVars,                      /* out -- Number of zVariables in the CDF. */
TYPE numAttrs);                     /* out -- Number of attributes in the CDF. */
                                    /* TYPE -- int* or "out int" */
                                    /* TYPE2 -- int* or "out int[]" */
```

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

The arguments to CDFinquireCDF are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
numDims	The number of dimensions for the rVariables in the CDF. Note that all the rVariables' dimensionality in the same CDF file must be the same.
dimSizes	The dimension sizes of the rVariables in the CDF (note that all the rVariables' dimension sizes in the same CDF file must be the same). dimSizes is a 1-dimensional array containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional rVariables this argument is ignored (but must be present).
encoding	The encoding of the variable data and attribute entry data. The encodings are defined in Section 2.7.
majority	The majority of the variable data. The majorities are defined in Section 2.9.
maxrRec	The maximum record number written to an rVariable in the CDF. Note that the maximum record number written is also kept separately for each rVariable in the CDF. The value of maxRec is the largest of these.
numrVars	The number of rVariables in the CDF.
maxzRec	The maximum record number written to a zVariable in the CDF. Note that the maximum record number written is also kept separately for each zVariable in the CDF. The value of maxRec is the largest of these. Some zVariables may have fewer records than actually written. Use CDFgetzVarMaxWrittenRecNum to inquire the actual number of records written for an individual zVariable.
numzVars	The number of zVariables in the CDF.

numAttrs	The number of attributes in the CDF.

4.2.31.1. Example(s)

The following example returns the basic information about a CDF.

```
.
.
.
void*          id;                  /*  CDF  identifier.  */
int    status;                      /*  Returned status code.  */
int    numDims;                     /*  Number of dimensions,  rVariables.  */
int[]  dimSizes;                    /*  Dimension sizes,  rVariables (allocate to
                                        allow the maximum # of dimensions).  */
int    encoding;                    /*  Data encoding.  */
int    majority;                    /*  Variable majority.  */
int    maxrRec;                     /*  Maximum record number,  rVariables.  */
int    numrVars;                    /*  Number of rVariables in CDF.  */
int    maxzRec;                     /*  Maximum record number,  zVariables.  */
int    numzVars;                    /*  Number of zVariables in CDF.  */
int    numAttrs;                    /*  Number of attributes in CDF.  */
.
.
try {
   dimSizes = new int[CDF_MAX_DIMS];
   fixed (int* pdimSizes = dimSizes) {
     status =  CDFinquireCDF  (id,  &numDims,  pdimSizes,  &encoding,  &majority,
                               &maxrRec, &numrVars, &maxzRec,  &numzVars,  &numAttrs);
   }
 Or,
     status =  CDFinquireCDF  (id,  out numDims,  out dimSizes,  out encoding,  out majority,
                               out maxrRec, out numrVars, out maxzRec,  out numzVars,  out numAttrs);

  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.32 CDFopen

```
int CDFopen(                        /*  out --  Completion status code.  */
string  CDFname,                    /*  in --  CDF file name.  */
void*  *id);                        /*  out --  CDF  identifier.  */
```

CDFopen, a legacy CDF function, 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 method 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. 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.

4.2.32.1. Example(s)

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

```
.
.
.
void*       id;                                  /* CDF identifier. */
int  status;                                     /* Returned status code. */
string      CDFname = "NOAA1";                   /* file name of CDF. */
.
.
try {
   status = CDFopen (CDFname, &id);
.
} catch (CDFException ex) {
  …
}.
```

4.2.33 CDFopenCDF

```
int CDFopenCDF(                                  /* out -- Completion status code. */
string CDFname,                                  /* in -- CDF file name. */
void* *id);                                      /* out -- CDF identifier. */
```

CDFopenCDF opens an existing CDF. This method is identical to CDFopen, and the use of this method is strongly encouraged over CDFopen as it might not be supported in the future. The CDF is initially opened with only read access. This allows multiple applications to read the same CDF simultaneously. When an attempt to modify the CDF is made, it is automatically closed and reopened with read/write access. The method will fail if the application does not have or cannot get write access to the CDF.

The arguments to CDFopenCDF are defined as follows:

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

UNIX: File names are case-sensitive.

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

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

4.2.33.1. Example(s)

The following example will open a CDF named "NOAA1.cdf".

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
int    status;                                      /*  Returned status code.  */
string         CDFname  =  "NOAA1";                 /* file name of CDF.  */
.
.
try {
  ….
  status =  CDFopenCDF  (CDFname,  &id);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.34 CDFselect

```
int CDFselect(                                      /*  out --  Completion status code.  */
void*  id);                                         /*  in --  CDF  identifier.  */
```

CDFselect selects an opened CDF as the current CDF. Only one CDF is allowed to be current. To access data from a CDF, that CDF must be selected as the current. This method is no longer needed as the methods involved CDF operations always need the CDF identifier, as the first argument, so it can be set as current before other operations can be applied.

The arguments to CDFselect are defined as follows:

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

NOTE: When a CDF is opened, it becomes the current. No CDF is current after CDFcloseCDF is called to close the file.

4.2.34.1. Example(s)

The following example will select a CDF named “NOAA1.cdf” as the current CDF while another file “NOAA2.cdf” is also opened.

```
.
.
.
void*          id1, id2;                                  /*  CDF  identifier.  */
int    status;                                            /*  Returned status code.  */
string         CDFname1  =  "NOAA1";                      /* file name of CDF.  */
string         CDFname2  =  "NOAA2";                      /* file name of CDF.  */.
.
try {
  ….
  status =  CDFopenCDF  (CDFname1,  &id1);

  status =  CDFopenCDF  (CDFname2,  &id2);

  status = CDFselect(id1);
  ....
  status = CDFclose(id1);
  status = CDFclose(id2);
} catch (CDFException ex) {
   …
 }...
```

4.2.35 CDFselectCDF

```
int CDFselectCDF(                                         /*  out --  Completion status code.  */
void*  id);                                               /*  in --  CDF  identifier.  */
```

CDFselectCDF selects an opened CDF as the current CDF. Only one CDF is allowed to be current. To access data from a CDF, that CDF must be selected as the current. This method is no longer needed as the methods involved CDF operations always need the CDF identifier, as the first argument, so it can be set as current before other operations can be applied. This method is identical to CDFselect.

The arguments to CDFselectCDF are defined as follows:

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

NOTE: When a CDF is opened, it becomes the current. No CDF is current after CDFcloseCDF is called to close the file.

4.2.35.1. Example(s)

The following example will select a CDF named “NOAA1.cdf” as the current CDF while another file “NOAA2.cdf” is also opened.

```
.
.
.
void*          id1, id2;                          /*  CDF  identifier.  */
int    status;                                    /*  Returned status code.  */
string         CDFname1  =  "NOAA1";              /* file name of CDF.  */
string         CDFname2  =  "NOAA2";              /* file name of CDF.  */.
.
try {
  ….
  status =  CDFopenCDF  (CDFname1,  &id1);

  status =  CDFopenCDF  (CDFname2,  &id2);

  status = CDFselectCDF(id1);
  ....
  status = CDFclose(id1);
  status = CDFclose(id2);
} catch (CDFException ex) {
   …
 }...
```

4.2.36 CDFsetCacheSize

```
int CDFsetCacheSize (                             /*  out --  Completion status code.  */
void*  id,                                        /*  in --  CDF  identifier.  */
int  numBuffers);                                 /*  in --  CDF’s cache buffers.  */
```

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

The arguments to CDFsetCacheSize are defined as follows:

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

numBuffers The number of cache buffers.

4.2.36.1. Example(s)

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

```
.
```

```
.
.
void*           id;                     /* CDF identifier. */
int    status;                          /* Returned status code. */
int    cacheBuffers;                    /* CDF's cache buffers. */
.
.
cacheBuffers = 500;
try {
   ….
 status = CDFsetCacheSize (id, cacheBuffers);
 …
...
} catch (CDFException ex) {
  …
}.
```

4.2.37 CDFsetChecksum

```
int CDFsetChecksum (                    /* out -- Completion status code. */
void*  id,                              /* in -- CDF identifier. */
int  checksum);                         /* in -- CDF's checksum mode. */
```

CDFsetChecksum specifies the checksum mode for the CDF. The CDF checksum mode is described in Section 2.20.

The arguments to CDFsetChecksum are defined as follows:

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

4.2.37.1. Example(s)

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

```
.
.
.
void*           id;                     /* CDF identifier. */
int    status;                          /* Returned status code. */
int    checksum;                        /* CDF's checksum. */
.
.
checksum= NO_CHECKSUM;
try {
   ….
 status = CDFsetChecksum (id, checksum);
 …
 ...
```

```
} catch (CDFException ex) {
  …
}.
```

4.2.38 CDFsetCompression

```
int CDFsetCompression (                          /* out -- Completion status code.  */
void* id,                                        /* in -- CDF identifier.  */
int compressionType,                             /* in -- CDF's compression type.  */
TYPE compressionParms);                          /* in -- CDF's compression parameters.  */
                                                 /* TYPE -- int* or int[]  */
```

CDFsetCompression specifies the compression type and parameters for a CDF. This compression refers to the CDF, not of any variables. The compressions are described in Section 2.11.

The arguments to CDFsetCompression are defined as follows:

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

4.2.38.1. Example(s)

The following example uses GZIP.6 to compress the CDF file.

```
.
.
.
void*          id;                               /* CDF identifier.  */
int    status;                                   /* Returned status code.  */
int    compressionType;                          /* CDF's compression type.  */
int[]  compressionParms = new int[1]             /* CDF's compression parameters.  */
.
.
compressionType = GZIP_COMPRESSION;
compressionParms[0] = 6;
try {
   ….
   fixed(int* pcompressionParms = compressionParms) {
     status = CDFsetCompression (id, compressionType, pcompressionParms);
    }
 Or,
   status = CDFsetCompression (id, compressionType, compressionParms); …
 ...
} catch (CDFException ex) {
  …
}.
```

4.2.39 CDFsetCompressionCacheSize

```
int CDFsetCompressionCacheSize (                    /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  compressionNumBuffers);                        /*  in --  CDF's compressed cache buffers.  */
```

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

The arguments to CDFsetCompressionCacheSize are defined as follows:

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

4.2.39.1. Example(s)

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

```
.
.
.
void*           id;                                 /*  CDF  identifier.  */
int    status;                                      /*  Returned status code.  */
int    compressionNumBuffers;                       /*  CDF's compression cache buffers.  */
.
.
compressionNumBuffers = 100;
try {
   ….
 status = CDFsetCompressionCacheSize (id, compressionNumBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.40 CDFsetDecoding

```
int CDFsetDecoding (                                /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  decoding);                                     /*  in --  CDF decoding.  */
```

CDFsetDecoding sets the decoding of a CDF. The decodings are described in Section 2.8.

The arguments to CDFsetDecoding are defined as follows:

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

4.2.40.1. Example(s)

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

```
.
.
.
void*           id;                          /*  CDF identifier.  */
int    status;                               /*  Returned status code.  */
int    decoding;                             /*  Decoding.  */
.
.
decoding = NETWORK_DECODING;
try {
    ….
  status = CDFsetDecoding (id, decoding);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.41 CDFsetEncoding

```
int CDFsetEncoding (                         /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  encoding);                              /*  in --  CDF encoding.  */
```

CDFsetEncoding specifies the data encoding of the CDF. A CDF's encoding may not be changed after any variable values have been written. The encodings are described in Section 2.7.

The arguments to CDFsetEncoding are defined as follows:

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

4.2.41.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF identifier.  */
int    status;                                        /*  Returned status code.  */
int    encoding;                                      /*  Encoding.  */
.
.
encoding = HOST_ENCODING;
try {
  status = CDFsetEncoding(id, encoding);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.42 CDFsetFileBackward

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

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

The arguments to CDFsetFileBackward are defined as follows:

mode	The backward mode.

4.2.42.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */

.
.
try {
```

```
    ….
  CDFsetFileBackward  (BACKWARDFILEoff);
   …
   …
} catch (CDFException ex) {
   …
}.
.
```

4.2.43 CDFsetFormat

```
int CDFsetFormat (                                  /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  format);                                       /*  in --  CDF format.  */
```

CDFsetFormat specifies the file format, either single or multi-file format, of the CDF. A CDF's format may not be changed after any variable values have been written. The formats are described in Section 2.5.

The arguments to CDFsetFormat are defined as follows:

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

4.2.43.1. Example(s)

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

```
.
.
.
void*           id;                                 /*  CDF identifier.  */
int    status;                                      /*  Returned status code.  */
int    format;                                      /*  Format.  */
.
.
format = MULTI_FILE;
try {
    ….
  status = CDFsetFormat(id, format);
   ...
   …
} catch (CDFException ex) {
   …
}.
```

4.2.44 CDFsetLeapSecondLastUpdated

```
int CDFsetLeapSecondLastUpdated (                    /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  lastUpdated);                                   /*  in --  CDF format.  */
```

CDFsetLeapSecondLastUpdated respecifies the date that the leap second was last updated in the CDF. The date, in YYYYMMDD form, has to be either a valid entry in the currently used leap second table, or zero (0). This function is called usually for older CDFs that have not had that information set. This field is onlt relevant to TT2000 data in a CDF.

The arguments to CDFsetLeapSecondLastUpdated are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
lastUpdated	The leap second last updated date.

4.2.44.1. Example(s)

The following example resets the the leap second last updated date to 20150701 in the CDF. The field might not be set (an older CDF).

```
.
.
.
void*            id;                                 /*  CDF identifier.  */
int    status;                                       /*  Returned status code.  */
int    lastUpdated;                                  /*  The last date the leap second was updated.  */
.
.
lastUpdated = 20150701;
try {
   ….
 status = CDFsetLeapSecondLastUpdated (id, lastUpdated);
  ...
  …
} catch (CDFException ex) {
   …
}.
```

4.2.45 CDFsetMajority

```
int CDFsetMajority (                                 /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  majority);                                      /*  in --  CDF variable majority.  */
```

CDFsetMajority specifies the variable majority, either row or column-major, of the CDF. A CDF's majority may not be changed after any variable values have been written. The majorities are described in Section 2.9.

The arguments to CDFsetMajority are defined as follows:

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

4.2.45.1. Example(s)

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

```
.
.
.
void*           id;                          /*  CDF identifier.  */
int    status;                               /*  Returned status code.  */
int    majority;                             /*  Majority.  */
.
.
majority = COLUMN_MAJOR;
try {
   ….
 status = CDFsetMajority (id, majority);
 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.2.46 CDFsetNegtoPosfp0Mode

```
int CDFsetNegtoPosfp0Mode (                  /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  negtoPosfp0);                           /*  in --  -0.0 to 0.0 mode.  */
```

CDFsetNegtoPosfp0Mode specifies the –0.0 to 0.0 mode of the CDF. The –0.0 to 0.0 modes are described in Section 2.16.

The arguments to CDFsetNegtoPosfp0Mode are defined as follows:

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

4.2.46.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF  identifier.  */
int    status;                                        /*  Returned status code.  */
int    negtoPosfp0;                                   /*  -0.0 to 0.0 mode.  */
.
.
negtoPosfp0 = NEGtoPOSfp0on;
try {
   ….
 status = CDFsetNegtoPosfp0Mode (id, negtoPosfp0);
 ...
} catch (CDFException ex) {
  …
}.
```

4.2.47 CDFsetReadOnlyMode

```
int CDFsetReadOnlyMode(                              /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  readOnlyMode);                                  /*  in --  CDF  read-only mode.  */
```

CDFsetReadOnlyMode specifies the read-only mode for a CDF. The read-only modes are described in Section 2.14.

The arguments to CDFsetReadOnlyMode are defined as follows:

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

4.2.47.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    readMode;                                      /*  CDF read-only mode.  */
.
.
readMode = READONLYoff;
try {
   ….
 status = CDFsetReadOnlyMode (id, readMode);
 …
```

```
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.48 CDFsetStageCacheSize

```
int CDFsetStageCacheSize(                    /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  numBuffers);                            /*  in --  The stage cache size.  */
```

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

The arguments to CDFsetStageCacheSize are defined as follows:

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

4.2.48.1. Example(s)

The following example sets the number of stage cache buffers to 10 for a CDF.

```
.
.
.
void*              id;                       /*  CDF identifier. */
int     numBufffers;                         /*  The number of cache buffers. */
.
.
numBufffers = 10;
try {
    ….
  status = CDFsetStageCacheSize (id, numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.2.49 CDFsetValidate

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

CDFsetValidate sets the data validation mode. The validation mode dedicates whether certain data in an open CDF file will be validated. This mode should be set before the any files are opened. Refer to Data Validation Section 2.21.

The arguments to CDFgetVersion are defined as follows:

mode	The validation mode.

4.2.49.1. Example(s)

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

```
.
.
.
try {
   ….
  CDFsetValidate  (VALIDATEFILEon);
  …
} catch (CDFException ex) {
   …
}
```

4.2.50 CDFsetzMode

```
int  CDFsetzMode(                                    /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  zMode);                                         /*  in --  CDF zMode.  */
```

CDFsetzMode specifies the zMode for a CDF file. The zModes are described in Section 2.15 and see the Concepts chapter in the CDF User's Guide for a more detailed information on zModes. zMode is used when dealing with a CDF file that contains 1) rVariables, or 2) rVariables and zVariables. If you want to treat rVariables as zVariables, it's highly recommended to set the value of zMode to zMODEon2.

The arguments to CDFsetzMode are defined as follows:

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

4.2.50.1. Example(s)

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

```
.
.
.
void*          id;                    /*  CDF  identifier.  */
int    status;                        /*  Returned status code.  */
int    zMode;                         /*  CDF zMode.  */
.
.
zMode = zMODEon2;
try {
   ….
 status =  CDFsetzMode  (id, zMode);
 …
 …
} catch (CDFException ex) {
  …
}.
```

4.3 Variables

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

4.3.1 CDFcloserVar

```
int CDFcloserVar(                     /*  out --  Completion status code.  */
void*  id,                            /*  in --  CDF  identifier.  */
int  varNum)                          /*  in --  rVariable number.  */
```

CDFcloserVar closes the specified rVariable file from a multi-file format CDF. Note that rVariables in a single-file CDF don't need to be closed. The variable's cache buffers are flushed before the variable's open file is closed. However, the CDF file is still open.

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

The arguments to CDFcloserVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The variable number for the open rVariable's file. This identifier must have been initialized by a call to CDFcreaterVar or CDFgetVarNum.

4.3.1.1. Example(s)

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

```
.
.
.
void*           id;                     /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
int    varNum;                          /*  rVariable number.  */
.
.
try {
    ….
 varNum = CDFgetVarNum (id, “VAR_NAME1”);
 .
 status =  CDFcloserVar  (id, varNum);
...
} catch (CDFException ex) {
   …
}.
```

4.3.2 CDFclosezVar

```
int CDFclosezVar(                       /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum)                            /*  in --  zVariable number.  */
```

CDFclosezVar closes the specified zVariable file from a multi-file format CDF. Note that zVariables in a single-file CDF don’t need to be closed. The variable's cache buffers are flushed before the variable's open file is closed. However, the CDF file is still open.

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

The arguments to CDFclosezVar are defined as follows:

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

4.3.2.1. Example(s)

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

```
.
.
.
void*           id;                     /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
int    varNum;                          /*  zVariable number.  */
.
.
try {
   ….
 varNum = CDFgetVarNum (id, “VAR_NAME1”);
 .
 status =  CDFclosezVar  (id, varNum);
...
} catch (CDFException ex) {
  …
}.
```

4.3.3 CDFconfirmrVarExistence

```
int CDFconfirmrVarExistence(            /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
string  varName);                       /*  in --  rVariable name.  */
```

CDFconfirmrVarExistence confirms the existence of a rVariable with a given name in a CDF. If the rVariable does not exist, an error code will be returned. No exception is thrown if the variable is not found.

The arguments to CDFconfirmrEntryExistence are defined as follows:

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

4.3.3.1. Example(s)

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

```
.
.
.
void*           id;                     /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
.
.
try {
   ….
 status =  CDFconfirmrVarExistence  (id, “MY_VAR”);
 if (status !=  CDF_OK)  UserStatusHandler  (status);
…
```

```
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.4 CDFconfirmrVarPadValueExistence

```
int CDFconfirmrVarPadValueExistence(          /* out -- Completion status code.  */
void*  id,                                    /* in -- CDF  identifier.  */
int  varNum)                                  /* in -- rVariable number.  */
```

CDFconfirmrVarPadValueExistence confirms the existence of an explicitly specified pad value for the specified rVariable in a CDF. If an explicit pad value has not been specified, the informational status code NO_PADVALUE_SPECIFIED will be returned. No exception is thrown if the variable's pad value is not defined.

The arguments to CDFconfirmrVarPadValueExistence are defined as follows:

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

4.3.4.1. Example(s)

The following example checks the existence of the pad value of rVariable "MY_VAR" in a CDF.

```
.
.
.
void*          id;                            /* CDF  identifier.  */
int    status;                                /* Returned status code.  */
int    varNum;                                /* rVariable number.  */
.
.
try {
    ….
  varNum = CDFgetVarNum(id, "MY_VAR");
  status =  CDFconfirmrVarPadValueExistence  (id, varNum);
  if (status != NO_PADVALUE_SPECIFIED)  {
.
  }
 …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.5 CDFconfirmzVarExistence

```
int CDFconfirmzVarExistence(                    /* out -- Completion status code. */
void* id,                                       /* in -- CDF identifier. */
string varName);                                /* in -- zVariable name. */
```

CDFconfirmzVarExistence confirms the existence of a zVariable with a given name in a CDF. If the zVariable does not exist, an error code will be returned. No exception is thrown if the variable is not found.

The arguments to CDFconfirmrEntryExistence are defined as follows:

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

4.3.5.1. Example(s)

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

```
.
.
.
void*          id;                              /* CDF identifier. */
int    status;                                  /* Returned status code. */
.
.
try {
    ….
  status = CDFconfirmzVarExistence (id, "MY_VAR");
  if (status != CDF_OK) UserStatusHandler (status);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.6 CDFconfirmzVarPadValueExistence

```
int CDFconfirmzVarPadValueExistence(            /* out -- Completion status code. */
void* id,                                       /* in -- CDF identifier. */
int varNum)                                     /* in -- zVariable number. */
```

CDFconfirmzVarPadValueExistence confirms the existence of an explicitly specified pad value for the specified zVariable in a CDF. If an explicit pad value has not been specified, the informational status code NO_PADVALUE_SPECIFIED will be returned. No exception is thrown if the variable's pad value is not defined.

The arguments to CDFconfirmzVarPadValueExistence are defined as follows:

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

varNum The zVariable number.

4.3.6.1. Example(s)

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

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    status;                                       /* Returned status code. */
int    varNum;                                       /* zVariable number. */
.
.
try {
    ….
  varNum = CDFgetVarNum(id, “MY_VAR”);
  status =  CDFconfirmzVarPadValueExistence  (id, varNum);
  if (status != NO_PADVALUE_SPECIFIED)  {
.
  }
 …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.7 CDFcreaterVar

```
int CDFcreaterVar(                                  /* out -- Completion status code. */
void*  id,                                          /* in -- CDF identifier. */
string  varName,                                    /* in -- rVariable name. */
int  dataType,                                      /* in -- Data type. */
int  numElements,                                   /* in -- Number of elements (of the data type). */
int  recVariance,                                   /* in -- Record variance. */
int[]  dimVariances,                                /* in -- Dimension variances. */
TYPE  varNum);                                          /* out -- rVariable number. */
                                                    /* TYPE -- int* or “out int” */
```

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

The arguments to CDFcreaterVar are defined as follows:

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

varName	The name of the rVariable to create. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
dataType	The data type of the new rVariable. Specify one of the data types defined in Section 2.6.
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 2.10.
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 2.10. 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 CDFgetVarNum function.

4.3.7.1. Example(s)

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

```
.
.
.
void*          id;                                          /* CDF identifier. */
int    status;                                              /* Returned status code. */
int    EPOCHrecVary = VARY;                                 /* EPOCH record variance. */
int    LATrecVary = NOVARY;                                 /* LAT record variance. */
int    LONrecVary = NOVARY;                                 /* LON record variance. */
int    TMPrecVary = VARY;                                   /* TMP record variance. */
int[]  EPOCHdimVarys = new int[] {NOVARY,NOVARY};           /* EPOCH dimension variances. */
int[]  LATdimVarys = new int[] {VARY,VARY};                 /* LAT dimension variances. */
int[]  LONdimVarys = new int[] {VARY,VARY};                 /* LON dimension variances. */
int[]  TMPdimVarys = new int[] {VARY,VARY};                 /* TMP dimension variances. */
int    EPOCHvarNum;                                         /* EPOCH rVariable number. */
int    LATvarNum;                                           /* LAT rVariable number. */
int    LONvarNum;                                           /* LON rVariable number. */
int    TMPvarNum;                                           /* TMP rVariable number. */
.
.
try {
 status = CDFcreaterVar (id, "EPOCH", CDF_EPOCH, 1, EPOCHrecVary,
                         EPOCHdimVarys, &EPOCH varNum);
 status = CDFcreaterVar (id, "LATITUDE", CDF_INT2, 1, LATrecVary, LATdimVarys, &LATvarNum);
 status = CDFcreaterVar (id, "INTITUDE", CDF_INT2, 1, LONrecVary, LONdimVarys, &LONvarNum);
 status = CDFcreaterVar (id, "TEMPERATURE", CDF_REAL4, 1, TMPrecVary,
                         TMPdimVarys, &TMPvarNum);
```

```
Or,
status = CDFcreaterVar (id, "EPOCH", CDF_EPOCH, 1, EPOCHrecVary,
                        EPOCHdimVarys, out EPOCHvarNum);
status = CDFcreaterVar (id, "LATITUDE", CDF_INT2, 1, LATrecVary, LATdimVarys,
                        out LATvarNum);
status = CDFcreaterVar (id, "INTITUDE", CDF_INT2, 1, LONrecVary, LONdimVarys,
                        out LONvarNum);
status = CDFcreaterVar (id, "TEMPERATURE", CDF_REAL4, 1, TMPrecVary,
                        TMPdimVarys, out TMPvarNum);

.
} catch (CDFException ex) {
  …
}.
```

4.3.8 CDFcreatezVar

```
int CDFcreatezVar(
void*  id,                                  /*  out --  Completion status code.  */
string  varName,                            /*  in --  CDF  identifier.  */
int  dataType,                              /*  in --  zVariable name.  */
int  numElements,                           /*  in --  Data type.  */
int  numDims,                               /*  in --  Number of elements  (of the data type).  */
int[]  dimSizes,                            /*  in --  Number of dimensions.  */
int  recVariance,                           /*  in --  Dimension sizes  */
int[]  dimVariances,                        /*  in --  Record  variance.  */
TYPE  varNum);                              /*  in --  Dimension variances.  */
                                                /*  out --  zVariable number.  */
                                            /*  TYPE  --  int* or "out int"  */
```

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

The arguments to CDFcreatezVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varName	The name of the zVariable to create. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
dataType	The data type of the new zVariable. Specify one of the data types defined in Section 2.6.
numElements	The number of elements of the data type at each value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (each value consists of the entire string). For all other data types this must always be one (1) - multiple elements at each value are not allowed for non-character data types.
numDims	Number of dimensions the zVariable. This may be as few as zero (0) and at most CDF_MAX_DIMS.

dimSizes	The size of each dimension. Each element of dimSizes specifies the corresponding dimension size. Each size must be greater then zero (0). For 0-dimensional zVariables this argument is ignored (but must be present).
recVariance	The zVariable's record variance. Specify one of the variances defined in Section 2.10.
dimVariances	The zVariable's dimension variances. Each element of dimVariances specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 2.10. For 0-dimensional zVariables this argument is ignored (but must be present).
varNum	The number assigned to the new zVariable. This number must be used in subsequent CDF function calls when referring to this zVariable. An existing zVariable's number may be determined with the CDFgetVarNum function.

4.3.8.1. Example(s)

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

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    status;                                       /* Returned status code. */
int    EPOCHrecVary = VARY;                          /* EPOCH record variance. */
int    LATrecVary = NOVARY;                          /* LAT record variance. */
int    LONrecVary = NOVARY;                          /* LON record variance. */
int    TMPrecVary = VARY;                            /* TMP record variance. */
int[]  EPOCHdimVarys = new int[] {NOVARY};           /* EPOCH dimension variances. */
int[]  LATdimVarys = new int[] {VARY,VARY};          /* LAT dimension variances. */
int[]  LONdimVarys = new int[] {VARY,VARY};          /* LON dimension variances. */
int[]  TMPdimVarys = new int[] {VARY,VARY};          /* TMP dimension variances. */
int    EPOCHvarNum;                                  /* EPOCH zVariable number. */
int    LATvarNum;                                    /* LAT zVariable number. */
int    LONvarNum;                                    /* LON zVariable number. */
int    TMPvarNum;                                    /* TMP zVariable number. */
int[]  EPOCHdimSizes = new int[] {3};                /* EPOCH dimension sizes. */
int[]  LATLONdimSizes = new int[] {2,3}              /* LAT/LON dimension sizes. */
int[]  TMPdimSizes = new int[] {3};                  /* TMP dimension sizes. */
.
.
try {
 status = CDFcreatezVar (id, "EPOCH", CDF_EPOCH, 1, 0, EPOCHdimSizes, EPOCHrecVary,
                         EPOCHdimVarys, &EPOCHvarNum);
 status = CDFcreatezVar (id, "LATITUDE", CDF_INT2, 1, 2, LATLONdimSizes,LATrecVary, LATdimVarys,
                         &LATvarNum);
 status = CDFcreatezVar (id, "INTITUDE", CDF_INT2, 1, 2, LATLONdimSizes, LONrecVary, LONdimVarys,
                         &LONvarNum);
 status = CDFcreatezVar (id, "TEMPERATURE", CDF_REAL4, 1, 1, TMPdimSizes, TMPrecVary,
                         TMPdimVarys, &TMPvarNum);

 Or,
```

```
  status =  CDFcreatezVar  (id,  "EPOCH", CDF_EPOCH,  1, 0, EPOCHdimSizes, EPOCHrecVary,
                            EPOCHdimVarys,  out EPOCHvarNum);
  status =  CDFcreatezVar  (id,  "LATITUDE",  CDF_INT2,  1, 2, LATLONdimSizes,LATrecVary,  LATdimVarys,
                            out LATvarNum);
  status =  CDFcreatezVar  (id,  "INTITUDE", CDF_INT2,  1, 2, LATLONdimSizes, LONrecVary,  LONdimVarys,
                            out LONvarNum);
  status =  CDFcreatezVar  (id, "TEMPERATURE", CDF_REAL4, 1, 1, TMPdimSizes, TMPrecVary,
                            TMPdimVarys, out TMPvarNum);

.
} catch (CDFException ex) {
  …
}.
```

4.3.9 CDFdeleterVar

```
int CDFdeleterVar(                                  /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  varNum);                                       /*  in --  rVariable  identifier.  */
```

CDFdeleterVar deletes the specified rVariable from a CDF.

The arguments to CDFdeleterVar are defined as follows:

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

4.3.9.1. Example(s)

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

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
int    status;                                      /*  Returned status code.  */
int    varNum;                                      /*  rVariable number.  */
.
.
try {
   ….
 varNum =  CDFgetVarNum (id, “MY_VAR”);
 status = CDFdeleterVar (id, varNum);
…
 ...
} catch (CDFException ex) {
  …
```

```
}.
```

4.3.10 CDFdeleterVarRecords

```
int CDFdeleterVarRecords(                  /*  out --  Completion status code.  */
void*  id,                                 /*  in --  CDF  identifier.  */
int  varNum,                               /*  in --  rVariable  identifier.  */
int  startRec,                             /*  in --  Starting record number.  */
int  endRec);                              /*  in --  Ending record number.  */
```

CDFdeleterVarRecords deletes a range of data records from the specified rVariable in a CDF.

The arguments to CDFdeleterVarRecords are defined as follows:

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

4.3.10.1. Example(s)

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

```
.
.
.
void*          id;                         /*  CDF  identifier.  */
int    status;                             /*  Returned status code.  */
int    varNum;                             /*  rVariable number.  */
int    startRec;                           /*  Starting record number.  */
int    endRec;                             /*  Ending record number.  */
.
.
try {
    ….
  varNum =  CDFgetVarNum (id, “MY_VAR”);
  startRec = 10;
  endRec = 20;
  status = CDFdeleterVarRecords (id, varNum, startRec, endRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.11 CDFdeletezVar

```
int CDFdeletezVar(                      /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum);                            /* in -- zVariable identifier. */
```

CDFdeletezVar deletes the specified zVariable from a CDF.

The arguments to CDFdeletezVar are defined as follows:

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

4.3.11.1. Example(s)

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

```
.
.
.
void*          id;                      /* CDF identifier. */
int    status;                          /* Returned status code. */
int    varNum;                          /* zVariable number. */
.
.
try {
    ….
  varNum =  CDFgetVarNum (id, “MY_VAR”);
  status = CDFdeletezVar (id, varNum);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.12 CDFdeletezVarRecords

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

CDFdeletezVarRecords deletes a range of data records from the specified zVariable in a CDF. If this is a variable with sparse records, the remaining records after deletion will not be renumbered.[7]

The arguments to CDFdeletezVarRecords are defined as follows:

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

4.3.12.1. Example(s)

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

```
.
.
.
void*          id;                                   /*  CDF  identifier.  */
int    status;                                       /*  Returned status code.  */
int    varNum;                                       /*  zVariable number.  */
int    startRec;                                     /*  Starting record number.  */
int    endRec;                                       /*  Ending record number.  */
.
.
try {
   ….
  varNum =  CDFgetVarNum (id, "MY_VAR");
  startRec = 10;
  endRec = 20;
  status = CDFdeletezVarRecords (id, varNum, startRec, endRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.13 CDFdeletezVarRecordsRenumber

```
int CDFdeletezVarRecordsRenumber(                    /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  zVariable  identifier.  */
int  startRec,                                       /*  in --  Starting record number.  */
```

[7] Normal variables without sparse records have contiguous physical records. Once a section of the records get deleted, the remaining ones automatically fill the gap.

```
int  endRec);                                   /*  in --  Ending record number.  */
```

CDFdeletezVarRecordsRenumber deletes a range of data records from the specified zVariable in a CDF. If this is a variable with sparse records, the remaining records after deletion will be renumbered, just like non-sparse variable's records.

The arguments to CDFdeletezVarRecordsRenumber are defined as follows:

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

4.3.13.1. Example(s)

The following example deletes 11 records (from record numbered 11 to 21) from the zVariable "MY_VAR" in a CDF. Note: The first record is numbered as 0. If the last record number is 100, then after the deletion, the record will be 89.

```
.
.
.
void*          id;                              /*  CDF  identifier.  */
int    status;                                  /*  Returned status code.  */
int    varNum;                                  /*  zVariable number.  */
int    startRec;                                /*  Starting record number.  */
int    endRec;                                  /*  Ending record number.  */
.
.
try {
    ….
  varNum =  CDFgetVarNum (id, "MY_VAR");
  startRec = 10;
  endRec = 20;
  status = CDFdeletezVarRecordsRenumber (id, varNum, startRec, endRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.14 CDFgetMaxWrittenRecNums

```
int CDFgetMaxWrittenRecNums (                   /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
TYPE  rVarsMaxNum,                              /*  out --  Maximum record number among all rVariables.  */
```

```
TYPE  zVarsMaxNum);                    /*  out --  Maximum record number among all zVariables.  */
                                       /*  TYPE  --  int* or "out int"  */
```

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

The arguments to CDFgetMaxWrittenRecNums are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
rVarsMaxNum	The maximum record number among all rVariables.
zVarsMaxNum	The maximum record number among all zVariables.

4.3.14.1. Example(s)

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

```
.
.
.
void*          id;                     /*  CDF  identifier.  */
int    status;                         /*  Returned status code.  */
int    rVarsMaxNum;                    /* Maximum record number among all rVariables.  */
int    zVarsMaxNum;                    /* Maximum record number among all zVariables.  */
.
.
try {
    ….
   status = CDFgetMaxWrittenRecNums (id, &rVarsMaxNum, &zVarsMaxNum);
  Or,
   status = CDFgetMaxWrittenRecNums (id, out rVarsMaxNum, out zVarsMaxNum);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.15 CDFgetNumrVars

```
int CDFgetNumrVars (                   /*  out --  Completion status code.  */
void*  id,                             /*  in --  CDF  identifier.  */
TYPE  numVars);                        /*  out --  Total number of rVariables.  */
                                       /*  TYPE  --  int* or "out int"  */
```

CDFgetNumrVars returns the total number of rVariables in a CDF.

The arguments to CDFgetNumrVars are defined as follows:

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

4.3.15.1. Example(s)

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

```
.
.

int     status;                          /*  Returned status code.  */
void*           id;                      /*  CDF identifier.  */
int     numVars;                         /*  Number of zVariables.  */

.
.
try {
    ….
    status = CDFgetNumrVars (id, &numVars);
  Or,
    status = CDFgetNumrVars (id, out numVars);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.16 CDFgetNumzVars

```
int CDFgetNumzVars (                     /*  out --  Completion status code.  */
void*  id,                               /*  in --  CDF  identifier.  */
TYPE  numVars);                          /*  out --  Total number of zVariables.  */
                                         /*  TYPE  --  int* or “out int”.  */
```

CDFgetNumzVars returns the total number of zVariables in a CDF.

The arguments to CDFgetNumzVars are defined as follows:

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

4.3.16.1. Example(s)

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

```
.
.
.
int     status;                                  /*  Returned status code.  */
void*           id;                              /*  CDF identifier.  */
int     numVars;                                 /*  Number of zVariables.  */

.
.
try {
    ….
    status = CDFgetNumzVars (id, &numVars);
  Or,
    status = CDFgetNumzVars (id, out numVars);

…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.17 CDFgetrVarAllocRecords

```
int CDFgetrVarAllocRecords(              /*  out --  Completion status code.  */
void*  id,                               /*  in --  CDF  identifier.  */
int  varNum,                             /*  in --  Variable number.  */
TYPE  numRecs);                          /*  out --  Allocated number of records.  */
                                         /*  TYPE  --  int* or “out int”.  */
```

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

The arguments to CDFgetrVarAllocRecords are defined as follows:

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

4.3.17.1. Example(s)

The following example returns the number of allocated records for rVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
int     varNum;                         /* rVariable number. */
int     numRecs;                        /* The allocated records. */
.
.
try {
    ....
      varNum = CDFgetVarNum (id, "MY_VAR");
.
      status = CDFgetrVarAllocRecords (id, varNum,  &numRecs);
  Or,
     status = CDFgetrVarAllocRecords (id, varNum,  out numRecs);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.18 CDFgetrVarBlockingFactor

```
int CDFgetrVarBlockingFactor(           /* out -- Completion status code.  */
void*  id,                              /* in -- CDF  identifier.  */
int  varNum,                            /* in --  Variable number.  */
TYPE  bf);                              /* out -- Blocking factor.  */
                                        /* TYPE  --  int* or "out int".  */
```

CDFgetrVarBlockingFactor returns the blocking factor for the specified rVariable in a CDF. Refer to the CDF User's Guide for a description of the blocking factor.

The arguments to CDFgetrVarBlockingFactor are defined as follows:

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

4.3.18.1. Example(s)

The following example returns the blocking factor for the rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
```

```
int    varNum;                                          /* rVariable number. */
int    bf;                                              /* The blocking factor. */
.
.
try {
  ….
    varNum = CDFgetVarNum (id, “MY_VAR”);

    status = CDFgetrVarBlockingFactor (id, varNum, &bf);
..Or,
    status = CDFgetrVarBlockingFactor (id, varNum, out bf);.
} catch (CDFException ex) {
  …
}.
```

4.3.19 CDFgetrVarCacheSize

```
int CDFgetrVarCacheSize(                               /* out -- Completion status code. */
void*  id,                                             /* in -- CDF identifier. */
int  varNum,                                           /* in -- Variable number. */
TYPE  numBuffers);                                     /* out -- Number of cache buffers. */
                                                       /* TYPE -- int* or “out int”. */
```

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

The arguments to CDFgetrVarCacheSize are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
numBuffers	The number of cache buffers.

4.3.19.1. Example(s)

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

```
.
.
.
void*          id;                                     /* CDF identifier. */
int    varNum;                                         /* rVariable number. */
int    numBuffers;                                     /* The number of cache buffers. */
.
.
try {
   ….
```

```
    varNum = CDFgetVarNum (id, "MY_VAR");
.
    status = CDFgetrVarCacheSize (id, varNum, &numBuffers);
  Or,
    status = CDFgetrVarCacheSize (id, varNum, out numBuffers);
…
 ...
} catch (CDFException ex) {
  …
}.
```

4.3.20 CDFgetrVarCompression

```
int CDFgetrVarCompression(                  /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int varNum,                                 /* in -- Variable number. */
TYPE cType,                                 /* out -- Compression type. */
TYPE2 cParms,                                       /* out -- Compression parameters. */
TYPE cPct);                                 /* out -- Compression percentage. */
                                            /* TYPE -- int* or "out int". */
                                            /* TYPE2 -- int* or "out int[]". */
```

CDFgetrVarCompression returns the compression type/parameters and compression percentage of the specified rVariable in a CDF. Refer to Section 2.11 for a description of the CDF supported compression types/parameters. The compression percentage is the result of the compressed size from all variable records divided by its original, uncompressed variable size.

The arguments to CDFgetrVarCompression are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
cType	The compression type.
cParms	The compression parameters.
cPct	The percentage of the uncompressed size of rVariable's data values needed to store the compressed values.

4.3.20.1. Example(s)

The following example returns the compression information for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*          id;                          /* CDF identifier. */
int    varNum;                              /* rVariable number. */
int    cType;                               /* The compression type. */
```

```
int[]  cParms;                                          /*  The compression parameters. */
int    cPct;                                            /*  The compression percentage. */
.
.
try {
   ….
   varNum = CDFgetVarNum (id, “MY_VAR”);
   cParms = new int[1];
   fixed (int* pcParms = cParms) {
       status = CDFgetrVarCompression (id, varNum, &cType, pcParms, &cPct);
    }
   Or
   status = CDFgetrVarCompression (id, varNum, out cType,  out cParms, out cPct);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.21 CDFgetrVarData

```
int CDFgetrVarData(                                     /*  out --  Completion status code.  */
void*  id,                                              /*  in --  CDF  identifier.  */
int  varNum,                                            /*  in --  Variable number.  */
int  recNum,                                            /*  in --  Record number.  */
int[]  indices,                                         /*  in --  Dimension indices.  */
TYPE  value);                                           /*  out --  Data value.  */
                                                        /*  TYPE  --  void*, “out string” or “out object”.  */
```

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

The arguments to CDFgetrVarData are defined as follows:

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

4.3.21.1. Example(s)

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

```
.
.
.
void* id;                                   /*  CDF identifier. */
int    varNum;                              /*  rVariable number. */
int    recNum;                              /*  The record number. */
int[]  indices = new int[2];                /*  The dimension indices. */
double          value1, value2;             /*  The data values. */
.
.
try {
   ….
   varNum = CDFgetVarNum (id, “MY_VAR”);
   recNum = 0;
   indices[0] = 0;
   indices[1] = 0;
   status = CDFgetrVarData (id, varNum, recNum, indices, &value1);
   indices[0] = 1;
   indices[1] = 1;
   object value2a;
   status = CDFgetrVarData (id, varNum, recNum, indices, out value2a);
   value2 = (double) value2a;
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.22 CDFgetrVarDataType

```
int CDFgetrVarDataType(                     /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
TYPE  dataType);                            /*  out --  Data type.  */
                                            /*  TYPE  --  int* or “out int”  */
```

CDFgetrVarDataType returns the data type of the specified rVariable in a CDF. Refer to Section 2.6 for a description of the CDF data types.

The arguments to CDFgetrVarDataType are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
dataType	The data type.

4.3.22.1. Example(s)

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

```
.
.
.
void*           id;                    /* CDF identifier. */
int    varNum;                         /* rVariable number. */
int    dataType;                       /* The data type. */
.
.
try {
   ….
    varNum = CDFgetVarNum (id, “MY_VAR”);
    status = CDFgetrVarDataType (id, varNum, &dataType);
  Or
    status = CDFgetrVarDataType (id, varNum, out dataType);
…
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.23 CDFgetrVarDimVariances

```
int CDFgetrVarDimVariances(            /* out -- Completion status code. */
void* id,                              /* in -- CDF identifier. */
int varNum,                            /* in -- Variable number. */
TYPE dimVarys);                        /* out -- Dimension variances. */
                                       /* TYPE -- int* or “out int[]”. */
```

CDFgetrVarDimVariances returns the dimension variances of the specified rVariable in a CDF. For 0-dimensional rVariable, this operation is not applicable. The dimension variances are described in section 2.10.

The arguments to CDFgetrVarDimVariances are defined as follows:

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

4.3.23.1. Example(s)

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

```
.
```

```
.
.
void*           id;                                   /*  CDF identifier. */
int[]  dimVarys;                                      /*  The dimension variances. */
.
.
try {
   ….

    dimVarys = new int[2];
    fixed (int* pdimVarys = dimVarys) {
…     status = CDFgetrVarDimVariances (id, CDFgetVarNum (id, “MY_VAR”), pdimVarys);
     }
   Or
    status = CDFgetrVarDimVariances (id, CDFgetVarNum (id, “MY_VAR”), out dimVarys);


 ...
} catch (CDFException ex) {
  …
}.
```

4.3.24 CDFgetrVarInfo

```
int CDFgetrVarInfo(                   /*  out --  Completion status code.  */
void*  id,                            /*  in --  CDF  identifier.  */
int  varNum,                          /*  in --  Variable number.  */
TYPE  dataType,                               /*  out --  Data type.  */
TYPE  numElems,                       /*  out --  Number of elements.  */
TYPE  numDims,                        /*  out --  Number of dimensions.  */
TYPE2  dimSizes);                     /*  out --  Dimension sizes.  */
                                      /*  TYPE  --  int* or “out int”.  */
                                      /*  TYPE2  --  int* or “out int[]”.  */
```

CDFgetrVarInfo returns the basic information about the specified rVariable in a CDF.

The arguments to CDFgetrVarInfo are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
dataType	The data type of the variable.
numElems	The number of elements for the data type of the variable.
numDims	The number of dimensions.
dimSizes	The dimension sizes.

4.3.24.1. Example(s)

The following example returns the basic information of rVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                 /* CDF identifier. */
int    dataType;                    /* The data type. */
int    numElems;                    /* The number of elements. */
int    numDims;                     /* The number of dimensions. */
int[]  dimSizes;                    /* The dimension sizes. */
.
.
try {
   ….

     dimVarys = new int[CDF_MAX_DIMS];
     fixed (int* pdimVarys = dimVarys) {
…     status = CDFgetrVarInfo (id, CDFgetVarNum (id, “MY_VAR”), &dataType, &numElems,
                                  &numDims, pdimVarys);
      }
   Or
     status = CDFgetrVarInfo (id, CDFgetVarNum (id, “MY_VAR”), out dataType, out numElems,
                                 out numDims, out dimVarys);


  ...
} catch (CDFException ex) {
   …
}.
```

4.3.25 CDFgetrVarMaxAllocRecNum

```
int CDFgetrVarMaxAllocRecNum(       /* out -- Completion status code. */
void*  id,                          /* in -- CDF identifier. */
int  varNum,                        /* in -- Variable number. */
TYPE  maxRec);                            /* out -- Maximum allocated record
#. */

                                    /* TYPE -- int* or “out int”. */
```

CDFgetrVarMaxAllocRecNum returns the number of records allocated for the specified rVariable in a CDF.

The arguments to CDFgetrVarMaxAllocRecNum are defined as follows:

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

4.3.25.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    maxRec;                                        /*  The maximum record number. */
.
.
try {
    ….
    status = CDFgetrVarMaxAllocRecNum (id, CDFgetVarNum (id, “MY_VAR”), &maxRec);
  Or
    status = CDFgetrVarMaxAllocRecNum (id, CDFgetVarNum (id, “MY_VAR”), out maxRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.26 CDFgetrVarMaxWrittenRecNum

```
int CDFgetrVarMaxWrittenRecNum (                  /*  out --  Completion status code.  */
void*  id,                                        /*  in --  CDF  identifier.  */
int  varNum,                                      /*  in --  Variable number.  */
TYPE  maxRec);                                          /*    out --    Maximum    written    record
number.  */
                                                  /*  TYPE  --  int* or “out int”.  */
```

CDFgetrVarMaxWrittenRecNum returns the maximum record number written for the specified rVariable in a CDF.

The arguments to CDFgetrVarMaxWrittenRecNum are defined as follows:

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

4.3.26.1. Example(s)

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

```
.
```

```
.
.
void*           id;                                   /*  CDF identifier. */
int    maxRec;                                        /*  The maximum record number. */
.
.
try {
    ….
    status = CDFgetrVarMaxWrittenRecNum (id, CDFgetVarNum (id, “MY_VAR”), &maxRec);
  Or
    status = CDFgetrVarMaxWrittenRecNum (id, CDFgetVarNum (id, “MY_VAR”), out maxRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.27 CDFgetrVarName

```
int CDFgetrVarName(                                   /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
out string  varName);                                 /*  out --  Variable name.  */
```

CDFgetrVarName returns the name of the specified rVariable, by its number, in a CDF.

The arguments to CDFgetrVarName are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
varName	The name of the variable.

4.3.27.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    varNum;                                        /*  rVariable number. */
string          varName;                              /*  The name of the variable. */
.
.
varNum = 1;
try {
    ….
```

```
    status = CDFgetrVarName (id, varNum, out varName);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.28 CDFgetrVarNumElements

```
int CDFgetrVarNumElements(                  /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
TYPE  numElems);                            /*  out --  Number of elements.  */
                                            /*  TYPE  --  int* or "out int".  */
```

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

The arguments to CDFgetrVarNumElements are defined as follows:

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

4.3.28.1. Example(s)

The following example returns the number of elements for the data type from rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                         /*  CDF identifier. */
int    numElems;                            /*  The number of elements. */
.
.
try {
   ….
   status = CDFgetrVarNumElements (id, CDFgetVarNum (id, "MY_VAR"), &numElems);
  Or
   status = CDFgetrVarNumElements (id, CDFgetVarNum (id, "MY_VAR"), out numElems);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.29 CDFgetrVarNumRecsWritten

```
int CDFgetrVarNumRecsWritten(                  /*  out --  Completion status code.  */
void*  id,                                     /*  in --  CDF  identifier.  */
int  varNum,                                   /*  in --  Variable number.  */
TYPE  numRecs);                                /*  out --  Number of written records.  */
                                               /*  TYPE  --  int* or "out int".  */
```

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

The arguments to CDFgetrVarNumRecsWritten are defined as follows:

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

4.3.29.1. Example(s)

The following example returns the number of written records from rVariable "MY_VAR" in a CDF.

```
.
.
.
void*            id;                           /*  CDF identifier. */
int    numRecs;                                /*  The number of written records. */
.
.
try {
    ….
    status = CDFgetrVarNumRecsWritten (id, CDFgetVarNum (id, "MY_VAR"), &numRecs);
  Or
    status = CDFgetrVarNumRecsWritten (id, CDFgetVarNum (id, "MY_VAR"), out numRecs);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.30 CDFgetrVarPadValue

```
int CDFgetrVarPadValue(                        /*  out --  Completion status code.  */
void*  id,                                     /*  in --  CDF  identifier.  */
```

```
int  varNum,                          /*  in --  Variable number.  */
TYPE  value);                         /*  out --  Pad value.  */
                                      /*  TYPE  --  void*, "out string" or "out object".  */
```

CDFgetrVarPadValue returns the pad value of the specified rVariable in a CDF. If a pad value has not been explicitly specified for the rVariable through CDFsetrVarPadValue, the informational status code **NO_PADVALUE_SPECIFIED** will be returned. Since a variable's pad value is an optional, no exception is thrown while trying to get its value if its value is not set. It's recommended to check the returned status after the method is called.

The arguments to CDFgetrVarPadValue are defined as follows:

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

4.3.30.1. Example(s)

The following example returns the pad value from rVariable "MY_VAR", a CDF_INT4 type variable, in a CDF.

```
.
.
.
void*            id;                          /*  CDF identifier. */
int    padValue;                              /*  The pad value. */
.
.
try {
    ….
    status = CDFgetrVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), &padValue);
   Or
    object padValueo;
    status = CDFgetrVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), out padValueo);
    if (status !=  NO_PADVALUE_SPECIFIED)  {
 .    padValue = (int) padValueo;
    }
 .
   }
. …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.31 CDFgetrVarRecordData

```
int CDFgetrVarRecordData(                  /*  out --  Completion status code.  */
void*  id,                                 /*  in --  CDF  identifier.  */
int  varNum,                               /*  in --  Variable number.  */
int  recNum,                               /*  in --  Record number.  */
TYPE  buffer);                             /*  out --  Record data.  */
                                           /*  TYPE --  void*, "out string", "out string[]" or "out object".
*/
```

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

The arguments to CDFgetrVarRecordData are defined as follows:

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

4.3.31.1. Example(s)

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

```
.
.
.
void*            id;                       /*  CDF identifier. */
int    varNum;                             /*  rVariable number. */
int[,] buffer1;                            /*  The data holding buffer – pre-allocation. */
int[,] buffer2;                            /*  The data holding buffer – API allocation.  */
.
.
try {
   ….
    varNum = CDFgetVarNum (id, "MY_VAR");
    buffer1 = new int[2,3];
    fixed (void* pbuffer1 = buffer1) {
      status = CDFgetrVarRecordData (id, varNum, 2, pbuffer1);
    }
    status = CDFgetrVarRecordData (id, varNum, 5, out buffer2);
 …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.32 CDFgetrVarRecVariance

```
int CDFgetrVarRecVariance(                  /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int varNum,                                 /* in -- Variable number. */
TYPE recVary);                                      /* out -- Record variance. */
                                            /* TYPE -- int* or "out int". */
```

CDFgetrVarRecVariance returns the record variance of the specified rVariable in a CDF. The record variances are described in Section 2.10.

The arguments to CDFgetrVarRecVariance are defined as follows:

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

4.3.32.1. Example(s)

The following example returns the record variance for the rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                         /* CDF identifier. */
int    recVary;                             /* The record variance. */
.
.
try {
  ….
    status = CDFgetrVarRecVariance (id, CDFgetVarNum (id, "MY_VAR"), &recVary);
  Or
    status = CDFgetrVarRecVariance (id, CDFgetVarNum (id, "MY_VAR"), out recVary);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.33 CDFgetrVarReservePercent

```
int CDFgetrVarReservePercent(               /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int varNum,                                 /* in -- Variable number. */
TYPE percent);                              /* out -- Reserve percentage. */
                                            /* TYPE -- int* or "out int". */
```

CDFgetrVarReservePercent returns the compression reserve percentage being used for the specified rVariable in a CDF. This operation only applies to compressed rVariables. Refer to the CDF User's Guide for a description of the reserve scheme used by the CDF library.

The arguments to CDFgetrVarReservePercent are defined as follows:

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

4.3.33.1. Example(s)

The following example returns the compression reserve percentage from the compressed rVariable "MY_VAR" in a CDF.

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    percent;                                      /*  The compression reserve percentage. */
.
.
try {
  ….
    status = CDFgetrVarReservePercent (id, CDFgetVarNum (id, "MY_VAR"), &percent);
  Or
    status = CDFgetrVarReservePercent (id, CDFgetVarNum (id, "MY_VAR"), out percent);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.34 CDFgetrVarsDimSizes

```
int CDFgetrVarsDimSizes(                          /*  out --  Completion status code.  */
void*  id,                                        /*  in --  CDF  identifier.  */
TYPE  dimSizes);                                  /*  out --  Dimension sizes.  */
                                                  /*  TYPE  --  int* or "out int[].  */
```

CDFgetrVarsDimSizes returns the size of each dimension for the rVariables in a CDF. (all rVariables have the same dimensional sizes.) For 0-dimensional rVariables, this operation is not applicable.

The arguments to CDFgetrVarsDimSizes are defined as follows:

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

dimSizes	The dimension sizes. Each element of dimSizes receives the corresponding dimension size.

4.3.34.1. Example(s)

The following example returns the dimension sizes for rVariables in a CDF.

```
.
.
.
void*       id;                                   /*  CDF identifier. */
.int[]      dimSizes;                             /* Dimensional sizes.  */

.try {
    ….
    dimSizes = new int[CDF_MAX_DIMS];
    fixed (int* pdimSizes = dimSizes) {
      status = CDFgetrVarsDimSizes (id, pdimSizes);
    }
  Or
       status = CDFgetrVarsDimSizes (id, out dimSizes);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.35 CDFgetrVarSeqData

```
int CDFgetrVarSeqData(                  /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum,                            /*  in --  Variable number.  */
TYPE  value);                           /*  out --  Data value.  */
                                        /*  TYPE  --  void*, “out string” or “out object”.  */
```

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

The arguments to CDFgetrVarSeqData are defined as follows:

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

4.3.35.1. Example(s)

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

```
.
.
.
void*           id;                          /*  CDF identifier. */
int     varNum;                              /*  The variable number from which to read data */
int     value1, value2;                      /*  The data value. */
int[]  indices = new int[2];                 /*  The indices in a record. */
int     recNum;                              /*  The record number. */
.
.
recNum = 2;
indices[0] = 0;
indices[1] = 0;
try {
  ….
  status = CDFsetrVarSeqPos (id, varNum, recNum, indices);
  status = CDFgetrVarSeqData (id, varNum, &value1);
  object value2o;
  status = CDFgetrVarSeqData (id, varNum, out value2o);
  value2 = (int) value2o;
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.36 CDFgetrVarSeqPos

```
int CDFgetrVarSeqPos(                        /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  varNum,                                 /*  in --  Variable number.  */
TYPE  recNum,                                /*  out --  Record number.  */
TYPE2  indices);                                    /*  out --  Indices in a record.  */
                                             /*  TYPE  --  int* or “out int”.  */
                                             /*  TYPE2 --  int* or “out int[]”.  */
```

CDFgetrVarSeqPos returns the current sequential value (position) for sequential access for the specified rVariable in a CDF. Note that a current sequential value is maintained for each rVariable individually. Use CDFsetrVarSeqPos method to set the current sequential value.

The arguments to CDFgetrVarSeqPos are defined as follows:

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

recNum — The rVariable record number.

indices — The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional rVariable, this argument is ignored, but must be presented.

4.3.36.1. Example(s)

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

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    recNum;                                       /* The record number. */
int[]  indices;                                      /* The indices. */
.
.
try {
  ….
    indices = new int[2];
    fixed (int* pindices = indices) {
      status = CDFgetrVarSeqPos (id, CDFgetVarNum(id, “MY_VAR”), &recNum, pindices);
    }
  Or
    status = CDFgetrVarSeqPos (id, CDFgetVarNum(id, “MY_VAR”), out recNum, out indices);
...
} catch (CDFException ex) {
  …
}.
```

4.3.37 CDFgetrVarsMaxWrittenRecNum

```
int CDFgetrVarsMaxWrittenRecNum(                     /* out -- Completion status code. */
void* id,                                            /* in -- CDF identifier. */
TYPE recNum);                                        /*    out -- Maximum record number.
*/
                                                     /* TYPE -- int* or “out int”. */
```

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

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

The arguments to CDFgetrVarsMaxWrittenRecNum are defined as follows:

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

recNum The maximum written record number.

4.3.37.1. Example(s)

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

```
.
.
void*          id;                                   /*  CDF identifier. */
int    recNum;                                       /*  The maximum record number. */
.
.
try {
  ….
    status = CDFgetrVarsMaxWrittenRecNum (id,  &recNum);
  Or
    status = CDFgetrVarsMaxWrittenRecNum (id,  out recNum);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.38 CDFgetrVarsNumDims

```
int CDFgetrVarsNumDims(                            /*  out --  Completion status code.  */
void*  id,                                         /*  in --  CDF  identifier.  */
TYPE  numDims);                                    /*  out --  Number of dimensions.  */
                                                   /*  TYPE  --  int* or “out int”.  */
```

CDFgetrVarsNumDims returns the number of dimensions (dimensionality) for the rVariables in a CDF.

The arguments to CDFgetrVarsNumDims are defined as follows:

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

numDims The number of dimensions.

4.3.38.1. Example(s)

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

```
.
.
.
```

```
void*           id;                     /* CDF identifier. */
int     numDims;                        /* The dimensionality of the variable. */
.
.
try {
    ….
   status = CDFgetrVarsNumDims (id, &numDims);
  Or
   status = CDFgetrVarsNumDims (id, out numDims);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.39 CDFgetrVarSparseRecords

```
int CDFgetrVarSparseRecords(                /* out -- Completion status code.  */
void*  id,                                  /* in -- CDF  identifier.  */
int  varNum,                                /* in -- The variable number.  */
TYPE  sRecordsType);                        /* out -- The sparse records type.  */
                                            /* TYPE  -- int* or "out int".  */
```

CDFgetrVarSparseRecords returns the sparse records type of the rVariable in a CDF. Refer to Section 2.12.1 for the description of sparse records.

The arguments to CDFgetrVarSparseRecords are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The variable number.
sRecordsType	The sparse records type.

4.3.39.1. Example(s)

The following example returns the sparse records type of the rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
int     sRecordsType;                   /* The sparse records type. */
.
.
try {
  ….
   status = CDFgetrVarSparseRecords (id,  CDFgetVarNum(id, "MY_VAR"), &sRecordsType);
```

```
  Or
    status = CDFgetrVarSparseRecords (id,  CDFgetVarNum(id, "MY_VAR"), out sRecordsType);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.40 CDFgetVarNum [8]

```
int CDFgetVarNum(                       /*  out --  Variable number.  */
void*  id,                              /*  in --  CDF  identifier.  */
string  varName);                       /*  in --  Variable name.  */
```

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

The arguments to CDFgetVarNum are defined as follows:

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

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

4.3.40.1. Example(s)

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

```
.
.
.
void*          id;                      /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
string         varName;                 /*  Variable name.  */
int    dataType;                        /*  Data type of the zVariable.  */
int    numElements;                     /*  Number of elements  (of the data type).  */
int    numDims;                         /*  Number of dimensions.  */
int[]  dimSizes;                        /*  Dimension sizes.  */
int    recVariance;                     /*  Record  variance.  */
int[]  dimVariances;                    /*  Dimension variances.  */
.
```

[8] Since no two variables, either rVariable or zVariable, can have the same name, this function now returns the variable number for the given rVariable or zVariable name (if the variable name exists in a CDF).

```
.
try {
   ….
   dimSizes = new int[1];
   dimVariances = new int[1];
   fixed (int* pdimSizes = dimSizes, pdimVariances = dimVariances) {
     status =  CDFinquirezVar  (id,  CDFgetVarNum(id,"LATITUDE"),  out varName,  &dataType,
                                  &numElements, &numDims,  pdimSizes , &recVariance,  pdimVariances);
   }
  Or,
    status =  CDFinquirezVar  (id,  CDFgetVarNum(id,"LATITUDE"),  out varName,  out dataType,
                                  out numElements, out numDims,  out dimSizes , out recVariance,
                                  out dimVariances);
…
  ...
} catch (CDFException ex) {
   …
}.
```

In this example the zVariable named LATITUDE was inquired. Note that if LATITUDE did not exist in the CDF, the call to CDFgetVarNum would have returned an error code. Passing that error code to CDFinquirezVar as a zVariable number would have resulted in CDFinquirezVar also returning an error code. Also note that the name written into varName is already known (LATITUDE). In some cases the zVariable names will be unknown - CDFinquirezVar would be used to determine them. CDFinquirezVar is described in Section 4.3.65.

4.3.41 CDFgetzVarAllocRecords

```
int CDFgetzVarAllocRecords(                 /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
TYPE  numRecs);                             /*  out --  Allocated number of records.  */
                                            /*  TYPE  --  int* or “out int”.  */
```

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

The arguments to CDFgetzVarAllocRecords are defined as follows:

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

4.3.41.1. Example(s)

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

```
.
```

```
.
.
void*           id;                     /* CDF identifier. */
int    varNum;                          /* zVariable number. */
int    numRecs;                         /* The allocated records. */
.
.
try {
   ….
     varNum = CDFgetVarNum (id, “MY_VAR”);
.
     status = CDFgetzVarAllocRecords (id, varNum,  &numRecs);
  Or,
    status = CDFgetzVarAllocRecords (id, varNum,  out numRecs);
…
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.42 CDFgetzVarBlockingFactor

```
int CDFgetzVarBlockingFactor(           /* out -- Completion status code.  */
void*  id,                              /* in -- CDF  identifier.  */
int  varNum,                            /* in --  Variable number.  */
TYPE  bf);                              /* out --  Blocking factor.  */
                                        /* TYPE  --  int* or “out int”.  */
```

CDFgetzVarBlockingFactor returns the blocking factor for the specified zVariable in a CDF. Refer to the CDF User’s Guide for a description of the blocking factor.

The arguments to CDFgetzVarBlockingFactor are defined as follows:

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

4.3.42.1. Example(s)

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

```
.
.
.
void*           id;                     /* CDF identifier. */
int    varNum;                          /* zVariable number. */
```

```
int     bf;                                          /*  The blocking factor. */
.
.
try {
   ….
     varNum = CDFgetVarNum (id, “MY_VAR”);

     status = CDFgetzVarBlockingFactor (id, varNum, &bf);
..Or,
     status = CDFgetzVarBlockingFactor (id, varNum, out bf);.
} catch (CDFException ex) {
   …
}.
```

4.3.43 CDFgetzVarCacheSize

```
int CDFgetzVarCacheSize(                     /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  varNum,                                 /*  in --  Variable number.  */
TYPE  numBuffers);                           /*  out --  Number of cache buffers.  */
                                             /*  TYPE  --  int* or “out int”.  */
```

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

The arguments to CDFgetzVarCacheSize are defined as follows:

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

4.3.43.1. Example(s)

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

```
.
.
.
void*          id;                           /*  CDF identifier. */
int    varNum;                               /*  zVariable number. */
int    numBuffers;                           /*  The number of cache buffers. */
.
.
try {
   ….
     varNum = CDFgetVarNum (id, “MY_VAR”);
```

```
.
    status = CDFgetzVarCacheSize (id, varNum, &numBuffers);
  Or,
    status = CDFgetzVarCacheSize (id, varNum, out numBuffers);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.44 CDFgetzVarCompression

```
int CDFgetzVarCompression(                  /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int varNum,                                 /* in -- Variable number. */
TYPE cType,                                 /* out -- Compression type. */
TYPE2 cParms,                                       /* out -- Compression parameters. */
TYPE cPct);                                 /* out -- Compression percentage. */
                                            /* TYPE -- int* or "out int". */
                                            /* TYPE2 -- int* or "out int[]". */
```

CDFgetzVarCompression returns the compression type/parameters and compression percentage of the specified zVariable in a CDF. Refer to Section 2.11 for a description of the CDF supported compression types/parameters. The compression percentage is the result of the compressed size from all variable records divided by its original, uncompressed variable size.

The arguments to CDFgetzVarCompression are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number.
cType	The compression type.
cParms	The compression parameters.
cPct	The percentage of the uncompressed size of zVariable's data values needed to store the compressed values.

4.3.44.1. Example(s)

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

```
.
.
.
void*          id;                          /* CDF identifier. */
int    varNum;                              /* zVariable number. */
int    cType;                               /* The compression type. */
```

```
int[]  cParms;                                          /*  The compression parameters. */
int    cPct;                                            /*  The compression percentage. */
.
.
try {
   ….
   varNum = CDFgetVarNum (id, “MY_VAR”);
   cParms = new int[1];
   fixed (int* pcParms = cParms) {
       status = CDFgetzVarCompression (id, varNum, &cType, pcParms, &cPct);
    }
   Or
   status = CDFgetzVarCompression (id, varNum, out cType,  out cParms, out cPct);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.45 CDFgetzVarData

```
int CDFgetzVarData(                                     /*  out --  Completion status code.  */
void*  id,                                              /*  in --  CDF  identifier.  */
int  varNum,                                            /*  in --  Variable number.  */
int  recNum,                                            /*  in --  Record number.  */
int[]  indices,                                         /*  in --  Dimension indices.  */
TYPE  value);                                           /*  out --  Data value.  */
                                                        /*  TYPE  --  void*, “out string” or “out object”.  */
```

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

The arguments to CDFgetzVarData are defined as follows:

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

4.3.45.1. Example(s)

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

```
.
.
.
void* id;                                          /*  CDF identifier. */
int     varNum;                                    /*  zVariable number. */
int     recNum;                                    /*  The record number. */
int[]  indices = new int[2];                       /*  The dimension indices. */
double          value1, value2;                    /*  The data values. */
.
.
try {
   ….
   varNum = CDFgetVarNum (id, “MY_VAR”);
   recNum = 0;
   indices[0] = 0;
   indices[1] = 0;
   status = CDFgetzVarData (id, varNum, recNum, indices, &value1);
   indices[0] = 1;
   indices[1] = 1;
   object value2a;
   status = CDFgetzVarData (id, varNum, recNum, indices, out value2a);
   value2 = (double) value2a;
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.46 CDFgetzVarDataType

```
int CDFgetzVarDataType(                            /*  out --  Completion status code.  */
void*  id,                                         /*  in --  CDF  identifier.  */
int  varNum,                                       /*  in --  Variable number.  */
TYPE  dataType);                                   /*  out --  Data type.  */
                                                   /*  TYPE  --  int* or “out int”  */
```

CDFgetzVarDataType returns the data type of the specified zVariable in a CDF. Refer to Section 2.6 for a description of the CDF data types.

The arguments to CDFgetzVarDataType are defined as follows:

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

4.3.46.1. Example(s)

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

```
.
.
.
void*           id;                    /* CDF identifier. */
int    varNum;                         /* zVariable number. */
int    dataType;                       /* The data type. */
.
.
try {
   ….
    varNum = CDFgetVarNum (id, “MY_VAR”);
    status = CDFgetzVarDataType (id, varNum, &dataType);
  Or
    status = CDFgetzVarDataType (id, varNum, out dataType);
…
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.47 CDFgetzVarDimSizes

```
int CDFgetzVarDimSizes(                /* out -- Completion status code. */
void* id,                              /* in -- CDF identifier. */
int varNum,                            /* in -- Variable number. */
TYPE dimSizes);                        /* out -- Dimension sizes. */
                                       /* TYPE -- int* or “out int[]. */
```

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

The arguments to CDFgetzVarDimSizes are defined as follows:

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

4.3.47.1. Example(s)

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

```
.
```

```
.
.
void*          id;                                   /*  CDF identifier. */
.int[]         dimSizes;                             /* Dimensional sizes.  */

.try {
   ….
   dimSizes = new int[CDF_MAX_DIMS];
   fixed (int* pdimSizes = dimSizes) {
     status = CDFgetzVarDimSizes (id, CDFgetVarNum(id, “MY_VAR”), pdimSizes);
   }
 Or
      status = CDFgetzVarDimSizes (id, CDFgetVarNum(id, “MY_VAR”), out dimSizes);
…
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.48 CDFgetzVarDimVariances

```
int CDFgetzVarDimVariances(                          /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  Variable number.  */
TYPE  dimVarys);                                     /*  out --  Dimension variances.  */
                                                     /*  TYPE  --  int* or “out int[]”.  */
```

CDFgetzVarDimVariances returns the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. The dimension variances are described in section 2.10.

The arguments to CDFgetzVarDimVariances are defined as follows:

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

4.3.48.1. Example(s)

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

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int[]  dimVarys;                                     /*  The dimension variances. */
.
.
```

```
try {
   ….

     dimVarys = new int[2];
     fixed (int* pdimVarys = dimVarys) {
…      status = CDFgetzVarDimVariances (id, CDFgetVarNum (id, “MY_VAR”), pdimVarys);
      }
   Or
     status = CDFgetzVarDimVariances (id, CDFgetVarNum (id, “MY_VAR”), out dimVarys);


  ...
} catch (CDFException ex) {
   …
}.
```

4.3.49 CDFgetzVarInfo

```
int CDFgetzVarInfo(                     /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum,                            /*  in --  Variable number.  */
TYPE  dataType,                                 /*  out --  Data type.  */
TYPE  numElems,                         /*  out --  Number of elements.  */
TYPE  numDims,                          /*  out --  Number of dimensions.  */
TYPE2  dimSizes);                       /*  out --  Dimension sizes.  */
                                        /*  TYPE  --  int* or “out int”.  */
                                        /*  TYPE2  --  int* or “out int[]”.  */
```

CDFgetzVarInfo returns the basic information about the specified zVariable in a CDF.

The arguments to CDFgetzVarInfo are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number.
dataType	The data type of the variable.
numElems	The number of elements for the data type of the variable.
numDims	The number of dimensions.
dimSizes	The dimension sizes.

4.3.49.1. Example(s)

The following example returns the basic information of zVariable “MY_VAR” in a CDF.

```
.
```

```
.
.
void*           id;                                   /*  CDF identifier. */
int    dataType;                                      /*  The data type. */
int    numElems;                                      /*  The number of elements. */
int    numDims;                                       /*  The number of dimensions. */
int[]  dimSizes;                                      /*  The dimension sizes. */
.
.
try {
   ….

     dimVarys = new int[CDF_MAX_DIMS];
     fixed (int* pdimVarys = dimVarys) {
…     status = CDFgetzVarInfo (id, CDFgetVarNum (id, “MY_VAR”), &dataType, &numElems,
                                  &numDims, pdimVarys);
      }
   Or
     status = CDFgetzVarInfo (id, CDFgetVarNum (id, “MY_VAR”), out dataType, out numElems,
                                   out numDims, out dimVarys);


  ...
} catch (CDFException ex) {
   …
}.
```

4.3.50 CDFgetzVarMaxAllocRecNum

```
int CDFgetzVarMaxAllocRecNum(                         /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
TYPE  maxRec);                                                /*   out --   Maximum allocated record
#.  */

                                                      /*  TYPE  --  int* or “out int”.  */
```

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

The arguments to CDFgetzVarMaxAllocRecNum are defined as follows:

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

4.3.50.1. Example(s)

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

```
.
.
.
void*           id;                                   /* CDF identifier. */
int    maxRec;                                        /* The maximum record number. */
.
.
try {
    ….
    status = CDFgetzVarMaxAllocRecNum (id, CDFgetVarNum (id, “MY_VAR”), &maxRec);
  Or
    status = CDFgetzVarMaxAllocRecNum (id, CDFgetVarNum (id, “MY_VAR”), out maxRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.51 CDFgetzVarMaxWrittenRecNum

```
int CDFgetzVarMaxWrittenRecNum (                  /* out -- Completion status code. */
void* id,                                         /* in -- CDF identifier. */
int varNum,                                       /* in -- Variable number. */
TYPE maxRec);                                     /* out -- Maximum written record
number. */
                                                  /* TYPE -- int* or “out int”. */
```

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

The arguments to CDFgetzVarMaxWrittenRecNum are defined as follows:

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

4.3.51.1. Example(s)

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

```
.
.
.
void*           id;                                   /* CDF identifier. */
int    maxRec;                                        /* The maximum record number. */
.
.
```

```
try {
   ….
   status = CDFgetzVarMaxWrittenRecNum (id, CDFgetVarNum (id, “MY_VAR”), &maxRec);
  Or
   status = CDFgetzVarMaxWrittenRecNum (id, CDFgetVarNum (id, “MY_VAR”), out maxRec);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.52 CDFgetzVarName

```
int CDFgetzVarName(                    /*  out --  Completion status code.  */
void*  id,                             /*  in --  CDF  identifier.  */
int  varNum,                           /*  in --  Variable number.  */
out string  varName);                  /*  out --  Variable name.  */
```

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

The arguments to CDFgetzVarName are defined as follows:

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

4.3.52.1. Example(s)

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

```
.
.
.
void*          id;                     /*  CDF identifier. */
int    varNum;                         /*  zVariable number. */
string         varName;                /*  The name of the variable. */
.
.
varNum = 1;
try {
   ….
   status = CDFgetzVarName (id, varNum, out varName);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.53 CDFgetzVarNumDims

```
int CDFgetzVarNumDims(                  /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum,                             /* in -- Variable number. */
TYPE numDims);                          /* out -- Number of dimensions. */
                                        /* TYPE -- int* or "out int". */
```

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

The arguments to CDFgetzVarNumDims are defined as follows:

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

4.3.53.1. Example(s)

The following example returns the number of dimensions for zVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
int    numDims;                         /* The dimensionality of the variable. */
.
.
try {
    ….
    status = CDFgetzVarNumDims (id, CDFgetVarNum(id, "MY_VAR"), &numDims);
  Or
    status = CDFgetzVarNumDims (id, CDFgetVarNum(id, "MY_VAR"), out numDims);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.54 CDFgetzVarNumElements

```
int CDFgetzVarNumElements(              /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
```

```
int  varNum,                                    /*  in --  Variable number.  */
TYPE  numElems);                                /*  out --  Number of elements.  */
                                                /*  TYPE  --  int* or "out int".  */
```

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

The arguments to CDFgetzVarNumElements are defined as follows:

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

4.3.54.1. Example(s)

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

```
.
.
.
void*            id;                            /*  CDF identifier. */
int    numElems;                                 /*  The number of elements. */
.
.
try {
    ….
    status = CDFgetzVarNumElements (id, CDFgetVarNum (id, "MY_VAR"), &numElems);
  Or
    status = CDFgetzVarNumElements (id, CDFgetVarNum (id, "MY_VAR"), out numElems);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.55 CDFgetzVarNumRecsWritten

```
int CDFgetzVarNumRecsWritten(                   /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  varNum,                                    /*  in --  Variable number.  */
TYPE  numRecs);                                 /*  out --  Number of written records.  */
                                                /*  TYPE  --  int* or "out int".  */
```

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

The arguments to CDFgetzVarNumRecsWritten are defined as follows:

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

4.3.55.1. Example(s)

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

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    numRecs;                                      /*  The number of written records. */
.
.
try {
    ….
   status = CDFgetzVarNumRecsWritten (id, CDFgetVarNum (id, "MY_VAR"), &numRecs);
  Or
   status = CDFgetzVarNumRecsWritten (id, CDFgetVarNum (id, "MY_VAR"), out numRecs);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.56 CDFgetzVarPadValue

```
int CDFgetzVarPadValue(                    /*  out --  Completion status code.  */
void*  id,                                 /*  in --  CDF  identifier.  */
int  varNum,                               /*  in --  Variable number.  */
TYPE  value);                              /*  out --  Pad value.  */
                                           /*  TYPE  --  void*, "out string" or "out object".  */
```

CDFgetzVarPadValue returns the pad value of the specified zVariable in a CDF. If a pad value has not been explicitly specified for the zVariable through CDFsetzVarPadValue, the informational status code **NO_PADVALUE_SPECIFIED** will be returned. Since a variable's pad value is an optional, no exception is thrown while trying to get its value if its value is not set. It's recommended to check the returned status after the method is called.

The arguments to CDFgetzVarPadValue are defined as follows:

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

varNum The zVariable number.

value The pad value.

4.3.56.1. Example(s)

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

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    padValue;                                     /*  The pad value. */
.
.
try {
   ….
   status = CDFgetzVarPadValue (id, CDFgetVarNum (id, “MY_VAR”), &padValue);
  Or
    Object padValueo;
    status = CDFgetzVarPadValue (id, CDFgetVarNum (id, “MY_VAR”), out padValueo);
    if (status !=  NO_PADVALUE_SPECIFIED)  {
 .    padValue = (int) padValueo;
    }
 .
   }
. …
 ...
} catch (CDFException ex) {
   …
}.
```

4.3.57 CDFgetzVarRecordData

```
int CDFgetzVarRecordData(                  /*  out --  Completion status code.  */
void*  id,                                 /*  in --  CDF  identifier.  */
int  varNum,                               /*  in --  Variable number.  */
int  recNum,                               /*  in --  Record number.  */
TYPE  buffer);                             /*  out --  Record data.  */
                                           /*  TYPE  --  void*, “out string”, “out string[]” or “out object”.
*/
```

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

The arguments to CDFgetzVarRecordData are defined as follows:

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

varNum	The zVariable number.
recNum	The record number.
buffer	The buffer holding the entire record data.

4.3.57.1. Example(s)

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

```
.
.
.
void*           id;                    /*  CDF identifier. */
int    varNum;                         /*  zVariable number. */
int[,] buffer1;                        /*  The data holding buffer – pre-allocation. */
int[,] buffer2;                        /*  The data holding buffer – API allocation.  */
.
.
try {
   ….
    varNum = CDFgetVarNum (id, "MY_VAR");
    buffer1 = new int[2,3];
    fixed (void* pbuffer1 = buffer1) {
      status = CDFgetzVarRecordData (id, varNum, 2, pbuffer1);
    }
    status = CDFgetzVarRecordData (id, varNum, 5, out buffer2);
 …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.58 CDFgetzVarRecVariance

```
int CDFgetzVarRecVariance(             /*  out --  Completion status code.  */
void*  id,                             /*  in --  CDF  identifier.  */
int  varNum,                           /*  in --  Variable number.  */
TYPE  recVary);                                /*  out --  Record variance.  */
                                       /*  TYPE  --  int* or "out int".  */
```

CDFgetzVarRecVariance returns the record variance of the specified zVariable in a CDF. The record variances are described in Section 2.10.

The arguments to CDFgetzVarRecVariance are defined as follows:

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

4.3.58.1. Example(s)

The following example returns the record variance for the zVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    recVary;                                       /*  The record variance. */
.
.
try {
  ….
    status = CDFgetzVarRecVariance (id, CDFgetVarNum (id, "MY_VAR"), &recVary);
  Or
    status = CDFgetzVarRecVariance (id, CDFgetVarNum (id, "MY_VAR"), out recVary);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.59 CDFgetzVarReservePercent

```
int CDFgetzVarReservePercent(                        /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  Variable number.  */
TYPE  percent);                                      /*  out --  Reserve percentage.  */
                                                     /*  TYPE  --  int* or "out int".  */
```

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

The arguments to CDFgetzVarReservePercent are defined as follows:

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

4.3.59.1. Example(s)

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

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int     percent;                                      /*  The compression reserve percentage. */
.
.
try {
  ….
    status = CDFgetzVarReservePercent (id, CDFgetVarNum (id, “MY_VAR”), &percent);
  Or
    status = CDFgetzVarReservePercent (id, CDFgetVarNum (id, “MY_VAR”), out percent);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.60 CDFgetzVarSeqData

```
int CDFgetzVarSeqData(                    /*  out --  Completion status code.  */
void*  id,                                /*  in --  CDF  identifier.  */
int  varNum,                              /*  in --  Variable number.  */
TYPE  value);                             /*  out --  Data value.  */
                                          /*  TYPE  --  void*, “out string” or “out object”.  */
```

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

The arguments to CDFgetzVarSeqData are defined as follows:

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

4.3.60.1. Example(s)

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

```
.
.
.
void*           id;                     /*  CDF identifier. */
int     varNum;                         /*  The variable number from which to read data */
int     value1, value2;                 /*  The data value. */
int[]  indices = new int[2];            /*  The indices in a record. */
int     recNum;                         /*  The record number. */
.
.
recNum = 2;
indices[0] = 0;
indices[1] = 0;
try {
  ….
  status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
  status = CDFgetzVarSeqData (id, varNum, &value1);
  object value2o;
  status = CDFgetzVarSeqData (id, varNum, out value2o);
  value2 = (int) value2o;
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.61 CDFgetzVarSeqPos

```
int CDFgetzVarSeqPos(                   /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum,                            /*  in --  Variable number.  */
TYPE  recNum,                           /*  out --  Record number.  */
TYPE2  indices);                                /*  out --  Indices in a record.  */
                                        /*  TYPE --  int* or “out int”.  */
                                        /*  TYPE2  --  int* or “out int[]”.  */
```

CDFgetzVarSeqPos returns the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDFsetzVarSeqPos method to set the current sequential value.

The arguments to CDFgetzVarSeqPos are defined as follows:

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

4.3.61.1. Example(s)

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

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    recNum;                                       /* The record number. */
int[]  indices;                                      /* The indices. */
.
.
try {
  ….
    indices = new int[2];
    fixed (int* pindices = indices) {
      status = CDFgetzVarSeqPos (id, CDFgetVarNum(id, “MY_VAR”), &recNum, pindices);
    }
  Or
    status = CDFgetzVarSeqPos (id, CDFgetVarNum(id, “MY_VAR”), out recNum, out indices);
...
} catch (CDFException ex) {
  …
}.
```

4.3.62 CDFgetzVarsMaxWrittenRecNum

```
int CDFgetzVarsMaxWrittenRecNum(                     /* out -- Completion status code. */
void* id,                                            /* in -- CDF identifier. */
TYPE recNum);                                                /* out -- Maximum record number.
*/
                                                     /* TYPE -- int* or “out int”. */
```

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

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

The arguments to CDFgetzVarsMaxWrittenRecNum are defined as follows:

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

4.3.62.1. Example(s)

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

```
.
.
void*           id;                                   /*  CDF identifier. */
int    recNum;                                        /*  The maximum record number. */
.
.
try {
  ….
    status = CDFgetzVarsMaxWrittenRecNum (id,  &recNum);
  Or
    status = CDFgetzVarsMaxWrittenRecNum (id,  out recNum);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.63 CDFgetzVarSparseRecords

```
int CDFgetzVarSparseRecords(                          /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  The variable number.  */
TYPE  sRecordsType);                                  /*  out --  The sparse records type.  */
                                                      /*  TYPE  --  int* or “out int”.  */
```

CDFgetzVarSparseRecords returns the sparse records type of the zVariable in a CDF. Refer to Section 2.12.1 for the description of sparse records.

The arguments to CDFgetzVarSparseRecords are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The variable number.
sRecordsType	The sparse records type.

4.3.63.1. Example(s)

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

```
.
.
.
```

```
void*           id;                                   /*  CDF identifier. */
int    sRecordsType;                                  /*  The sparse records type. */
.
.
try {
  ….
    status = CDFgetzVarSparseRecords (id,  CDFgetVarNum(id, “MY_VAR”), &sRecordsType);
  Or
    status = CDFgetzVarSparseRecords (id,  CDFgetVarNum(id, “MY_VAR”), out sRecordsType);…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.64 CDFhyperGetrVarData

```
int CDFhyperGetrVarData(                  /*  out --  Completion status code.  */
void*  id,                                /*  in --  CDF  identifier.  */
int  varNum,                              /*  in --  rVariable number.  */
int  recStart,                            /*  in --  Starting record number.  */
int  recCount,                            /*  in --  Number of records.  */
int  recInterval,                         /*  in --  Reading interval between records.  */
int[]  indices,                           /*  in --  Dimension indices of starting  value.  */
int[]  counts,                            /*  in --  Number of values along  each dimension.  */
int[]  intervals,                         /*  in --  Reading intervals along  each dimension.  */
TYPE  buffer);                            /*  out --  Buffer of values.  */
                                          /*  TYPE --  void*, “out string”, “out string[]” or “out object”
*/
```

CDFhyperGetrVarData is used to read one or more values for the specified rVariable. It is important to know the variable majority of the CDF before using this method because the values placed into the data buffer will be in that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The record number starts at 0, not 1. For example, if you want to read the first 5 records, the starting record number (recStart), the number of records to read (recCount), and the record interval (recInterval) should be 0, 5, and 1, respectively. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

The arguments to CDFhyperGetrVarData are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number from which to read data. This number may be determined with a call to CDFgetVarNum.
recStart	The record number at which to start reading.
recCount	The number of records to read.
recInterval	The reading interval between records (e.g., an interval of 2 means read every other record).

indices	The dimension indices (within each record) at which to start reading. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariable, this argument is ignored (but must be present).
counts	The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. For 0-dimensional rVariable, this argument is ignored (but must be present).
intervals	For each dimension, the dimension interval between reading (e.g., an interval of 2 means read every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional rVariable, this argument is ignored (but must be present).
buffer	The data holding buffer for the read values. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large to hold the values. CDFinquirerVar can be used to determine the rVariable's data type and number of elements (of that data type) at each value. If a dimensional array of strings is expected, then use **object** type.

4.3.64.1. Example(s)

The following example will read 3 records of data, starting at record number 13 (14th record), from a rVariable named Temperature The variable is a 3-dimensional array with sizes [180,91,10] and the CDF's variable majority is ROW_MAJOR. The record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF_REAL4. This example is similar to the CDFgetrVarData example except that it uses a single call to CDFhyperGetrVarData (rather than numerous calls to. CDFgetrVarData).

```
.
.
.
void*           id;                                  /*  CDF  identifier.  */
int    status;                                       /*  Returned status code.  */
float[,,,]      tmp;                                 /*  Temperature  values.  */
int    varN;                                         /*  rVariable number.  */
int    recStart  =  13;                              /*  Start record number.  */
int    recCount  =  3;                               /*  Number of records to read  */
int    recInterval  =  1;                            /*  Record interval – read every record  */
int[]  indices = new int[] {0,0,0};                  /*  Dimension indices.  */
int[]  counts = new int[] {180,91,10};               /*  Dimension counts.  */
int[]  intervals = new int[]{1,1,1};                 /*  Dimension intervals – read every value*/
.
.
try {
   varN  =  CDFgetVarNum  (id,  "Temperature");
   tmp = new float[3,180,91,10];
   fixed (void* ptmp = tmp) {
        status = CDFhyperGetrVarData (id,  varN,  recStart,  recCount,  recInterval,  indices,  counts, intervals,
                                      ptmp);
   }
  Or
   status = CDFhyperGetrVarData (id,  varN,  recStart,  recCount,  recInterval,  indices,  counts, intervals,
                                 out tmp);
…
  ...
} catch (CDFException ex) {
   …
```

}.

Note that if the CDF's variable majority had been COLUMN_MAJOR, the tmp array would have been declared float tmp[10,91,180,3] for proper indexing.

4.3.65 CDFhyperGetzVarData

```
int CDFhyperGetzVarData(                /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum,                             /* in -- zVariable number. */
int recStart,                           /* in -- Starting record number. */
int recCount,                           /* in -- Number of records. */
int recInterval,                        /* in -- Reading interval between records. */
int[] indices,                          /* in -- Dimension indices of starting value. */
int[] counts,                           /* in -- Number of values along each dimension. */
int[] intervals,                        /* in -- Reading intervals along each dimension. */
TYPE buffer);                           /* out -- Buffer of values. */
                                        /* TYPE -- void*, "out string", "out string[]" or "out
object".*/
```

CDFhyperGetzVarData is used to read one or more values for the specified zVariable. It is important to know the variable majority of the CDF before using this method because the values placed into the data buffer will be in that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The record number starts at 0, not 1. For example, if you want to read the first 5 records, the starting record number (recStart), the number of records to read (recCount), and the record interval (recInterval) should be 0, 5, and 1, respectively. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

The arguments to CDFhyperGetzVarData are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number from which to read data. This number may be determined with a call to CDFgetVarNum.
recStart	The record number at which to start reading.
recCount	The number of records to read.
recInterval	The reading interval between records (e.g., an interval of 2 means read every other record).
indices	The dimension indices (within each record) at which to start reading. Each element of indices specifies the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored (but must be present).
counts	The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. For 0-dimensional zVariable, this argument is ignored (but must be present).

intervals — For each dimension, the dimension interval between reading (e.g., an interval of 2 means read every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional zVariable, this argument is ignored (but must be present).

buffer — The data holding buffer for the read values. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large to hold the values. CDFinquirezVar can be used to determine the zVariable's data type and number of elements (of that data type) at each value. If a dimensional array of strings is expected, then use **object** type.

4.3.65.1. Example(s)

The following example will read 3 records of data, starting at record number 13 (14th record), from a zVariable named Temperature The variable is a 3-dimensional array with sizes [180,91,10] and the CDF's variable majority is ROW_MAJOR. The record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF_REAL4. This example is similar to the CDFgetzVarData example except that it uses a single call to CDFhyperGetzVarData (rather than numerous calls to. CDFgetzVarData).

```
.
.
.
void*           id;                                  /* CDF identifier. */
int    status;                                       /* Returned status code. */
float[,,,]      tmp;                                 /* Temperature values. */
int    varN;                                         /* zVariable number. */
int    recStart = 13;                                /* Start record number. */
int    recCount = 3;                                 /* Number of records to read */
int    recInterval = 1;                              /* Record interval – read every record */
int[]  indices = new int[] {0,0,0};                  /* Dimension indices. */
int[]  counts = new int[] {180,91,10};               /* Dimension counts. */
int[]  intervals = new int[]{1,1,1};                 /* Dimension intervals – read every value*/
.
.
try {
   varN = CDFgetVarNum (id, "Temperature");
   tmp = new float[3,180,91,10];
   fixed (void* ptmp = tmp) {
        status = CDFhyperGetzVarData (id, varN, recStart, recCount, recInterval, indices, counts, intervals,
                                      ptmp);
   }
 Or
   status = CDFhyperGetzVarData (id, varN, recStart, recCount, recInterval, indices, counts, intervals,
                                 out tmp);
…
  ...
} catch (CDFException ex) {
   …
}.
```

Note that if the CDF's variable majority had been COLUMN_MAJOR, the tmp array would have been declared float tmp[10,91,180,3] for proper indexing.

4.3.66 CDFhyperPutrVarData

```
int CDFhyperPutrVarData(                /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum,                             /* in -- rVariable number. */
int recStart,                           /* in -- Starting record number. */
int recCount,                           /* in -- Number of records. */
int recInterval,                        /* in -- Writing interval between records. */
int[] indices,                          /* in -- Dimension indices of starting value. */
int[] counts,                           /* in -- Number of values along each dimension. */
int[] intervals,                        /* in -- Writing intervals along each dimension. */
TYPE buffer);                           /* in -- Buffer of values. */
                                        /* TYPE -- void*, string or object. */
```

CDFhyperPutrVarData 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 method because the values in the data buffer will be written using that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The record number starts at 0, not 1. For example, if you want to write 2 records (10^{th} and 11^{th} record), the starting record number (recStart), the number of records to write (recCount), and the record interval (recInterval) should be 9, 2, and 1, respectively. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

The arguments to CDFhyperPutrVarData are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number to which write data. This number may be determined with a call to CDFgetVarNum.
recStart	The record number at which to start writing.
recCount	The number of records to write.
recInterval	The interval between records for writing (e.g., an interval of 2 means write every other record).
indices	The indices (within each record) at which to start writing. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariable this argument is ignored (but must be present).
counts	The number of values along each dimension to write. Each element of counts specifies the corresponding dimension count. For 0-dimensional rVariable this argument is ignored (but must be present).
intervals	For each dimension, the interval between values for writing (e.g., an interval of 2 means write every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional rVariable this argument is ignored (but must be present).
buffer	The data holding buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address buffer are written to the CDF.

4.3.66.1. Example(s)

The following example writes 2 records to a rVariable named LATITUDE that is a 1-dimensional array with dimension sizes [181]. The dimension variances are [VARY], and the data type is CDF_INT2. This example is similar to the CDFputrVarData example except that it uses a single call to CDFhyperPutrVarData rather than numerous calls to CDFputrVarData.

```
.
.
.
void*          id;                                  /* CDF identifier. */
int    status;                                      /* Returned status code. */
short lat;                                          /* Latitude value. */
short[,]       lats = new short[2,181];             /* Buffer of latitude values. */
int    varN;                                        /* rVariable number. */
int    recStart = 0;                                /* Record number. */
int    recCount = 2;                                /* Record counts. */
int    recInterval = 1;                             /* Record interval. */
int    indices = new int[] {0};                     /* Dimension indices. */
int    counts = new int[] {181};                    /* Dimension counts. */
int    intervals = new int[] {1};                   /* Dimension intervals. */

.
.
try {
  ….
    varN = CDFgetVarNum (id, "LATITUDE");
    for (int i= 0; i < 2; i ++)
      for (lat = -90; lat <= 90; lat ++)
        lats[i][90+lat] = (short) lat;

    fixed (void* plats = lats) {
         status = CDFhyperPutrVarData (id, varN, recStart, recCount, recInterval, indices, counts, intervals,
                                       plats);
    }
  Or
…status = CDFhyperPutrVarData (id, varN, recStart, recCount, recInterval, indices, counts, intervals, lats);

  ...
} catch (CDFException ex) {
   …
}.
```

4.3.67 CDFhyperPutzVarData

```
int CDFhyperPutzVarData(                            /* out -- Completion status code. */
void* id,                                           /* in -- CDF identifier. */
int varNum,                                         /* in -- zVariable number. */
int recStart,                                       /* in -- Starting record number. */
int recCount,                                       /* in -- Number of records. */
int recInterval,                                    /* in -- Writing interval between records. */
int[] indices,                                      /* in -- Dimension indices of starting value. */
```

```
int[] counts,                    /* in -- Number of values along each dimension. */
int[] intervals,                 /* in -- Writing intervals along each dimension. */
TYPE buffer);                    /* in -- Buffer of values. */
                                 /* TYPE -- void*, string or object. */
```

CDFhyperPutzVarData is used to write one or more values from the data holding buffer to the specified zVariable. It is important to know the variable majority of the CDF before using this method because the values in the data buffer will be written using that majority. CDFinquireCDF can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The record number starts at 0, not 1. For example, if you want to write 2 records (10^{th} and 11^{th} record), the starting record number (recStart), the number of records to write (recCount), and the record interval (recInterval) should be 9, 2, and 1, respectively. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

The arguments to CDFhyperPutzVarData are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number to which write data. This number may be determined with a call to CDFgetVarNum.
recStart	The record number at which to start writing.
recCount	The number of records to write.
recInterval	The interval between records for writing (e.g., an interval of 2 means write every other record).
indices	The indices (within each record) at which to start writing. Each element of indices specifies the corresponding dimension index. For 0-dimensional zVariable this argument is ignored (but must be present).
counts	The number of values along each dimension to write. Each element of counts specifies the corresponding dimension count. For 0-dimensional zVariable this argument is ignored (but must be present).
intervals	For each dimension, the interval between values for writing (e.g., an interval of 2 means write every other value). Each element of intervals specifies the corresponding dimension interval. For 0-dimensional zVariable this argument is ignored (but must be present).
buffer	The data holding buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address buffer are written to the CDF.

4.3.67.1. Example(s)

The following example writes 2 records to a zVariable named LATITUDE that is a 1-dimensional array with dimension sizes [181]. The dimension variances are [VARY], and the data type is CDF_INT2. This example is similar to the CDFputzVarData example except that it uses a single call to CDFhyperPutzVarData rather than numerous calls to CDFputzVarData.

```
.
.
```

```
.
void*           id;                                     /*  CDF  identifier.  */
int    status;                                          /*  Returned status code.  */
short lat;                                              /*  Latitude  value.  */
short[,]        lats = new short[2,181];                /*  Buffer of latitude  values.  */
int    varN;                                            /*  zVariable number.  */
int    recStart  =  0;                                  /*  Record number.  */
int    recCount  =  2;                                  /*  Record  counts.  */
int    recInterval  =  1;                               /*  Record interval.  */
int    indices = new int[] {0};                         /*  Dimension indices.  */
int    counts = new int[] {181};                        /*  Dimension counts.  */
int    intervals = new int[] {1};                       /*  Dimension intervals.  */

.
.
try {
  ….
    varN  =  CDFgetVarNum  (id,  "LATITUDE");
    for (int i= 0;  i < 2;  i ++)
      for (lat =  -90;  lat <=  90;  lat ++)
        lats[i][90+lat]  =  (short) lat;

    fixed (void* plats = lats) {
         status =  CDFhyperPutzVarData  (id,  varN,  recStart,  recCount,  recInterval, indices, counts, intervals,
                                          plats);
    }
  Or
…status =  CDFhyperPutzVarData  (id,  varN,  recStart,  recCount,  recInterval, indices, counts, intervals,  lats);

  ...
} catch (CDFException ex) {
   …
}.
```

4.3.68 CDFinquirerVar

```
int CDFinquirezVar(                                     /*  out --  Completion status code.  */
void*  id,                                              /*  in --  CDF  identifier.  */
int  varNum,                                            /*  in --  rVariable number.  */
out string  varName,                                    /*  out --  rVariable name.  */
TYPE  dataType,                                         /*  out --  Data type.  */
TYPE  numElements,                                      /*  out --  Number of elements  (of the data type).  */
TYPE  numDims,                                          /*  out --  Number of dimensions.  */
TYPE2  dimSizes,                                        /*  out --  Dimension sizes  */
TYPE  recVariance,                                      /*  out --  Record  variance.  */
TYPE2  dimVariances);                                   /*  out --  Dimension variances.  */
                                                        /*  TYPE  --  int* or “out int”.  */
                                                        /*  TYPE2  --  int* or “out int[]”.  */
```

CDFinquirerVar is used to inquire about the specified rVariable. This method would normally be used before reading rVariable values (with CDFgetrVarData or CDFhyperGetrVarData) to determine the data type and number of elements of that data type.

The arguments to CDFinquirezVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The number of the rVariable to inquire. This number may be determined with a call to CDFgetVarNum (see Section 4.3.40).
varName	The rVariable's name.
dataType	The data type of the rVariable. The data types are defined in Section 2.6.
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.
numDims	The number of dimensions.
dimSizes	The dimension sizes. It is a 1-dimensional array, containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional zVariables this argument is ignored (but must be present).
recVariance	The record variance. The record variances are defined in Section 2.10.
dimVariances	The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are described in Section 2.10. For 0-dimensional zVariables this argument is ignored (but a placeholder is necessary).

4.3.68.1. Example(s)

The following example returns information about a rVariable named HEAT_FLUX in a CDF.

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
string          varName;            /* rVariable name. */
int    dataType;                    /* Data type of the rVariable. */
int    numElems;                    /* Number of elements (of data type). */
int    recVary;                     /* Record variance. */
int    numDims;                     /* Number of dimensions. */
int[]  dimSizes;                    /* Dimension sizes */
int[]  dimVarys;                    /* Dimension variances */
.
.
try {
  ….
   dimSizes = new int[CDF_MAX_DIMS];
   dimVarys = new int[CDF_MAX_DIMS];
   fixed (int* pdimSizes = dimSizes, pdimVarys = dimVarys) {
```

```
      status =  CDFinquirerVar(id,  CDFgetVarNum(id,"HEAT_FLUX"),  out varName,  &dataType,
                                  &numElems,  &numDims, pdimSizes, &recVary,  pdimVarys);
    }
  Or
    status =  CDFinquirerVar(id,  CDFgetVarNum(id,"HEAT_FLUX"),  out varName,  out dataType,
                                 out numElems,  out numDims, out dimSizes, out recVary,  out dimVarys);
…
   ...
} catch (CDFException ex) {
   …
}.
```

4.3.69 CDFinquirezVar

```
int CDFinquirezVar(                                  /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  zVariable number.  */
out string  varName,                                 /*  out --  zVariable name.  */
TYPE  dataType,                                            /*  out --  Data type.  */
TYPE  numElements,                                   /*  out --  Number of elements  (of the data type).  */
TYPE  numDims,                                       /*  out --  Number of dimensions.  */
TYPE2  dimSizes,                                     /*  out --  Dimension sizes  */
TYPE  recVariance,                                   /*  out --  Record  variance.  */
TYPE2  dimVariances);                                /*  out --  Dimension variances.  */
                                                     /*  TYPE  --  int* or “out int”.  */
                                                     /*  TYPE2  --  int* or “out int[]”.  */
```

CDFinquirezVar is used to inquire about the specified zVariable. This method would normally be used before reading zVariable values (with CDFgetzVarData or CDFhyperGetzVarData) to determine the data type and number of elements of that data type.

The arguments to CDFinquirezVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The number of the zVariable to inquire. This number may be determined with a call to CDFgetVarNum (see Section 4.3.40).
varName	The zVariable's name.
dataType	The data type of the zVariable. The data types are defined in Section 2.6.
numElements	The number of elements of the data type at each zVariable value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. (Each value consists of the entire string.) For all other data types, this will always be one (1) - multiple elements at each value are not allowed for non-character data types.
numDims	The number of dimensions.

dimSizes	The dimension sizes. It is a 1-dimensional array, containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional zVariables this argument is ignored (but must be present).
recVariance	The record variance. The record variances are defined in Section 2.10.
dimVariances	The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are described in Section 2.10. For 0-dimensional zVariables this argument is ignored (but a placeholder is necessary).

4.3.69.1. Example(s)

The following example returns information about an zVariable named HEAT_FLUX in a CDF.

```
.
.
.
void*           id;                                  /* CDF identifier. */
int    status;                                       /* Returned status code. */
string          varName;                             /* zVariable name. */
int    dataType;                                     /* Data type of the zVariable. */
int    numElems;                                     /* Number of elements (of data type). */
int    recVary;                                      /* Record variance. */
int    numDims;                                      /* Number of dimensions. */
int[]  dimSizes;                                     /* Dimension sizes */
int[]  dimVarys;                                     /* Dimension variances */
.
.
try {
  ….
    dimSizes = new int[CDF_MAX_DIMS];
    dimVarys = new int[CDF_MAX_DIMS];
    fixed (int* pdimSizes = dimSizes, pdimVarys = dimVarys) {
      status =  CDFinquirezVar(id,  CDFgetVarNum(id,"HEAT_FLUX"),  out varName,  &dataType,
                                  &numElems,  &numDims, pdimSizes, &recVary,  pdimVarys);
    }
  Or
    status =  CDFinquirezVar(id,  CDFgetVarNum(id,"HEAT_FLUX"),  out varName,  out dataType,
                                 out numElems,  out numDims, out dimSizes, out recVary,  out dimVarys);
…
    ...
} catch (CDFException ex) {
   …
}.
```

4.3.70 CDFputrVarData

```
int CDFputrVarData(                                  /* out -- Completion status code. */
void*  id,                                           /* in -- CDF identifier. */
int  varNum,                                         /* in -- Variable number. */
```

```
int  recNum,                    /*  in --  Record number.  */
int[]  indices,                 /*  in --  Dimension indices.  */
TYPE  value);                   /*  in --  Data value.  */
                                /*  TYPE --  void*, string or object.  */
```

CDFputrVarData writes a single data value to the specified index, the location of the element, in the given record of the specified rVariable in a CDF.

The arguments to CDFputrVarData are defined as follows:

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

4.3.70.1. Example(s)

The following example will write two data values, the first and the fifth element, in Record 0 from rVariable "MY_VAR", a 2-dimensional (2 by 3), CDF_DOUBLE type variable, in a row-major CDF. The first put operation passes the pointer of the data value, while the second operation passes the data value as an object.

```
.
.
.
void*           id;             /*  CDF identifier. */
int    varNum;                  /*  rVariable number. */
int    recNum;                  /*  The record number. */
int[]  indices = new int[2];    /*  The dimension indices. */
double          value1, value2; /*  The data values. */
.
.
try {
  ….
  varNum = CDFgetVarNum (id, "MY_VAR");
  recNum = 0;
  indices[0] = 0;
  indices[1] = 0;
  value1 = 10.1;
  status = CDFputrVarData (id, varNum, recNum, indices, &value1);
  indices[0] = 1;
  indices[1] = 1;
  value2 = 20.2;
  status = CDFputrVarData (id, varNum, recNum, indices, value2);
  …
  ...
} catch (CDFException ex) {
   …
```

}.

4.3.71 CDFputrVarPadValue

```
int CDFputrVarPadValue(                 /* out -- Completion status code.  */
void*  id,                              /* in --  CDF  identifier.  */
int  varNum,                            /* in --  Variable number.  */
TYPE  value);                           /* in --  Pad value.  */
                                        /* TYPE  --  void*, string or object */
```

CDFputrVarPadValue specifies the pad value for the specified rVariable in a CDF. A 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.

The arguments to CDFputrVarPadValue are defined as follows:

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

4.3.71.1. Example(s)

The following example sets the pad value to –9999 for rVariable “MY_VAR”, a CDF_INT4 type variable, and “*****” for another rVariable “MY_VAR2”, a CDF_CHAR type with a number of elements of five (5), in a CDF.

```
.
.
.
void*           id;                     /* CDF identifier. */
int    padValue1 = -9999;               /* An int pad value. */
string          padValue2 = “*****”;     /* A string pad value. */`
.
.
try {
   ….
 status = CDFputrVarPadValue (id, CDFgetVarNum (id, “MY_VAR”), padValue1);
 Or
 status = CDFputrVarPadValue (id, CDFgetVarNum (id, “MY_VAR”), &padValue1);

 status = CDFputrVarPadValue (id, CDFgetVarNum (id, “MY_VAR2”), padValue2);
 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.3.72 CDFputrVarRecordData

```
int CDFputrVarRecordData(                    /* out -- Completion status code. */
void* id,                                    /* in -- CDF identifier. */
int varNum,                                  /* in -- Variable number. */
int recNum,                                  /* in -- Record number. */
TYPE buffer);                                /* in -- Record data. */
                                             /* TYPE -- void*, string, string[] or
                                          object.*/"
```

CDFputrVarRecordData writes an entire record at a given record number for the specified rVariable in a CDF. The buffer should hold the entire data values for the variable. The data values in the buffer should be in the order that corresponds to the variable majority defined for the CDF.

The arguments to CDFputrVarRecordData are defined as follows:

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

4.3.72.1. Example(s)

The following example will write one full record (numbered 2) from rVariable "MY_VAR", a 2-dimension (2 by 3), CDF_INT4 type variable, in a CDF. The variable's dimension variances are all VARY.

```
.
.
.
void*          id;                           /* CDF identifier. */
int    varNum;                               /* rVariable number. */
int[,] buffer = new[2,3] {{1,2,3},{4,5,6}};  /* The data holding buffer. */
.
.
try {
  ….
   varNum = CDFvarNum(id,"MY_VAR");
   status = CDFputrVarRecordData (id, varNum, 2, buffer);
  Or
     fixed (void* pbuffer = buffer) {
       status = CDFputrVarRecordData (id, varNum, 2, pbuffer) ;
     }
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.73 CDFputrVarSeqData

```
int CDFputrVarSeqData(                  /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum,                             /* in -- Variable number. */
TYPE value);                            /* in -- Data value. */
                                        /* TYPE -- void*, string or object. */
```

CDFputrVarSeqData writes one value to the specified rVariable in a CDF at the current sequential value (position) for that variable. After the write, the current sequential value is automatically incremented to the next value. Use CDFsetrVarSeqPos method to set the current sequential value (position).

The arguments to CDFputrVarSeqData are defined as follows:

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

4.3.73.1. Example(s)

The following example will write two data values starting at record number 2 from a 2-dimensional rVariable whose data type is CDF_INT4. The first write will pass in a pointer from the data value, while the second write will pass in the data value object directly.

```
.
.
.
void*           id;                     /* CDF identifier. */
int    varNum;                          /* The variable number. */
int    value1, value2;                  /* The data value. */
int[]  indices = new int[2];            /* The indices in a record. */
int    recNum;                          /* The record number. */
.
.
recNum = 2;
indices[0] = 1;
indices[1] = 2;
try {
  ….
  value1 = 10;
  value2 = -20.
  status = CDFsetrVarSeqPos (id, varNum, recNum, indices);
  status = CDFputrVarSeqData (id, varNum, &value1);
  status = CDFputrVarSeqData (id, varNum, value2);
…
  ...
```

```
} catch (CDFException ex) {
  …
}.
```

4.3.74 CDFputzVarData

```
int CDFputzVarData(                    /* out -- Completion status code.  */
void* id,                              /* in -- CDF identifier.  */
int varNum,                            /* in -- Variable number.  */
int recNum,                            /* in -- Record number.  */
int[] indices,                         /* in -- Dimension indices.  */
TYPE value);                           /* in -- Data value.  */
                                       /* TYPE -- void*, string or object.  */
```

CDFputzVarData writes a single data value to the specified index, the location of the element, in the given record of the specified zVariable in a CDF.

The arguments to CDFputzVarData are defined as follows:

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

4.3.74.1. Example(s)

The following example will write two data values, the first and the fifth element, in Record 0 from zVariable "MY_VAR", a 2-dimensional (2 by 3), CDF_DOUBLE type variable, in a row-major CDF. The first put operation passes the pointer of the data value, while the second operation passes the data value as an object.

```
.
.
.
void*          id;                     /* CDF identifier. */
int    varNum;                         /* zVariable number. */
int    recNum;                         /* The record number. */
int[]  indices = new int[2];           /* The dimension indices. */
double         value1, value2;         /* The data values. */
.
.
try {
 ….
 varNum = CDFgetVarNum (id, "MY_VAR");
 recNum = 0;
```

```
  indices[0] = 0;
  indices[1] = 0;
  value1 = 10.1;
  status = CDFputzVarData (id, varNum, recNum, indices, &value1);
  indices[0] = 1;
  indices[1] = 1;
  value2 = 20.2;
  status = CDFputzVarData (id, varNum, recNum, indices, value2);
  …
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.75 CDFputzVarPadValue

```
int CDFputzVarPadValue(                 /* out -- Completion status code.  */
void*  id,                              /* in --  CDF  identifier.  */
int  varNum,                            /* in --  Variable number.  */
TYPE  value);                           /* in --  Pad value.  */
                                        /* TYPE --  void*, string or object */
```

CDFputzVarPadValue specifies the pad value for the specified zVariable in a CDF. A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values.

The arguments to CDFputzVarPadValue are defined as follows:

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

4.3.75.1. Example(s)

The following example sets the pad value to –9999 for zVariable “MY_VAR”, a CDF_INT4 type variable, and “*****” for another zVariable “MY_VAR2”, a CDF_CHAR type with a number of elements of five (5), in a CDF.

```
.
.
.
void*          id;                      /* CDF identifier. */
int    padValue1 = -9999;               /* An int pad value. */
string         padValue2 = “*****”;     /* A string pad value. */`
.
.
try {
   ….
```

```
 status = CDFputzVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), padValue1);
Or
 status = CDFputzVarPadValue (id, CDFgetVarNum (id, "MY_VAR"), &padValue1);

 status = CDFputzVarPadValue (id, CDFgetVarNum (id, "MY_VAR2"), padValue2);
 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.3.76 CDFputzVarRecordData

```
int CDFputzVarRecordData(                  /*  out --  Completion status code.  */
void*  id,                                 /*  in --  CDF  identifier.  */
int  varNum,                               /*  in --  Variable number.  */
int  recNum,                               /*  in --  Record number.  */
TYPE  buffer);                             /*  in --  Record data.  */
                                           /*  TYPE --  void*, string, string[] or object. /"
```

CDFputzVarRecordData writes an entire record at a given record number for the specified zVariable in a CDF. The buffer should hold the entire data values for the variable. The data values in the buffer should be in the order that corresponds to the variable majority defined for the CDF.

The arguments to CDFputzVarRecordData are defined as follows:

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

4.3.76.1. Example(s)

The following example will write one full record (numbered 2) from zVariable "MY_VAR", a 2-dimension (2 by 3), CDF_INT4 type variable, in a CDF. The variable's dimension variances are all VARY.

```
.
.
.
void*            id;                       /*  CDF identifier. */
int    varNum;                             /*  zVariable number. */
int[,] buffer = new[2,3] {{1,2,3},{4,5,6}}; /*  The data holding buffer. */
.
.
try {
  ….
```

```
    varNum = CDFvarNum(id,"MY_VAR");
    status = CDFputzVarRecordData (id, varNum, 2, buffer);
  Or
     fixed (void* pbuffer = buffer) {
       status = CDFputzVarRecordData (id, varNum, 2, pbuffer) ;
     }
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.77 CDFputzVarSeqData

```
int CDFputzVarSeqData(                  /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum,                            /*  in --  Variable number.  */
TYPE  value);                           /*  in --  Data value.  */
                                        /*  TYPE --  void*, string or object.  */
```

CDFputzVarSeqData writes one value to the specified zVariable in a CDF at the current sequential value (position) for that variable. After the write, the current sequential value is automatically incremented to the next value. Use CDFsetzVarSeqPos method to set the current sequential value (position).

The arguments to CDFputzVarSeqData are defined as follows:

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

4.3.77.1. Example(s)

The following example will write two data values starting at record number 2 from a 2-dimensional zVariable whose data type is CDF_INT4. The first write will pass in a pointer from the data value, while the second write will pass in the data value object directly.

```
.
.
.
void*          id;                      /*  CDF identifier. */
int    varNum;                          /*  The variable number. */
int    value1, value2;                  /*  The data value. */
int[]  indices = new int[2];            /*  The indices in a record. */
int    recNum;                          /*  The record number. */
.
.
recNum = 2;
```

```
indices[0] = 1;
indices[1] = 2;
try {
  ….
  value1 = 10;
  value2 = -20.
  status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
  status = CDFputzVarSeqData (id, varNum, &value1);
  status = CDFputzVarSeqData (id, varNum, value2);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.78 CDFrenamerVar

```
int CDFrenamerVar(                      /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  varNum,                            /*  in --  rVariable number.  */
string  varName);                       /*  in --  New  name.  */
```

CDFrenamerVar is used to rename an existing rVariable. A variable (rVariable or zVariable) with the same name must not already exist in the CDF.

The arguments to CDFrenamerVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The number of the rVariable to rename. This number may be determined with a call to CDFgetVarNum.
varName	The new rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.

4.3.78.1. Example(s)

In the following example the rVariable named TEMPERATURE is renamed to TMP (if it exists). Note that if CDFgetVarNum returns a value less than zero (0) then that value is not an rVariable number but rather an error code.

```
.
.
.
void*          id;                      /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
int    varNum;                          /*  zVariable number.  */
.
.
try {
```

```
  ….
  varNum = CDFgetVarNum (id, "TEMPERATURE");
  status = CDFrenamerVar (id, varNum, "TMP");
  …
  ...
} catch (CDFException ex) {
  …
}.
```

4.3.79 CDFrenamezVar

```
int CDFrenamezVar(                      /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int varNum,                             /* in -- zVariable number. */
string varName);                        /* in -- New name. */
```

CDFrenamezVar is used to rename an existing zVariable. A variable (rVariable or zVariable) with the same name must not already exist in the CDF.

The arguments to CDFrenamezVar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The number of the zVariable to rename. This number may be determined with a call to CDFgetVarNum.
varName	The new zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.

4.3.79.1. Example(s)

In the following example the zVariable named TEMPERATURE is renamed to TMP (if it exists). Note that if CDFgetVarNum returns a value less than zero (0) then that value is not an zVariable number but rather an error code.

```
.
.
.
void*          id;                      /* CDF identifier. */
int    status;                          /* Returned status code. */
int    varNum;                          /* zVariable number. */
.
.
try {
  ….
  varNum = CDFgetVarNum (id, "TEMPERATURE");
  status = CDFrenamezVar (id, varNum, "TMP");
  …
  ...
} catch (CDFException ex) {
```

```
  …
}.
```

4.3.80 CDFsetrVarAllocBlockRecords

```
int CDFsetrVarAllocBlockRecords(                /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  varNum,                                    /*  in --  Variable number.  */
int  firstRec,                                  /*  in --  First record number.  */
int  lastRec);                                  /*  in --  Last record number.  */
```

CDFsetrVarAllocBlockRecords specifies a range of records to be allocated (not written) for the specified rVariable in a CDF. This operation is only applicable to uncompressed rVariable in single-file CDFs. Refer to the CDF User's Guide for the descriptions of allocating variable records.

The arguments to CDFsetrVarAllocBlockRecords are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
firstRec	The first record number to allocate.
lastRec	The last record number to allocate.

4.3.80.1. Example(s)

The following example allocates 10 records, from record numbered 10 to 19, for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*            id;                            /*  CDF identifier. */
int     firstRec, lastRec;                      /*  The first/last record numbers.  */
.
.
firstRec = 10;
lastRec = 19;
try {
  ….
  status = CDFsetrVarAllocBlockRecords (id, CDFgetVarNum(id, "MY_VAR"), firstRec, lastRec);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.81 CDFsetrVarAllocRecords

```
int CDFsetrVarAllocRecords(                 /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
int  numRecs);                              /*  in --  Number of records.  */
```

CDFsetrVarAllocRecords specifies a number of records to be allocated (not written) for the specified rVariable in a CDF. The records are allocated beginning at record number zero (0). This operation is only applicable to uncompressed rVariable in single-file CDFs. Refer to the CDF User's Guide for the descriptions of allocating variable records.

The arguments to CDFsetrVarAllocRecords are defined as follows:

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

4.3.81.1. Example(s)

The following example allocates 100 records, from record numbered 0 to 99, for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*            id;                        /*  CDF identifier. */
int    numRecs;                             /*  The number of records.  */
.
.
numRecs = 100;
try {
  ….
  status = CDFsetrVarAllocRecords (id, CDFgetVarNum(id, "MY_VAR"), numRecs);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.82 CDFsetrVarBlockingFactor

```
int CDFsetrVarBlockingFactor(               /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
int  bf);                                   /*  in --  Blocking factor.  */
```

CDFsetrVarBlockingFactor specifies the blocking factor (number of records allocated) for the specified rVariable in a CDF. Refer to the CDF User's Guide for a description of the blocking factor.

The arguments to CDFsetrVarBlockingFactor are defined as follows:

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

4.3.82.1. Example(s)

The following example sets the blocking factor to 100 records for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    bf;                                           /*  The blocking factor. */
.
.
bf = 100;
try {
  ….
  status = CDFsetrVarBlockingFactor (id, CDFgetVarNum(id, "MY_VAR"), bf);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.83 CDFsetrVarCacheSize

```
int CDFsetrVarCacheSize(                            /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  varNum,                                        /*  in --  Variable number.  */
int  numBuffers);                                   /*  in --  Number of cache buffers.  */
```

CDFsetrVarCacheSize specifies the number of cache buffers being for the rVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User's Guide for description about caching scheme used by the CDF library.

The arguments to CDFsetrVarCacheSize are defined as follows:

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

varNum	The rVariable number.
numBuffers	The number of cache buffers.

4.3.83.1. Example(s)

The following example sets the number of cache buffers to 10 for rVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    numBuffers;                                    /*  The number of cache buffers. */
.
.
numBuffers = 10;
try {
  ….
  status = CDFsetrVarCacheSize (id, CDFgetVarNum(id, “MY_VAR”), numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.84 CDFsetrVarCompression

```
int CDFsetrVarCompression(                            /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
int  cType,                                           /*  in --  Compression type.  */
int[]  cParms);                                       /*  in --  Compression parameters.  */
```

CDFsetrVarCompression specifies the compression type/parameters for the specified rVariable in a CDF. Refer to Section 2.11 for a description of the CDF supported compression types/parameters.

The arguments to CDFsetrVarCompression are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
cType	The compression type.
cParms	The compression parameters.

4.3.84.1. Example(s)

The following example sets the compression to GZIP.6 for rVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    cType;                                         /*  The compression type. */
int[]  cParms = new int[1];                           /*  The compression parameters. */
.
.
cType = GZIP_COMPRESSION;
cParms[0] = 6;
try {
  ….
  status = CDFsetrVarCompression (id, CDFgetVarNum (id, “MY_VAR”), cType, cParms);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.85 CDFsetrVarDataSpec

```
int CDFsetrVarDataSpec(                               /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
int  dataType)                                        /*  in --  Data type.  */
```

CDFsetrVarDataSpec respecifies the data type of the specified rVariable in a CDF. The variable’s data type cannot be changed if the new data type is not equivalent (type having a different data size) to the old data type and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent. Refer to the CDF User’s Guide for equivalent data types.

The arguments to CDFsetrVarDataSpec are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
dataType	The new data type.

4.3.85.1. Example(s)

The following example respecifies the data type to CDF_INT2 (from its original CDF_UINT2) for rVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    dataType;                                      /*  The data type. */
.
.
dataType = CDF_INT2;
try {
  ….
  status = CDFsetrVarDataSpec (id, CDFgetVarNum (id, “MY_VAR”), dataType);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.86 CDFsetrVarDimVariances

```
int CDFsetrVarDimVariances(                          /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  Variable number.  */
int[]  dimVarys);                                    /*  in --  Dimension variances.  */
```

CDFsetrVarDimVariances respecifies the dimension variances of the specified rVariable in a CDF. For 0-dimensional rVariable, this operation is not applicable. The dimension variances are described in Section 2.10.

The arguments to CDFsetrVarDimVariances are defined as follows:

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

4.3.86.1. Example(s)

The following example resets the dimension variances to true (VARY) and true (VARY) for rVariable “MY_VAR”, a 2-dimensional variable, in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    varNum;                                       /*  rVariable number. */
int[]  dimVarys = new int[] {VARY, VARY};            /*  The dimension variances. */
.
.
```

```
try {
  ….
  varNum = CDFgetVarNum (id, “MY_VAR”);
  status = CDFsetrVarDimVariances (id, varNum, dimVarys);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.87 CDFsetrVarInitialRecs

```
int CDFsetrVarInitialRecs(                  /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
int  numRecs);                              /*  in --  Number of records.  */
```

CDFsetrVarInitialRecs specifies a number of records to initially write to the specified rVariable in a 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.

The arguments to CDFsetrVarInitialRecs are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The rVariable number.
numRecs	The initially written records.

4.3.87.1. Example(s)

The following example writes the initial 100 records to rVariable “MY_VAR” in a CDF.

```
.
.
..
void*          id;                          /*  CDF identifier. */
int    varNum;                              /*  rVariable number. */
int    numRecs                              /*  The number of records. */
.
.
try {
   …
  varNum = CDFgetVarNum (id, “MY_VAR”);
  numRecs = 100;
  status = CDFsetrVarInitialRecs (id, varNum, numRecs);
  …
```

```
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.88 CDFsetrVarRecVariance

```
int CDFsetrVarRecVariance(                  /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  varNum,                                /*  in --  Variable number.  */
int  recVary);                              /*  in --  Record variance.  */
```

CDFsetrVarRecVariance specifies the record variance of the specified rVariable in a CDF. The record variances are described in Section 2.10.

The arguments to CDFsetrVarRecVariance are defined as follows:

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

4.3.88.1. Example(s)

The following example sets the record variance to VARY (from NOVARY) for rVariable “MY_VAR” in a CDF.

```
.
.
.
void*            id;                        /*  CDF identifier. */
int    recVary;                             /*  The record variance. */
.
.
recVary = VARY;
try {
   ….
  status = CDFsetrVarRecVariance (id, CDFgetVarNum (id, “MY_VAR”), recVary);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.89 CDFsetrVarReservePercent

```
int CDFsetrVarReservePercent(                   /* out -- Completion status code. */
void* id,                                       /* in -- CDF identifier. */
int varNum,                                     /* in -- Variable number. */
int percent);                                   /* in -- Reserve percentage. */
```

CDFsetrVarReservePercent specifies the compression reserve percentage being used for the specified rVariable in a CDF. This operation only applies to compressed rVariables. Refer to the CDF User's Guide for a description of the reserve scheme used by the CDF library.

The arguments to CDFsetrVarReservePercent are defined as follows:

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

4.3.89.1. Example(s)

The following example sets the reserve percentage to 10 for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                             /* CDF identifier. */
int     percent;                                /* The reserve percentage. */
.
.
percent = 10;
try {
    ….
  status = CDFsetrVarReservePercent (id, CDFgetVarNum (id, "MY_VAR"), percent);
  …
  ...
} catch (CDFException ex) {
   …
}.
.
```

4.3.90 CDFsetrVarsCacheSize

```
int CDFsetrVarsCacheSize(                       /* out -- Completion status code. */
void* id,                                       /* in -- CDF identifier. */
int numBuffers);                                /* in -- Number of cache buffers. */
```

CDFsetrVarsCacheSize specifies the number of cache buffers to be used for all of the rVariable files in a 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.

The arguments to CDFsetrVarsCacheSize are defined as follows:

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

4.3.90.1. Example(s)

The following example sets the number of cache buffers to 10 for all rVariables in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    numBuffers;                                    /*  The number of cache buffers. */
.
.
numBuffers = 10;
try {
   ….
 status = CDFsetrVarsCacheSize (id, numBuffers);
 …
 ...
} catch (CDFException ex) {
  …
}.
```

4.3.91 CDFsetrVarSeqPos

```
int CDFsetrVarSeqPos(                                 /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
int  recNum,                                          /*  in --  Record number.  */
int[]  indices);                                      /*  in --  Indices in a record.  */
```

CDFsetrVarSeqPos specifies the current sequential value (position) for sequential access for the specified rVariable in a CDF. Note that a current sequential value is maintained for each rVariable individually. Use CDFgetrVarSeqPos method to get the current sequential value.

The arguments to CDFsetrVarSeqPos are defined as follows:

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

varNum — The rVariable number.

recNum — The rVariable record number.

indices — The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional rVariable, this argument is ignored, but must be presented.

4.3.91.1. Example(s)

The following example sets the current sequential value to the first value element in record number 2 for a rVariable, a 2-dimensional variable, in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int     varNum;                                       /*  The variable number. */
int     recNum;                                       /*  The record number. */
int[]  indices = new int[2];                          /*  The indices. */
.
.
recNum = 2;
indices[0] = 0;
indices[1] = 0;
try {
  status = CDFsetrVarSeqPos (id, varNum, recNum, indices);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.92 CDFsetrVarSparseRecords

```
int CDFsetrVarSparseRecords(                          /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  The variable number.  */
int  sRecordsType);                                   /*  in --  The sparse records type.  */
```

CDFsetrVarSparseRecords specifies the sparse records type of the specified rVariable in a CDF. Refer to Section 2.12.1 for the description of sparse records.

The arguments to CDFsetrVarSparseRecords are defined as follows:

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

varNum — The rVariable number.

sRecordsType — The sparse records type.

4.3.92.1. Example(s)

The following example sets the sparse records type to PAD_SPARSERECORDS from its original type for rVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int    sRecordsType;                                  /*  The sparse records type. */
.
.
sRecordsType = PAD_ SPARSERECORDS;
try {
  status = CDFsetrVarSparseRecords (id,  CDFgetVarNum(id, "MY_VAR"), sRecordsType);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.93 CDFsetzVarAllocBlockRecords

```
int CDFsetzVarAllocBlockRecords(                    /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  varNum,                                        /*  in --  Variable number.  */
int  firstRec,                                      /*  in --  First record number.  */
int  lastRec);                                      /*  in --  Last record number.  */
```

CDFsetzVarAllocBlockRecords specifies a range of records to be allocated (not written) for the specified zVariable in a CDF. This operation is only applicable to uncompressed zVariable in single-file CDFs. Refer to the CDF User's Guide for the descriptions of allocating variable records.

The arguments to CDFsetzVarAllocBlockRecords are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number.
firstRec	The first record number to allocate.
lastRec	The last record number to allocate.

4.3.93.1. Example(s)

The following example allocates 10 records, from record numbered 10 to 19, for zVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int     firstRec, lastRec;                            /*  The first/last record numbers.  */
.
.
firstRec = 10;
lastRec = 19;
try {
  ….
  status = CDFsetzVarAllocBlockRecords (id, CDFgetVarNum(id, “MY_VAR”), firstRec, lastRec);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.94 CDFsetzVarAllocRecords

```
int CDFsetzVarAllocRecords(                           /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
int  numRecs);                                        /*  in --  Number of records.  */
```

CDFsetzVarAllocRecords specifies a number of records to be allocated (not written) for the specified zVariable in a CDF. The records are allocated beginning at record number zero (0). This operation is only applicable to uncompressed zVariable in single-file CDFs. Refer to the CDF User’s Guide for the descriptions of allocating variable records.

The arguments to CDFsetzVarAllocRecords are defined as follows:

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

4.3.94.1. Example(s)

The following example allocates 100 records, from record numbered 0 to 99, for zVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int     numRecs;                                      /*  The number of records.  */
.
.
```

```
numRecs = 100;
try {
  ….
  status = CDFsetzVarAllocRecords (id, CDFgetVarNum(id, “MY_VAR”), numRecs);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.95 CDFsetzVarBlockingFactor

```
int CDFsetzVarBlockingFactor(                  /*  out --  Completion status code.  */
void*  id,                                     /*  in --  CDF  identifier.  */
int  varNum,                                   /*  in --  Variable number.  */
int  bf);                                      /*  in --  Blocking factor.  */
```

CDFsetzVarBlockingFactor specifies the blocking factor (number of records allocated) for the specified zVariable in a CDF. Refer to the CDF User’s Guide for a description of the blocking factor.

The arguments to CDFsetzVarBlockingFactor are defined as follows:

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

4.3.95.1. Example(s)

The following example sets the blocking factor to 100 records for zVariable “MY_VAR” in a CDF.

```
.
.
.
void*          id;                             /*  CDF identifier. */
int    bf;                                     /*  The blocking factor. */
.
.
bf = 100;
try {
  ….
  status = CDFsetzVarBlockingFactor (id, CDFgetVarNum(id, “MY_VAR”), bf);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.96 CDFsetzVarCacheSize

```
int CDFsetzVarCacheSize(                    /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int varNum,                                 /* in -- Variable number. */
int numBuffers);                            /* in -- Number of cache buffers. */
```

CDFsetzVarCacheSize specifies the number of cache buffers being for the zVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User's Guide for description about caching scheme used by the CDF library.

The arguments to CDFsetzVarCacheSize are defined as follows:

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

4.3.96.1. Example(s)

The following example sets the number of cache buffers to 10 for zVariable "MY_VAR" in a CDF.

```
.
.
.
void*           id;                         /* CDF identifier. */
int    numBuffers;                          /* The number of cache buffers. */
.
.
numBuffers = 10;
try {
  ….
  status = CDFsetzVarCacheSize (id, CDFgetVarNum(id, "MY_VAR"), numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.97 CDFsetzVarCompression

```
int CDFsetzVarCompression(                  /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
```

```
int  varNum,                                    /*  in --  Variable number.  */
int  cType,                                     /*  in --  Compression type.  */
int[]  cParms);                                 /*  in --  Compression parameters.  */
```

CDFsetzVarCompression specifies the compression type/parameters for the specified zVariable in a CDF. Refer to Section 2.11 for a description of the CDF supported compression types/parameters.

The arguments to CDFsetzVarCompression are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
varNum	The zVariable number.
cType	The compression type.
cParms	The compression parameters.

4.3.97.1. Example(s)

The following example sets the compression to GZIP.6 for zVariable “MY_VAR” in a CDF.

```
.
.
.
void*            id;                            /*  CDF identifier. */
int    cType;                                   /*  The compression type. */
int[]  cParms = new int[1];                     /*  The compression parameters. */
.
.
cType = GZIP_COMPRESSION;
cParms[0] = 6;
try {
  ….
  status = CDFsetzVarCompression (id, CDFgetVarNum (id, “MY_VAR”), cType, cParms);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.98 CDFsetzVarDataSpec

```
int CDFsetzVarDataSpec(                         /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  varNum,                                    /*  in --  Variable number.  */
int  dataType)                                  /*  in --  Data type.  */
```

CDFsetzVarDataSpec respecifies the data type of the specified zVariable in a CDF. The variable’s data type cannot be changed if the new data type is not equivalent (type having a different data size) to the old data type and any values

(including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent. Refer to the CDF User's Guide for equivalent data types.

The arguments to CDFsetzVarDataSpec are defined as follows:

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

4.3.98.1. Example(s)

The following example respecifies the data type to CDF_INT2 (from its original CDF_UINT2) for zVariable "MY_VAR" in a CDF.

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    dataType;                                     /*  The data type. */
.
.
dataType = CDF_INT2;
try {
  ….
  status = CDFsetzVarDataSpec (id, CDFgetVarNum (id, "MY_VAR"), dataType);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.99 CDFsetzVarDimVariances

```
int CDFsetzVarDimVariances(                          /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  Variable number.  */
int[]  dimVarys);                                    /*  in --  Dimension variances.  */
```

CDFsetzVarDimVariances respecifies the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. The dimension variances are described in Section 2.10.

The arguments to CDFsetzVarDimVariances are defined as follows:

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

varNum	The zVariable number.
dimVarys	The dimension variances.

4.3.99.1. Example(s)

The following example resets the dimension variances to true (VARY) and true (VARY) for zVariable "MY_VAR", a 2-dimensional variable, in a CDF.

```
.
.
.
void*          id;                                   /*  CDF identifier. */
int    varNum;                                       /*  zVariable number. */
int[]  dimVarys = new int[] {VARY, VARY};            /*  The dimension variances. */
.
.
try {
  ….
  varNum = CDFgetVarNum (id, "MY_VAR");
  status = CDFsetzVarDimVariances (id, varNum, dimVarys);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.100 CDFsetzVarInitialRecs

```
int CDFsetzVarInitialRecs(                           /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  varNum,                                         /*  in --  Variable number.  */
int  numRecs);                                       /*  in --  Number of records.  */
```

CDFsetzVarInitialRecs specifies a number of records to initially write to the specified zVariable in a CDF. The records are written beginning at record number 0 (zero). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records.

The arguments to CDFsetzVarInitialRecs are defined as follows:

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

4.3.100.1. Example(s)

The following example writes the initial 100 records to zVariable “MY_VAR” in a CDF.

```
.
.
..
void*           id;                                   /* CDF identifier. */
int    varNum;                                        /* zVariable number. */
int    numRecs                                        /* The number of records. */
.
.
try {
   …
  varNum = CDFgetVarNum (id, “MY_VAR”);
  numRecs = 100;
  status = CDFsetzVarInitialRecs (id, varNum, numRecs);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.101 CDFsetzVarRecVariance

```
int CDFsetzVarRecVariance(                          /* out -- Completion status code. */
void* id,                                           /* in -- CDF identifier. */
int varNum,                                         /* in -- Variable number. */
int recVary);                                       /* in -- Record variance. */
```

CDFsetzVarRecVariance specifies the record variance of the specified zVariable in a CDF. The record variances are described in Section 2.10.

The arguments to CDFsetzVarRecVariance are defined as follows:

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

4.3.101.1. Example(s)

The following example sets the record variance to VARY (from NOVARY) for zVariable “MY_VAR” in a CDF.

```
.
.
.
```

```
void*           id;                                   /*  CDF identifier. */
int     recVary;                                      /*  The record variance. */
.
.
recVary = VARY;
try {
    ….
  status = CDFsetzVarRecVariance (id, CDFgetVarNum (id, “MY_VAR”), recVary);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.102 CDFsetzVarReservePercent

```
int CDFsetzVarReservePercent(                         /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  varNum,                                          /*  in --  Variable number.  */
int  percent);                                        /*  in --  Reserve percentage.  */
```

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

The arguments to CDFsetzVarReservePercent are defined as follows:

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

4.3.102.1. Example(s)

The following example sets the reserve percentage to 10 for zVariable “MY_VAR” in a CDF.

```
.
.
.
void*           id;                                   /*  CDF identifier. */
int     percent;                                      /*  The reserve percentage. */
.
.
percent = 10;
try {
    ….
  status = CDFsetzVarReservePercent (id, CDFgetVarNum (id, “MY_VAR”), percent);
  …
```

```
  ...
} catch (CDFException ex) {
   …
}.
.
```

4.3.103 CDFsetzVarsCacheSize

```
int CDFsetzVarsCacheSize(                    /*  out --  Completion status code.  */
void*  id,                                   /*  in --  CDF  identifier.  */
int  numBuffers);                            /*  in --  Number of cache buffers.  */
```

CDFsetzVarsCacheSize specifies the number of cache buffers to be used for all of the zVariable files in a 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.

The arguments to CDFsetzVarsCacheSize are defined as follows:

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

4.3.103.1. Example(s)

The following example sets the number of cache buffers to 10 for all zVariables in a CDF.

```
.
.
.
void*          id;                           /*  CDF identifier. */
int    numBuffers;                           /*  The number of cache buffers. */
.
.
numBuffers = 10;
try {
   ….
  status = CDFsetzVarsCacheSize (id, numBuffers);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.104 CDFsetzVarSeqPos

```
int CDFsetzVarSeqPos(                    /* out -- Completion status code. */
void* id,                                /* in -- CDF identifier. */
int varNum,                              /* in -- Variable number. */
int recNum,                              /* in -- Record number. */
int[] indices);                          /* in -- Indices in a record. */
```

CDFsetzVarSeqPos specifies the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDFgetzVarSeqPos method to get the current sequential value.

The arguments to CDFsetzVarSeqPos are defined as follows:

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

4.3.104.1. Example(s)

The following example sets the current sequential value to the first value element in record number 2 for a zVariable, a 2-dimensional variable, in a CDF.

```
.
.
.
void*          id;                       /* CDF identifier. */
int    varNum;                           /* The variable number. */
int    recNum;                           /* The record number. */
int[]  indices = new int[2];             /* The indices. */
.
.
recNum = 2;
indices[0] = 0;
indices[1] = 0;
try {
  status = CDFsetzVarSeqPos (id, varNum, recNum, indices);
…
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.105 CDFsetzVarSparseRecords

```
int CDFsetzVarSparseRecords(             /* out -- Completion status code. */
void* id,                                /* in -- CDF identifier. */
```

```
int  varNum,                                    /*  in --  The variable number.  */
int  sRecordsType);                             /*  in --  The sparse records type.  */
```

CDFsetzVarSparseRecords specifies the sparse records type of the specified zVariable in a CDF. Refer to Section 2.12.1 for the description of sparse records.

The arguments to CDFsetzVarSparseRecords are defined as follows:

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

4.3.105.1. Example(s)

The following example sets the sparse records type to PAD_SPARSERECORDS from its original type for zVariable "MY_VAR" in a CDF.

```
.
.
.
void*            id;                            /*  CDF identifier. */
int    sRecordsType;                            /*  The sparse records type. */
.
.
sRecordsType = PAD_ SPARSERECORDS;
try {
  status = CDFsetzVarSparseRecords (id,  CDFgetVarNum(id, "MY_VAR"), sRecordsType);
  …
  ...
} catch (CDFException ex) {
   …
}.
```

4.3.106 CDFvarClose[9]

```
int CDFvarClose(                                /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  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.

[9] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFcloserVar is the preferred function for it.

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 CDFcreateCDF) or CDFopen.
varNum	The variable number for the open rVariable's file. This identifier must have been initialized by a call to CDFgetVarNum.

4.3.106.1. Example(s)

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

```
.
.
.
void*          id;                    /* CDF identifier. */
int     status;                       /* Returned status code. */
.
.
try {

   status =  CDFvarClose  (id, CDFvarNum (id, "Flux"));

} catch (CDFException ex) {
  …
}.
```

4.3.107 CDFvarCreate[10]

```
int CDFvarCreate(
void*  id,                            /* out --  Completion status code.  */
string  varName,                      /* in --  CDF  identifier.  */
int  dataType,                        /* in --  rVariable name.  */
int  numElements,                     /* in --  Data type.  */
int  recVariance,                     /* in --  Number of elements  (of the data type).  */
int[]  dimVariances,                  /* in --  Record  variance.  */
TYPE  varNum);                        /* in --  Dimension variances.  */
                                              /* out --  rVariable number.  */
                                      /* TYPE --  int* or "out int".  */
```

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.

[10] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFcreaterVar is the preferred function for it.

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 CDFcreateCDF) or CDFopen.
varName	The name of the rVariable to create. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
dataType	The data type of the new rVariable. Specify one of the data types defined in Section 2.6.
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 2.10.
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 2.10. 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.

4.3.107.1. Example(s)

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

```
.
.
.
void*          id;                                          /* CDF identifier. */
Int    status;                                              /* Returned status code. */
int    EPOCHrecVary = {VARY};                               /* EPOCH record variance. */
int    LATrecVary = {NOVARY};                               /* LAT record variance. */
int    LONrecVary = {NOVARY};                               /* LON record variance. */
int    TMPrecVary = {VARY};                                 /* TMP record variance. */
int[]  EPOCHdimVarys = new int[] {NOVARY,NOVARY};           /* EPOCH dimension variances. */
int[]  LATdimVarys = new int[] {VARY,VARY};                 /* LAT dimension variances. */
int[]  LONdimVarys = new int[] {VARY,VARY};                 /* LON dimension variances. */
int[]  TMPdimVarys = new int[] {VARY,VARY};                 /* TMP dimension variances. */
int    EPOCHvarNum;                                         /* EPOCH zVariable number. */
int    LATvarNum;                                           /* LAT zVariable number. */
int    LONvarNum;                                           /* LON zVariable number. */
int    TMPvarNum;                                           /* TMP zVariable number. */
.
.
try {
   ....
   status = CDFvarCreate (id, "EPOCH", CDF_EPOCH, 1,
                          EPOCHrecVary, EPOCHdimVarys, &EPOCHvarNum);
```

```
    status = CDFvarCreate (id, "LATITUDE", CDF_INT2, 1,
                           LATrecVary, LATdimVarys, &LATvarNum);

    status = CDFvarCreate (id, "INTITUDE", CDF_INT2, 1,
                           LONrecVary, LONdimVarys, &LONvarNum);

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

  Or
    status = CDFvarCreate (id, "EPOCH", CDF_EPOCH, 1,
                           EPOCHrecVary, EPOCHdimVarys, out EPOCHvarNum);

    status = CDFvarCreate (id, "LATITUDE", CDF_INT2, 1,
                           LATrecVary, LATdimVarys, out LATvarNum);

    status = CDFvarCreate (id, "INTITUDE", CDF_INT2, 1,
                           LONrecVary, LONdimVarys, out LONvarNum);

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

  .
  } catch (CDFException ex) {
    …
  }.
```

4.3.108 CDFvarGet[11]

```
int CDFvarGet(
void*  id,
int  varNum,
int  recNum,
int[]  indices,
TYPE  value);
```

```
/*  out --  Completion status code.  */
/*  in --  CDF  identifier.  */
/*  in --  rVariable number.  */
/*  in --  Record number.  */
/*  in --  Dimension indices.  */
/*  out --  Value.  */
/*  TYPE --  void*, “out string” or “out object”.  */
```

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 CDFcreateCDF) or CDFopen.
varNum	The rVariable number from which to read data.
recNum	The record number at which to read.

[11] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFgetrVarData is the preferred function for it.

indices	The dimension indices within the record.
value	The data value read. This buffer must be large enough to hold the value.

4.3.108.1. 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. The first get operation passes the value pointer, while the second operation uses "out" argument modifier.

```
.
.
.
void*           id;                                  /*  CDF identifier. */
int    varNum;                                       /*  rVariable number. */
int    recNum;                                       /*  The record number. */
int[]  indices = new int[2];                         /*  The dimension indices. */
double         value1, value2;                       /*  The data values. */
.
.
try {
   ….
   varNum = CDFvarNum (id, "MY_VAR");
   recNum = 0;
   indices[0] = 0;
   indices[1] = 0;
   status = CDFvarGet (id, varNum, recNum, indices, &value1);
   indices[0] = 1;
   indices[1] = 1;
   object value2o;
   status = CDFvarGet (id, varNum, recNum, indices, out value2o);
   value2 = (double) value2o;
} catch (CDFException ex) {
   …
}.
```

4.3.109 CDFvarHyperGet[12]

```
int CDFvarHyperGet(                  /*  out --  Completion status code.  */
void*  id,                           /*  in --  CDF  identifier.  */
int  varNum,                         /*  in --  rVariable number.  */
int  recStart,                       /*  in --  Starting record number.  */
int  recCount,                       /*  in --  Number of records.  */
int  recInterval,                    /*  in --  Subsampling interval between records.  */
int[]  indices,                      /*  in --  Dimension indices of starting  value.  */
int[]  counts,                       /*  in --  Number of values along  each dimension.  */
int[]  intervals,                    /*  in --  Subsampling intervals along  each dimension.  */
```

[12] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFhyperGetrVarData is the preferred function for it.

```
TYPE  buffer);                    /* out -- Values. */
                                  /* TYPE -- void*, "out string", "out string[]" or "out object".
*/
```

CDFvarHyperGet is used to fill a buffer of one or more values from the specified rVariable. It is important to know the variable majority of the CDF before using CDFvarHyperGet because the values placed into the buffer will be in that majority. CDFinquire can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

4.3.109.1. Example(s)

The following example will read an entire record of data from an rVariable. The CDF's rVariables are 3-dimensional with sizes [180,91,10] and CDF's variable majority is ROW_MAJOR. For the rVariable the record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF_REAL4. This example is similar to the example provided for CDFvarGet except that it uses a single call to CDFvarHyperGet rather than numerous calls to CDFvarGet.

```
.
.
.
void*          id;                    /* CDF identifier. */
int    status;                        /* Returned status code. */
float[,,]      tmp;                   /* Temperature values. */
int    varN;                          /* rVariable number. */
int    recStart = 13;                 /* Record number. */
int    recCount = 1;                  /* Record counts. */
int    recInterval = 1;               /* Record interval. */
int[]  indices = new int[] {0,0,0};   /* Dimension indices. */
int[]  counts = new int[] {180,91,10}; /* Dimension counts. */
int[]  intervals = new int[] {1,1,1}; /* Dimension intervals. */
.
.
try {
   varN = CDFgetVarNum (id, "Temperature");
  …
   tmp = new float[180,91,10];
   fixed (void* ptmp = tmp) {
    status = CDFvarHyperGet (id, varN, recStart, recCount, recInterval, indices, counts, intervals, ptmp);
    }
  Or
   status = CDFvarHyperGet (id, varN, recStart, recCount, recInterval, indices, counts, intervals, out tmp);
.
} catch (CDFException ex) {
  …
}.
```

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.

4.3.110 CDFvarHyperPut[13]

```
int CDFvarHyperPut(                 /*  out --  Completion status code.  */
void*  id,                          /*  in --  CDF  identifier.  */
int  varNum,                        /*  in --  rVariable number.  */
int  recStart,                      /*  in --  Starting record number.  */
int  recCount,                      /*  in --  Number of records.  */
int  recInterval,                   /*  in --  Interval between records.  */
int[]  indices,                     /*  in --  Dimension indices of starting  value.  */
int[]  counts,                      /*  in --  Number of values along  each dimension.  */
int[]  intervals,                   /*  in --  Interval between values along each dimension.*/
TYPE  buffer);                      /*  in --  Buffer of values.  */
                                    /*  TYPE --  void*, string, string[] or object.  */
```

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. **Note:** you need to provide dummy arrays, with at least one (1) element, for indices, counts and intervals for scalar variables.

4.3.110.1. 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 CDFvarHyperPut rather than numerous calls to CDFvarPut.

```
.
.
.
void*           id;                         /*  CDF  identifier.  */
Int    status;                              /*  Returned status code.  */
int             lat;                        /*  Latitude  value.  */
short[]         lats = new short[181];      /*  Buffer of latitude  values.  */
int             varN;                       /*  rVariable number.  */
int             recStart  =  0;             /*  Record number.  */
int             recCount  =  1;             /*  Record  counts.  */
int             recInterval  =  1;          /*  Record interval.  */
int[]  indices = new int[] {0,0};           /*  Dimension indices.  */
int[]  counts = new int[] {1,181};          /*  Dimension counts.  */
int[]  intervals = new int[] {1,1};         /*  Dimension intervals.  */

.
.
try {
   ….
   varN  =  CDFvarNum  (id,  "LATITUDE");
   for (lat =  -90;  lat <=  90;  lat ++)
      lats[90+lat]  =  (short) lat;
   fixed (void* plats = lats) {
```

[13] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFhyperPutrVarData is the preferred function for it.

```
        status = CDFvarHyperPut (id, varN, recStart, recCount, recInterval, indices, counts, intervals, plats);
    }
    Or
     status = CDFvarHyperPut (id, varN, recStart, recCount, recInterval, indices, counts, intervals, lats);
…..
} catch (CDFException ex) {
  …
}.
```

4.3.111 CDFvarInquire

```
int CDFvarInquire(
void* id,                                   /* out -- Completion status code. */
int varNum,                                 /* in -- CDF identifier. */
out string varName,                         /* in -- rVariable number. */
TYPE dataType,                              /* out -- rVariable name. */
TYPE numElements,                           /* out -- Data type. */
TYPE recVariance,                           /* out -- Number of elements (of the data type). */
TYPE2 dimVariances);                        /* out -- Record variance. */
                                            /* out -- Dimension variances. */
                                            /* TYPE -- int* or "out int". */
                                            /* TYPE2 -- int* or "out int[]". */
```

CDFvarInquire is used to inquire about the specified rVariable. This method would normally be used before reading rVariable values (with CDFvarGet or CDFvarHyperGet) to determine the data type and number of elements (of that data type).

The arguments to CDFvarInquire are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
varNum	The number of the rVariable to inquire. This number may be determined with a call to CDFvarNum (see Section 4.3.112).
varName	The rVariable's name.
dataType	The data type of the rVariable. The data types are defined in Section 2.6.
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 2.10.
dimVariances	The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are defined in Section 2.10. For 0-dimensional rVariables this argument is ignored (but a placeholder is necessary).

4.3.111.1. 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.

```
.
.
.
void*          id;                  /* CDF identifier. */
int    status;                      /* Returned status code. */
string         varName;             /* rVariable name. */
int    dataType;                    /* Data type of the rVariable. */
int    numElems;                    /* Number of elements (of data type). */
int    recVary;                     /* Record variance. */
int[]  dimVarys;                    /* Dimension variances (allocate to allow the
                                       maximum number of dimensions). */
.
.
try {
  ….
    dimVarys = new int[CDF_MAX_DIMS];
    fixed (int* pdimVarys = dimVarys) {
      status = CDFvarInquire (id, CDFgetVarNum(id,"HEAT_FLUX"), out varName, &dataType,
                              &numElems, &recVary, pdimVarys);
    }
  Or
      status = CDFvarInquire (id, CDFgetVarNum(id,"HEAT_FLUX"), out varName, out dataType,
                              out numElems, out recVary, out dimVarys);
  …
} catch (CDFException ex) {
   …
}.
```

4.3.112 CDFvarNum[14]

```
int CDFvarNum(                      /* out -- Variable number. */
void* id,                           /* in -- CDF identifier. */
string varName);                    /* in -- Variable name. */
```

CDFvarNum is used to determine the number associated with a given variable name. If the variable is found, CDFvarNum returns its variable number - which will be equal to or greater than zero (0). If an error occurs (e.g., the variable does not exist in the CDF), an error code (of type Int) 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:

[14] A legacy CDF function. It used to handle only rVariables. It has been extended to include zVariables. While it is still available in V3.1, CDFgetVarNum is the preferred function for it.

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

4.3.112.1. Example(s)

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

```
.
.
.
void*       id;                 /* CDF identifier. */
int    status;                  /* Returned status code. */
string      varName;            /* Variable name. */
int    dataType;                /* Data type of the rVariable. */
int    numElements;             /* Number of elements (of the data type). */
int    recVariance;             /* Record variance. */
int[]  dimVariances;            /* Dimension variances. */
.
.
try {
  ….
   dimVariances = new int[CDF_MAX_DIMS];
   fixed (int* pdimVariances = dimVariances) {
     status = CDFvarInquire (id, CDFvarNum(id,"LATITUDE"), out varName, &dataType,
                             &numElements, &recVariance, pdimVariances);
   }
  Or
   status = CDFvarInquire (id, CDFvarNum(id,"LATITUDE"), out varName, out dataType,
                           out numElements, out recVariance, out dimVariances);
  .
} catch (CDFException ex) {
   …
}.
```

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 4.3.111.

4.3.113 CDFvarPut[15]

```
int CDFvarPut(                  /* out -- Completion status code. */
void*  id,                      /* in -- CDF identifier. */
int  varNum,                    /* in -- rVariable number. */
```

[15] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFputrVarData is the preferred function for it.

```
int  recNum,                    /*  in --  Record number.  */
int[]  indices,                 /*  in --  Dimension indices.  */
TYPE  value);                   /*  in --  Value.  */
                                /*  TYPE  --  void*, string or object.  */
```

CDFvarPut writes a single data value to an rVariable. CDFvarPut may be used to write more than one value with a single call.

The arguments to CDFvarPut are defined as follows:

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

4.3.113.1. 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.

```
.
.
.
void*              id;                  /*  CDF identifier. */
int     varNum;                         /*  rVariable number. */
int     recNum;                         /*  The record number. */
int[]  indices = new int[2];            /*  The dimension indices. */
double           value1, value2;        /*  The data values. */
.
.
try {
….
   varNum = CDFgetVarNum (id, “MY_VAR”);
   recNum = 0;
   indices[0] = 0;
   indices[1] = 0;
   value1 = 10.1;
   status = CDFvarPut (id, varNum, recNum, indices, &value1);
   indices[0] = 1;
   indices[1] = 1;
   value2 = 20.2;
   status = CDFvarPut (id, varNum, recNum, indices, value2);
.
} catch (CDFException ex) {
```

```
  …
}.
```

4.3.114 CDFvarRename[16]

```
int CDFvarRename(                    /* out -- Completion status code. */
void* id,                            /* in -- CDF identifier. */
int varNum,                          /* in -- rVariable number. */
string 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 CDFcreateCDF) or CDFopen.
varNum	The rVariable number to rename. This number may be determined with a call to CDFvarNum.
varName	The new rVariable name. The maximum length of the new name is CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.

4.3.114.1. 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.

```
.
.
.
void*          id;                   /* CDF identifier. */
int    status;                       /* Returned status code. */
int    varNum;                       /* rVariable number. */
.
.
try {
….
   varNum = CDFvarNum (id, "TEMPERATURE");
…

}
.
} catch (CDFException ex) {
  …
}.
```

[16] A legacy CDF function, handling rVariables only. While it is still available in V3.1, CDFrenamerVar is the preferred function for it.

4.4 Attributes/Entries

This section provides functions that are related to CDF attributes or attribute entries. An attribute is identified by its name or an number in the CDF. Before you can perform any operation on an attribute or attribute entry, the CDF in which it resides must be opened.

4.4.1 CDFattrCreate[17]

```
int CDFattrCreate(                  /* out -- Completion status code. */
void* id,                           /* in -- CDF identifier. */
string attrName,                    /* in -- Attribute name. */
int attrScope,                      /* in -- Scope of attribute. */
TYPE attrNum);                          /* out -- Attribute number. */
                                    /* TYPE -- int* or "out int". */
```

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

The arguments to CDFattrCreate are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrName	The name of the attribute to create. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.
attrScope	The scope of the new attribute. Specify one of the scopes described in Section 2.13.
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.

4.4.1.1. 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).

```
.
.
.
void*       id;                                 /* CDF identifier. */
int    status;                                  /* Returned status code. */
string      UNITSattrName = "Units";            /* Name of "Units" attribute. */
int    UNITSattrNum;                            /* "Units" attribute number. */
int    TITLEattrNum;                            /* "TITLE" attribute number. */
```

[17] Same as CDFcreateAttr.

```
int     TITLEattrScope  =  GLOBAL_SCOPE;                    /*  "TITLE" attribute scope.  */
.
.
try {
  …
  status =  CDFattrCreate  (id,  "TITLE",  TITLEattrScope,  &TITLEattrNum);
  status =  CDFattrCreate  (id,  UNITSattrName,  VARIABLE_SCOPE,  out UNITSattrnum);
 …
 …
} catch (CDFException ex) {
   …
}
.
.
```

4.4.2 CDFattrEntryInquire

```
int CDFattrEntryInquire(                    /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  attrNum,                               /*  in -- Attribute number.  */
int  entryNum,                              /*  in --  Entry number.  */
TYPE  dataType,                                     /*  out --  Data type.  */
TYPE  numElements);                         /*  out --  Number of elements  (of the data type).  */
                                            /*  TYPE --  int* or "out int".  */
```

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

The arguments to CDFattrEntryInquire are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrNum	The attribute number for which to inquire an entry. This number may be determined with a call to CDFattrNum (see Section 4.4.5).
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 2.6.
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.

4.4.2.1. 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.

```
.
.
.
void*          id;                  /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrN;                       /* attribute number. */
int    entryN;                      /* Entry number. */
string         attrName;            /* attribute name. */
int    attrScope;                   /* attribute scope. */
int    maxEntry;                    /* Maximum entry number used. */
int    dataType;                    /* Data type. */
int    numElems;                    /* Number of elements (of the data type). */
.
.
try {
  …
  attrN = CDFgetAttrNum (id, "TMP");
  status = CDFattrInquire (id, attrN, out attrName, &attrScope, &maxEntry);

  for (entryN = 0; entryN <= maxEntry; entryN++) {
    status = CDFattrEntryInquire (id, attrN, entryN, &dataType, &numElems);
  Or
    status = CDFattrEntryInquire (id, attrN, entryN, out dataType, out numElems);

.
        .
  }
…
} catch (CDFException ex) {
  …
}
```

4.4.3 CDFattrGet[18]

```
int CDFattrGet(                    /* out -- Completion status code. */
void* id,                          /* in -- CDF identifier. */
int attrNum,                       /* in -- Attribute number. */
int entryNum,                      /* in -- Entry number. */
TYPE value);                       /* out -- Attribute entry value. */
                                   /* TYPE -- void*, "out string" or "out object". */
```

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.

[18] A legacy CDF function. While it is still available in V3.1, CDFgetAttrgEntry or CDFgetAttrrEntry is the preferred function for it.

The arguments to CDFattrGet are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrNum	The attribute number. This number may be determined with a call to CDFattrNum (Section 4.4.5).
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 method CDFattrEntryInquire would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.

4.4.3.1. 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).

```
.
.
.
void*       id;                                  /* CDF identifier. */
Int    status;                                   /* Returned status code. */
int         attrN;                               /* Attribute number. */
int         entryN;                              /* Entry number. */
int         dataType;                            /* Data type. */
int         numElems;                            /* Number of elements (of data type). */
.
.
try {
   …
   attrN = CDFattrNum (id, "UNITS");
   entryN = CDFvarNum (id, "PRES_LVL");    /* The rEntry number is the rVariable number. */

   status = CDFattrEntryInquire (id, attrN, entryN, out dataType, out numElems);

   if (dataType == CDF_CHAR) {
     string buffer;
     status = CDFattrGet (id, attrN, entryN, out buffer);
   }
} catch (CDFException ex) {
   …
}.
.
```

4.4.4 CDFattrInquire[19]

```
int CDFattrInquire(                 /*  out --  Completion status code.  */
void*  id,                          /*  in --  CDF  identifier.  */
int  attrNum,                       /*  in --  Attribute number.  */
out string  attrName,               /*  out --  Attribute name.  */
TYPE  attrScope,                    /*  out --  Attribute scope.  */
TYPE  maxEntry);                    /*  out --  Maximum  gEntry/rEntry number.  */
                                    /*  TYPE --  int* or "out int".  */
```

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

The arguments to CDFattrInquire are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrNum	The number of the attribute to inquire. This number may be determined with a call to CDFattrNum (see Section 4.4.5).
attrName	The attribute's name. This string length is limited to CDF_ATTR_NAME_LEN256.
attrScope	The scope of the attribute. Attribute scopes are defined in Section 2.13.
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). If no entries exist for the attribute, then a value of -1 will be passed back.

4.4.4.1. 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 method CDFinquire. Note that attribute numbers start at zero (0) and are consecutive.

```
.
.
.
void*          id;                  /*  CDF  identifier.  */
int    status;                      /*  Returned status code.  */
int    numDims;                     /*  Number of dimensions.  */
int[]  dimSizes;                    /*  Dimension sizes (allocate to allow the maximum
                                        number of dimensions).  */
int    encoding;                    /*  Data encoding.  */
int    majority;                    /*  Variable majority.  */
int    maxRec;                      /*  Maximum record number in CDF.  */
int    numVars;                     /*  Number of variables in CDF.  */
int    numAttrs;                    /*  Number of attributes in CDF.  */
int    attrN;                       /* attribute number.  */
string         attrName;            /* attribute name.  */
```

[19] A legacy function. While it is still available in V3.1, CDFinquireAttr is the preferred function for it.

```
int    attrScope;                                   /* attribute scope.  */
int    maxEntry;                                    /*  Maximum entry number.  */
.
.
try {
   ….
   status =  CDFinquire  (id,  out numDims,  out dimSizes,  out encoding,  out majority, out maxRec,  out numVars,
                          out numAttrs);
   for (attrN  =  0; attrN  <  numAttrs; attrN++)  {
     status =  CDFattrInquire  (id, attrN, out attrName,  out attrScope,  out maxEntry);
    Or
    status =  CDFattrInquire  (id, attrN, out attrName,  &attrScope,  &maxEntry);
   }
} catch (CDFException ex) {
  …
}
.
```

4.4.5 CDFattrNum[20]

```
int  CDFattrNum(                                    /*  out -- attribute number.  */
void*  id,                                          /*  in --  CDF  id  */
string  attrName);                                  /*  in --  Attribute name */
```

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

The arguments to CDFattrNum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrName	The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.

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

4.4.5.1. 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.

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
```

[20] A legacy CDF function. While it is still available in V3.1, CDFgetAttrNum is the preferred function for it.

```
int     status;                                          /*  Returned status code.  */
.
.
try {
    ….
    status =  CDFattrRename (id,  CDFattrNum(id,"pressure"),  "PRESSURE");
    ….
} catch (CDFException ex) {
   …
}
```

4.4.6 CDFattrPut

```
int CDFattrPut(                          /*  out --  Completion status code.  */
void*  id,                               /*  in --  CDF  identifier.  */
int  attrNum,                            /*  in -- Attribute number.  */
int  entryNum,                           /*  in --  Entry number.  */
int  dataType,                           /*  in --  Data type of this entry.  */
int  numElements,                        /*  in --  Number of elements  (of the data type).  */
TYPE  value);                            /*  in --  Attribute entry value.  */
                                         /*  TYPE --  void*, string, string[] or object.  */
```

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

The arguments to CDFattrPut are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopen.
attrNum	The attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The entry number. If the attribute is global in scope, this is simply the gEntry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated rVariable (the rVariable being described in some way by the rEntry).
dataType	The data type of the specified entry. Specify one of the data types defined in Section 2.6.
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.

4.4.6.1. 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.

```
.
.
.
void*          id;                                  /*  CDF  identifier.  */
int    status;                                      /*  Returned status code.  */
int    TITLE_LEN = 10;                              /*  Entry string length.  */
int    entryNum;                                    /*  Entry number.  */
int    numElements;                                 /*  Number of elements  (of data type).  */
string         title  =  "CDF  title.";             /*  Value of TITLE attribute, entry number 0. */
short[]        TMPvalids  =  new short[] {15,30};   /*  Value(s) of VALIDs attribute,
                                                          rEntry for rVariable TMP.  */
.
.
entryNum  =  0;
try {
   status =  CDFattrPut  (id,  CDFgetAttrNum(id,"TITLE"), entryNum,  CDF_CHAR, TITLE_LEN,  title);
.
   numElements  =  2;
   status =  CDFattrPut  (id,  CDFgetAttrNum(id,"VALIDs"),  CDFgetVarNum(id,"TMP"),
                          CDF_INT2,  numElements,  TMPvalids);
 Or
   fixed (void* pTMPvalids = TMPvalids) {
     status =  CDFattrPut  (id,  CDFgetAttrNum(id,"VALIDs"),  CDFgetVarNum(id,"TMP"),
                            CDF_INT2,  numElements,  pTMPvalids);
   }

} catch (CDFException ex) {
   …
}.
.
```

4.4.7 CDFattrRename[21]

```
int CDFattrRename(                                 /*  out --  Completion status code.  */
void*  id,                                         /*  in --  CDF  identifier.  */
int  attrNum,                                      /*  in -- Attribute number.  */
string  attrName);                                 /*  in --  New attribute name.  */
```

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

The arguments to CDFattrRename are defined as follows:

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

[21] A legacy CDF function. While it is still available in V3.1, CDFrenameAttr is the preferred function for it.

attrNum — The number of the attribute to rename. This number may be determined with a call to CDFattrNum (see Section 4.4.5).

attrName — The new attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.

4.4.7.1. Example(s)

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

```
.
.
.
void*          id;                                          /* CDF identifier. */
int    status;                                              /* Returned status code. */
.
.
try {
   status = CDFattrRename (id, CDFgetAttrNum(id,"LAT"), "LATITUDE");
.
} catch (CDFException ex) {
  …
}.
```

4.4.8 CDFconfirmAttrExistence

```
int CDFconfirmAttrExistence(                               /* out -- Completion status code. */
void* id,                                                  /* in -- CDF identifier. */
string attrName)                                           /* in -- Attribute name. */
```

CDFconfirmAttrExistence confirms whether an attribute exists for the given attribute name in a CDF. If the attribute doesn't exist, the informational status code, NO_SUCH_ATTR, is returned and no exception is thrown.

The arguments to CDFconfirmAttrExistence are defined as follows:

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

attrName — The attribute name to check.

4.4.8.1. Example(s)

The following example checks whether an attribute by the name of "ATTR_NAME1" is in a CDF.

```
.
.
.
```

```
void*           id;                                          /* CDF identifier. */
int     status;                                              /* Returned status code. */
.
.
try {
   ….
   status =  CDFconfirmAttrExistence  (id, “ATTR_NAME1”);
    if (status == NO_SUCH_ATTR) {
     ….
    }
.
} catch (CDFException ex) {
  …
}.
```

4.4.9 CDFconfirmgEntryExistence

```
int CDFconfirmgEntryExistence(                       /* out -- Completion status code. */
void*  id,                                           /* in -- CDF identifier. */
int  attrNum,                                        /* in -- Attribute number. */
int  entryNum);                                      /* in -- gEntry number. */
```

CDFconfirmgEntryExistence confirms the existence of the specified entry (gEentry), in a global attribute from a CDF. If the gEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned and no exception is thrown.

The arguments to CDFconfirmgEntryExistence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The (global) attribute number.
entryNum	The gEntry number.

4.4.9.1. Example(s)

The following example checks the existence of a gEntry numbered 1 for attribute “MY_ATTR” in a CDF.

```
.
.
.
void*           id;                                  /* CDF identifier. */
int    status;                                       /* Returned status code. */
int    attrNum;                                      /* Attribute number. */
int     entryNum;                                    /* gEntry number. */
.
.
try {
```

```
    ….
    attrNum = CDFgetAttrNum(id, “MY_ATTR”);
    entryNum = 1;
    status =  CDFconfirmgEntryExistence  (id, attrNum, entryNum);
    if (status == NO_SUCH_ENTRY)  UserStatusHandler  (status);
.
.
```

4.4.10 CDFconfirmrEntryExistence

```
int CDFconfirmrEntryExistence(                  /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  attrNum,                                   /*  in --  Attribute number.  */
int  entryNum);                                 /*  in --  rEntry number.  */
```

CDFconfirmrEntryExistence confirms the existence of the specified entry (rEntry), corresponding to an rVariable, in a variable attribute from a CDF. If the rEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned and no exception is thrown.

The arguments to CDFconfirmrEntryExistence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The variable attribute number.
entryNum	The rEntry number.

4.4.10.1. Example(s)

The following example checks the existence of an rEntry, corresponding to rVariable “MY_VAR”, for attribute “MY_ATTR” in a CDF.

```
.
.
.
void*  id;                                      /*  CDF  identifier.  */
int status;                                     /*  Returned status code.  */
int attrNum;                                    /*  Attribute number.  */
int entryNum;                                   /*  rEntry number.  */
.
.
try {
    ….
    attrNum = CDFgetAttrNum(id, “MY_ATTR”);
    entryNum = CDFgetVarNum(id, “MY_VAR”);
    status =  CDFconfirmrEntryExistence  (id, attrNum, entryNum);
     if (status == NO_SUCH_ENTRY)  UserStatusHandler  (status);
.
} catch (CDFException ex) {
```

```
  …
}.
```

4.4.11 CDFconfirmzEntryExistence

```
int CDFconfirmzEntryExistence(          /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int attrNum,                            /* in -- Attribute number. */
int entryNum);                          /* in -- zEntry number. */
```

CDFconfirmzEntryExistence confirms the existence of the specified entry (zEntry), corresponding to a zVariable, in a variable attribute from a CDF. If the zEntry does not exist, the informational status code NO_SUCH_ENTRY will be returned and no exception is thrown.

The arguments to CDFconfirmzEntryExistence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The (variable) attribute number.
entryNum	The zEntry number.

4.4.11.1. Example(s)

The following example checks the existence of the zEntry corresponding to zVariable “MY_VAR” for the variable attribute “MY_ATTR” in a CDF.

```
.
.
.
void*       id;                     /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrNum;                     /* Attribute number. */
int    entryNum;                    /* zEntry number. */
.
.
try {
   ….
   attrNum = CDFgetAttrNum(id, “MY_ATTR”);
   entryNum = CDFgetVarNum(id, “MY_VAR”);
   status =  CDFconfirmzEntryExistence  (id, attrNum, entryNum);
   if (status == NO_SUCH_ENTRY)  UserStatusHandler  (status);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.12 CDFcreateAttr

```
int CDFcreateAttr(                  /* out -- Completion status code. */
void* id,                           /* in -- CDF identifier. */
string attrName,                    /* in -- Attribute name. */
int attrScope,                      /* in -- Scope of attribute. */
TYPE attrNum);                              /* out -- Attribute number. */
                                    /* TYPE -- int* or "out int". */
```

CDFcreateAttr creates an attribute with the specified scope in a CDF. It is identical to the method CDFattrCreate. An attribute with the same name must not already exist in the CDF.

The arguments to CDFcreateAttr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrName	The name of the attribute to create. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.
attrScope	The scope of the new attribute. Specify one of the scopes described in Section 2.13.
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.

4.4.12.1. 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).

```
.
.
.
void*          id;                                  /* CDF identifier. */
int    status;                                      /* Returned status code. */
string         UNITSattrName = "Units";             /* Name of "Units" attribute. */
int            UNITSattrNum;                        /* "Units" attribute number. */
int            TITLEattrNum;                        /* "TITLE" attribute number. */
int    TITLEattrScope = GLOBAL_SCOPE;               /* "TITLE" attribute scope. */
.
.
try {
   ….
   status = CDFcreateAttr (id, "TITLE", TITLEattrScope, &TITLEattrNum);
   status = CDFcreateAttr (id, UNITSattrName, VARIABLE_SCOPE, out UNITSattrnum);
.
} catch (CDFException ex) {
  …
}.
```

.

4.4.13 CDFdeleteAttr

```
int CDFdeleteAttr(                      /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int attrNum);                           /* in -- Attribute identifier. */
```

CDFdeleteAttr deletes the specified attribute from a CDF.

The arguments to CDFdeleteAttr are defined as follows:

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

4.4.13.1. Example(s)

The following example deletes an existing attribute named MY_ATTR from a CDF.

```
.
.
.
void*       id;                         /* CDF identifier. */
int    status;                          /* Returned status code. */
int    attrNum;                         /* Attribute number. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   status = CDFdeleteAttr (id, attrNum);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.14 CDFdeleteAttrgEntry

```
int CDFdeleteAttrgEntry(                /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int attrNum,                            /* in -- Attribute identifier. */
int entryNum);                          /* in -- gEntry identifier. */
```

CDFdeleteAttrgEntry deletes the specified entry (gEntry) in a global attribute from a CDF.

The arguments to CDFdeleteAttrgEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The global attribute number from which to delete an attribute entry.
entryNum	The gEntry number to delete.

4.4.14.1. Example(s)

The following example deletes the entry number 5 from an existing global attribute MY_ATTR in a CDF.

```
.
.
.
void*       id;                  /* CDF identifier. */
int    status;                   /* Returned status code. */
int    attrNum;                  /* Attribute number. */
int    entryNum;                 /* gEntry number. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   entryNum = 5;
   status = CDFdeleteAttrgEntry (id, attrNum, entryNum);

.
} catch (CDFException ex) {
  …
}.
.
```

4.4.15 CDFdeleteAttrrEntry

```
int CDFdeleteAttrrEntry(          /* out -- Completion status code. */
void* id,                         /* in -- CDF identifier. */
int attrNum,                      /* in -- Attribute identifier. */
int entryNum);                    /* in -- rEntry identifier. */
```

CDFdeleteAttrrEntry deletes the specified entry (rEntry), corresponding to an rVariable, in an (variable) attribute from a CDF.

The arguments to CDFdeleteAttrrEntry are defined as follows:

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

attrNum The (variable) attribute number.

entryNum The rEntry number.

4.4.15.1. Example(s)

The following example deletes the entry corresponding to rVariable “MY_VAR1” from the variable attribute “MY_ATTR” in a CDF.

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    status;                                       /* Returned status code. */
int    attrNum;                                      /* Attribute number. */
int    entryNum;                                     /* rEntry number. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   entryNum = CDFgetVarNum(id, “MY_VAR1”);
   status = CDFdeleteAttrrEntry (id, attrNum, entryNum);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.16 CDFdeleteAttrzEntry

```
int CDFdeleteAttrzEntry(                             /* out -- Completion status code. */
void*  id,                                           /* in -- CDF identifier. */
int attrNum,                                         /* in -- Attribute identifier. */
int entryNum);                                       /* in -- zEntry identifier. */
```

CDFdeleteAttrzEntry deletes the specified entry (zEntry), corresponding to a zVariable, in an (variable) attribute from a CDF.

The arguments to CDFdeleteAttrzEntry are defined as follows:

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

attrNum The identifier of the variable attribute.

entryNum The zEntry number to be deleted that is the zVariable number.

4.4.16.1. Example(s)

The following example deletes the variable attribute entry named MY_ATTR that is attached to the zVariable MY_VAR1.

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrNum;                     /* Attribute number. */
int    entryNum;                    /* zEntry number. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   entryNum = CDFgetVarNum(id, “MY_VAR1”);
   status = CDFdeleteAttrzEntry (id, attrNum, entryNum);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.17 CDFgetAttrgEntry

```
int CDFgetAttrgEntry (              /* out -- Completion status code. */
void* id,                           /* in -- CDF identifier. */
int attrNum,                        /* in -- Attribute identifier. */
int entryNum,                       /* in -- gEntry number. */
TYPE value);                        /* out -- gEntry data. */
                                    /* TYPE -- void*, “out string” or “out object”. */
```

This method is identical to the method CDFattrGet. CDFgetAttrgEntry is used to read a global attribute entry from a CDF. In most cases it will be necessary to call CDFinquireAttrgEntry before calling CDFgetAttrgEntry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDFgetAttrgEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The global attribute entry number.
value	The value read. This buffer must be large enough to hold the value. The method CDFattrEntryInquire would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.

4.4.17.1. Example(s)

The following example displays the value of the global attribute called HISTORY.

```
.
.
.
void*          id;                  /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrN;                       /* Attribute number. */
int    entryN;                      /* Entry number. */
int    dataType;                    /* Data type. */
int    numElems;                    /* Number of elements (of data type). */
string         buffer;              /* Buffer to receive value. */
.
.
try {
   ….
   attrN = CDFattrNum (id, "HISTORY");
   entryN = 0;
   status = CDFinquireAttrgEntry (id, attrN, entryN, out dataType, out numElems);
   if (dataType == CDF_CHAR) {
     status = CDFgetAttrgEntry (id, attrN, entryN, out buffer);

  }
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.18 CDFgetAttrgEntryDataType

```
int CDFgetAttrgEntryDataType (          /* out -- Completion status code. */
void* id,                               /* in -- CDF identifier. */
int attrNum,                            /* in -- Attribute identifier. */
int entryNum,                           /* in -- gEntry number. */
TYPE dataType);                         /* out -- gEntry data type. */
                                        /* TYPE -- int* or "out int". */
```

CDFgetAttrgEntryDataType returns the data type of the specified global attribute and gEntry number in a CDF. The data types are described in Section 2.6.

The arguments to CDFgetAttrgEntryDataType are defined as follows:

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

entryNum The gEntry number.

dataType The data type of the gEntry.

4.4.18.1. Example(s)

The following example gets the data type for the gEntry numbered 2 from the global attribute “MY_ATTR” in a CDF.

```
.
.
.
void*          id;                                   /* CDF identifier. */
int    status;                                       /* Returned status code. */
int    attrNum;                                      /* Attribute number. */
int    entryNum;                                     /* gEntry number. */
int    dataType;                                     /* gEntry data type. */
.
.
try {
    ….
    attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
    entryNum = 2;
    status = CDFgetAttrgEntryDataType (id, attrNum, entryNum, &dataType);
   Or
    status = CDFgetAttrgEntryDataType (id, attrNum, entryNum, out dataType);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.19 CDFgetAttrgEntryNumElements

```
int CDFgetAttrgEntryNumElements (                  /* out -- Completion status code. */
void* id,                                          /* in -- CDF identifier. */
int attrNum,                                       /* in -- Attribute identifier. */
int entryNum,                                      /* in -- gEntry number. */
TYPE numElems);                                    /* out -- gEntry’s number of elements. */
                                                   /* TYPE -- int* or “out int”. */
```

CDFgetAttrgEntryNumElements returns the number of elements of the specified global attribute and gentry number in a CDF.

The arguments to CDFgetAttrgEntryNumElements are defined as follows:

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

attrNum The identifier of the global attribute.

entryNum The gEntry number.

numElems The number of elements of the gEntry.

4.4.19.1. Example(s)

The following example gets the number of elements from the gEntry numbered 2 from the global attribute "MY_ATTR" in a CDF.

```
.
.
.
void*          id;                    /* CDF identifier. */
int    status;                        /* Returned status code. */
int    attrNum;                       /* Attribute number. */
int    entryNum;                      /* gEntry number. */
int    numElements;                   /* gEntry's number of elements. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, "MY_ATTR");
   entryNum = 2;
   status = CDFgetAttrgEntryNumElements (id, attrNum, entryNum, out numElements);
  Or
   status = CDFgetAttrgEntryNumElements (id, attrNum, entryNum, &numElements);

.
} catch (CDFException ex) {
  …
}.
.
```

4.4.20 CDFgetAttrMaxgEntry

```
int CDFgetAttrMaxgEntry (                /* out -- Completion status code. */
void*  id,                               /* in -- CDF identifier. */
int  attrNum,                            /* in -- Attribute identifier. */
TYPE  maxEntry);                         /* out -- The last gEntry number. */
                                         /* TYPE -- int* or "out int". */
```

CDFgetAttrMaxgEntry returns the last entry number of the specified global attribute in a CDF.

The arguments to CDFgetAttrMaxgEntry are defined as follows:

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

attrNum The identifier of the global attribute.

maxEntry The last gEntry number.

4.4.20.1. Example(s)

The following example gets the last entry number from the global attribute “MY_ATTR” in a CDF.

```
.
.
.
void*          id;                  /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrNum;                     /* Attribute number. */
int    maxEntry;                    /* The last gEntry number. */
.
.
try {
    ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   status = CDFgetAttrMaxgEntry (id, attrNum, out maxEntry);
  Or
   status = CDFgetAttrMaxgEntry (id, attrNum, &maxEntry);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.21 CDFgetAttrMaxrEntry

```
int CDFgetAttrMaxrEntry (                /* out -- Completion status code. */
void*  id,                               /* in -- CDF identifier. */
int  attrNum,                            /* in -- Attribute identifier. */
TYPE  maxEntry);                         /* out -- The maximum rEntry number. */
                                         /* TYPE -- int* or “out int”. */
```

CDFgetAttrMaxrEntry returns the last rEntry number (rVariable number) to which the given variable attribute is attached.

The arguments to CDFgetAttrMaxrEntry are defined as follows:

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

attrNum The identifier of the variable attribute.

maxEntry The last rEntry number (rVariable number) to which attrNum is attached..

4.4.21.1. Example(s)

The following example gets the last entry, corresponding to the last rVariable number, from the variable attribute "MY_ATTR" in a CDF.

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrNum;                     /* Attribute number. */
int    maxEntry;                    /* The last rEntry number. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, "MY_ATTR");
   status = CDFgetAttrMaxrEntry (id, attrNum, out maxEntry);
  Or
   status = CDFgetAttrMaxrEntry (id, attrNum, &maxEntry);

} catch (CDFException ex) {
  …
}.
.
```

4.4.22 CDFgetAttrMaxzEntry

```
int CDFgetAttrMaxzEntry (                /* out -- Completion status code. */
void*  id,                               /* in -- CDF identifier. */
int  attrNum,                            /* in -- Attribute identifier. */
TYPE  maxEntry);                         /* out -- The maximum zEntry number. */
                                         /* TYPE -- int* or "out int". */
```

CDFgetAttrMaxzEntry returns the last entry number, corresponding to the last zVariable number, to which the given variable attribute is attached.

The arguments to CDFgetAttrMaxzEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The identifier of the variable attribute.
maxEntry	The last zEntry number (zVariable number) to which attrNum is attached..

4.4.22.1. Example(s)

The following example gets the last entry, corresponding to the last zVariable number, attached to the variable attribute MY_ATTR in a CDF.

```
.
.
.
void*           id;                                  /*  CDF  identifier.  */
int    status;                                       /*  Returned status code.  */
int    attrNum;                                      /*  Attribute number.  */
int    maxEntry;                                     /*  The last zEntry number  */
.
.
try {
    ….
    attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
    status = CDFgetAttrMaxzEntry (id, attrNum, out maxEntry);
  Or
    status = CDFgetAttrMaxzEntry (id, attrNum, &maxEntry);.
} catch (CDFException ex) {
   …
}.
.
```

4.4.23 CDFgetAttrName

```
int CDFgetAttrName (                                 /*  out --  Completion status code.  */
void*  id,                                           /*  in --  CDF  identifier.  */
int  attrNum,                                        /*  in --  Attribute  identifier.  */
out string  attrName);                               /*  out --  The attribute name.  */
```

CDFgetAttrName gets the name of the specified attribute (by its number) in a CDF.

The arguments to CDFgetAttrName are defined as follows:

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

4.4.23.1. Example(s)

The following example retrieves the name of the attribute number 2, if it exists, in a CDF.

```
.
.
.
void*           id;                                  /*  CDF  identifier.  */
int    status;                                       /*  Returned status code.  */
```

```
int     attrNum;                                        /*  Attribute number.  */
string          attrName;                               /*  The attribute name.  */
.
.
attrNum = 2;
try {
   ….
   status = CDFgetAttrName (id, attrNum, out attrName);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.24 CDFgetAttrNum

```
int CDFgetAttrNum (                                     /*  out --  Attribute number.  */
void*  id,                                              /*  in --  CDF  identifier.  */
string  attrName);                                      /*  in --  The attribute name.  */
```

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

The arguments to CDFgetAttrNum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrName	The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.

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

4.4.24.1. Example(s)

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

```
.
.
.
void*           id;                                     /*  CDF  identifier.  */
int     status;                                         /*  Returned status code.  */
.
.
try {
   ….
```

```
    status =  CDFrenameAttr (id,  CDFgetAttrNum(id,"pressure"),  "PRESSURE");

} catch (CDFException ex) {
   …
}.
```

4.4.25 CDFgetAttrrEntry

```
int CDFgetAttrrEntry (                  /*  out --  Completion status code.  */
void*  id,                              /*  in --  CDF  identifier.  */
int  attrNum,                           /*  in --  Attribute  identifier.  */
int  entryNum,                          /*  in --  Entry number.  */
TYPE  value);                           /*  out --  Entry data.  */
                                        /*  TYPE --  void*, "out string" or "out object".  */
```

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

The arguments to CDFgetAttrrEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The rVariable attribute entry number that is the rVariable number from which the attribute is read.
value	The entry value read. This buffer must be large enough to hold the value. The method CDFattrEntryInquire would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.

4.4.25.1. 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).

```
.
.

.
void*          id;                      /*  CDF  identifier.  */
int    status;                          /*  Returned status code.  */
int    attrN;                           /*  Attribute number.  */
int    entryN;                          /*  Entry number.  */
int    dataType;                        /*  Data type.  */
int    numElems;                        /*  Number of elements  (of data type).  */
.
```

```
.
try {
   ….
   attrN = CDFattrNum (id, "UNITS");
   entryN = CDFvarNum (id, "PRES_LVL");    /* The rEntry number is the rVariable number. */
   status = CDFinquireAttrrEntry (id, attrN, entryN, out dataType, out numElems);
   if (dataType == CDF_CHAR) {
     string buffer;
     status = CDFgetAttrrEntry (id, attrN, entryN, out buffer);

   } .
} catch (CDFException ex) {
  …
}.
.
```

4.4.26 CDFgetAttrrEntryDataType

```
int CDFgetAttrrEntryDataType (                  /* out -- Completion status code. */
void* id,                                       /* in -- CDF identifier. */
int attrNum,                                    /* in -- Attribute identifier. */
int entryNum,                                   /* in -- rEntry number. */
TYPE dataType);                                 /* out -- rEntry data type. */
                                                /* TYPE -- int* or "out int". */
```

CDFgetAttrrEntryDataType returns the data type of the rEntry from an (variable) attribute in a CDF. The data types are described in Section 2.6.

The arguments to CDFgetAttrrEntryDataType are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The identifier of the variable attribute.
entryNum	The rEntry number.
dataType	The data type of the rEntry.

4.4.26.1. Example(s)

The following example gets the data type for the entry of rVariable "MY_VAR1" in the (variable) attribute "MY_ATTR" in a CDF.

```
.
.
.
void*       id;                                 /* CDF identifier. */
int    status;                                  /* Returned status code. */
int    attrNum;                                 /* Attribute number. */
```

```
int    entryNum;                                        /* rEntry number. */
int    dataType;                                        /* rEntry data type. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   entryNum = CDFgetVarNum(id, “MY_VAR1”);
   status = CDFgetAttrrEntryDataType (id, attrNum, entryNum, out dataType);
  Or
   status = CDFgetAttrrEntryDataType (id, attrNum, entryNum, &dataType);

.
} catch (CDFException ex) {
  …
}.
.
```

4.4.27 CDFgetAttrrEntryNumElements

```
int CDFgetAttrrEntryNumElements (                   /* out -- Completion status code. */
void*  id,                                          /* in -- CDF identifier. */
int  attrNum,                                       /* in -- Attribute identifier. */
int  startRec,                                      /* in -- rEntry number. */
TYPE  numElems);                                    /* out -- rEntry’s number of elements. */
                                                    /* TYPE -- int* or “out int”. */
```

CDFgetAttrrEntryNumElements returns the number of elements of the rEntry from an (variable) attribute in a CDF.

The arguments to CDFgetAttrrEntryNumElements are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The identifier of the variable attribute.
entryNum	The rEntry number.
numElems	The number of elements of the rEntry.

4.4.27.1. Example(s)

The following example gets the number of elements for the entry of rVariable “MY_VAR1” in the (variable) attribute “MY_ATTR” in a CDF.

```
.
.
.
void*          id;                                  /* CDF identifier. */
int    status;                                      /* Returned status code. */
```

```
int    attrNum;                                   /*  Attribute number.  */
int    entryNum;                                  /*  rEntry number.  */
int    numElements;                               /*  rEntry's number of elements.  */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, "MY_ATTR");
   entryNum = CDFgetVarNum(id, "MY_VAR1");
   status = CDFgetAttrrEntryNumElements (id, attrNum, entryNum, out numElements);
 Or
   status = CDFgetAttrrEntryNumElements (id, attrNum, entryNum, &numElements);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.28 CDFgetAttrScope

```
int CDFgetAttrScope (                             /*  out --  Completion status code.  */
void*  id,                                        /*  in --  CDF  identifier.  */
int  attrNum,                                     /*  in --  Attribute  number.  */
TYPE  attrScope);                                 /*  out --  Attribute scope.  */
                                                  /*  TYPE --  int* or "out int".  */
```

CDFgetAttrScope returns the attribute scope (GLOBAL_SCOPE or VARIABLE_SCOPE) of the specified attribute in a CDF. Refer to Section 2.13 for the description of the attribute scopes.

The arguments to CDFgetAttrScope are defined as follows:

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

4.4.28.1. Example(s)

The following example gets the scope of the attribute "MY_ATTR" in a CDF.

```
.
.
.
void*          id;                                /*  CDF  identifier.  */
int    status;                                    /*  Returned status code.  */
int    attrNum;                                   /*  Attribute number.  */
int    attrScope;                                 /*  Attribute scope.  */
.
```

```
.
try {
    ….
    attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
    status = CDFgetAttrScope (id, attrNum, out attrScope);
  Or
    status = CDFgetAttrScope (id, attrNum, &attrScope);

.
} catch (CDFException ex) {
   …
}.
.
```

4.4.29 CDFgetAttrzEntry

```
int CDFgetAttrzEntry(                  /*  out --  Completion status code.  */
void*  id,                             /*  in --  CDF  identifier.  */
int  attrNum,                          /*  in --  Variable attribute number.  */
int  entryNum,                         /*  in --  Entry number.  */
TYPE  value);                          /*  out --  Entry value.  */
                                       /*  TYPE --  void*, “out string” or “out object”.  */
```

CDFgetAttrzEntry is used to read zVariable’s attribute entry.. In most cases it will be necessary to call CDFinquireAttrzEntry before calling this method in order to determine the data type and number of elements (of that data type) for dynamical space allocation for the entry.

The arguments to CDFgetAttrzEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The variable attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The variable attribute entry number that is the zVariable number from which the attribute entry is read
value	The entry value read. This buffer must be large enough to hold the value. The method CDFattrEntryInquire would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.

4.4.29.1. Example(s)

The following example displays the value of the UNITS attribute for the PRES_LVL zVariable (but only if the data type is CDF_CHAR).

```
.
.
```

```
.
void*           id;                                   /* CDF identifier. */
int    status;                                        /* Returned status code. */
int    attrN;                                         /* Attribute number. */
int    entryN;                                        /* Entry number. */
int    dataType;                                      /* Data type. */
int    numElems;                                      /* Number of elements (of data type). */
.
try {
   ….
   attrN = CDFgetAttrNum (id, "UNITS");
   entryN = CDFgetVarNum (id, "PRES_LVL");          /* The zEntry number is the zVariable number. */
   status = CDFinquireAttrzEntry (id, attrN, entryN, out dataType, out numElems);
   if (dataType == CDF_CHAR) {
     string buffer;
    status = CDFgetAttrzEntry (id, attrN, entryN, out buffer);
  }
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.30 CDFgetAttrzEntryDataType

```
int CDFgetAttrzEntryDataType (                      /* out -- Completion status code. */
void* id,                                           /* in -- CDF identifier. */
int attrNum,                                        /* in -- Attribute identifier. */
int entryNum,                                       /* in -- zEntry number. */
TYPE dataType);                                     /* out -- zEntry data type. */
                                                    /* TYPE -- int* or "out int". */
```

CDFgetAttrzEntryDataType returns the data type of the zEntry for the specified variable attribute in a CDF. The data types are described in Section 2.6.

The arguments to CDFgetAttrzEntryDataType are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The identifier of the variable attribute.
entryNum	The zEntry number that is the zVariable number.
dataType	The data type of the zEntry.

4.4.30.1. Example(s)

The following example gets the data type of the attribute named MY_ATTR for the zVariable MY_VAR1 in a CDF.

```
.
.
.
void*           id;                 /*  CDF  identifier.  */
int    status;                      /*  Returned status code.  */
int    attrNum;                     /*  Attribute number.  */
int    entryNum;                    /*  zEntry number.  */
int    dataType;                    /*  zEntry data type.  */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, “MY_ATTR”);
   entryNum = CDFgetVarNum(id, “MY_VAR1”);
   status = CDFgetAttrzEntryDataType (id, attrNum, entryNum, out dataType);
  Or
   status = CDFgetAttrzEntryDataType (id, attrNum, entryNum, &dataType);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.31 CDFgetAttrzEntryNumElements

```
int CDFgetAttrzEntryNumElements (
void*  id,
int  attrNum,
int  entryNum,
TYPE  numElems);
```

```
/*  out --  Completion status code.  */
/*  in --  CDF  identifier.  */
/*  in --  Attribute  identifier.  */
/*  in --  zEntry number.  */
/*  out --  zEntry’s number of elements.  */
/*  TYPE --  int* or “out int”.  */
```

CDFgetAttrzEntryNumElements returns the number of elements of the zEntry for the specified variable attribute in a CDF.

The arguments to CDFgetAttrzEntryNumElements are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The identifier of the variable attribute.
entryNum	The zEntry number that is the zVariable number.
numElems	The number of elements of the zEntry.

4.4.31.1. Example(s)

The following example returns the number of elements for attribute named MY_ATTR for the zVariable MY_VAR1 in a CDF

```
.
.
.
void*           id;                 /* CDF identifier. */
int    status;                      /* Returned status code. */
int    attrNum;                     /* Attribute number. */
int    entryNum;                    /* zEntry number. */
int    numElements;                 /* zEntry's number of elements. */
.
.
try {
   ….
   attrNum =  CDFgetAttrNum (id, "MY_ATTR");
   entryNum = CDFgetVarNum(id, "MY_VAR1");
   status = CDFgetAttrzEntryNumElements (id, attrNum, entryNum, out numElements);
  Or
   status = CDFgetAttrzEntryNumElements (id, attrNum, entryNum, &numElements);

} catch (CDFException ex) {
   …
}.
.
```

4.4.32 CDFgetNumAttrgEntries

```
int CDFgetNumAttrgEntries (          /* out -- Completion status code. */
void* id,                            /* in -- CDF identifier. */
int  attrNum,                        /* in -- Attribute number. */
TYPE  entries);                      /* out -- Total gEntries. */
                                     /* TYPE -- int* or "out int". */
```

CDFgetNumAttrgEntries returns the total number of entries (gEntries) written for the specified global attribute in a CDF.

The arguments to CDFgetNumAttrgEntries are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number.
entries	Number of gEntries for attrNum.

4.4.32.1. Example(s)

The following example retrieves the total number of gEntries for the global attribute MY_ATTR in a CDF.

```
.
.
.
int    status;                                  /*  Returned status code.  */
void*          id;                              /*  CDF identifier.  */
int    attrNum;                                 /*  Attribute number.  */
int    numEntries;                              /*  Number of entries.  */
int    i;
.
.
try {
   ….
   attrNum = CDFgetAttrNum(id, “MUY_ATTR”);
   status = CDFgetNumAttrgEntries (id, attrNum, out numEntries);
  Or
   status = CDFgetNumAttrgEntries (id, attrNum, &numEntries);
  for (i=0; i < numEntries; i++) {
     .
     /* process an entry */
    .
   }
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.33 CDFgetNumAttributes

```
int CDFgetNumAttributes (                      /*  out --  Completion status code.  */
void*  id,                                     /*  in --  CDF  identifier.  */
TYPE  numAttrs);                               /*  out --  Total number of attributes.  */
                                               /*  TYPE --  int* or “out int”.  */
```

CDFgetNumAttributes returns the total number of global and variable attributes in a CDF.

The arguments to CDFgetNumAttributes are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
numAttrs	The total number of global and variable attributes.

4.4.33.1. Example(s)

The following example returns the total number of global and variable attributes in a CDF.

```
.
.
```

```
int     status;                                  /*  Returned status code.  */
void*           id;                              /*  CDF identifier.  */
int     numAttrs;                                /*  Number of attributes.  */

.
.
try {
    ….
    status = CDFgetNumAttributes (id, out numAttrs);
  Or
    status = CDFgetNumAttributes (id, &numAttrs);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.34 CDFgetNumAttrrEntries

```
int CDFgetNumAttrrEntries (              /*  out --  Completion status code.  */
void*  id,                               /*  in --  CDF  identifier.  */
int  attrNum,                            /*  in --  Attribute number.  */
TYPE  entries);                          /*  out --  Total rEntries.  */
                                         /*  TYPE --  int* or “out int”.  */
```

CDFgetNumAttrrEntries returns the total number of entries (rEntries) written for the rVariables in the specified (variable) attribute of a CDF.

The arguments to CDFgetNumAttrrEntries are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number.
entries	Total rEntries.

4.4.34.1. Example(s)

The following example returns the total number of rEntries from the variable attribute “MY_ATTR” in a CDF.

```
.
.
.
int     status;                          /*  Returned status code.  */
void* id;                                /*  CDF identifier.  */
int attrNum;                             /*  Attribute number.  */
int entries;                             /*  Number of entries.  */
```

```
.
.
try {
    ….
    attrNum = CDFgetAttrNum(id, “MY_ATTR”);
    status = CDFgetNumAttrrEntries (id, attrNum, out entries);
  Or
    status = CDFgetNumAttrrEntries (id, attrNum, &entries);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.35 CDFgetNumAttrzEntries

```
int CDFgetNumAttrzEntries (                 /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  attrNum,                               /*  in --  Attribute number.  */
TYPE  entries);                             /*  out --  Total zEntries.  */
                                            /*  TYPE --  int* or “out int”.  */
```

CDFgetNumAttrzEntries returns the total number of entries (zEntries) written for the zVariables in the specified variable attribute in a CDF.

The arguments to CDFgetNumAttrzEntries are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number.
entries	Total zEntries.

4.4.35.1. Example(s)

The following example returns the total number of zEntries for the variable attribute MY_ATTR in a CDF.

```
.
.
.
int    status;                              /*  Returned status code.  */
void* id;                                   /*  CDF identifier.  */
int attrNum;                                /*  Attribute number.  */
int entries;                                /*  Number of entries.  */
.
.
try {
   ….
```

```
    attrNum = CDFgetAttrNum(id, "MY_ATTR");
    status = CDFgetNumAttrzEntries (id, attrNum, out entries);
  Or
    status = CDFgetNumAttrzEntries (id, attrNum, &entries);
.
} catch (CDFException ex) {
   …
}.
.
```

4.4.36 CDFgetNumgAttributes

```
int CDFgetNumgAttributes (                  /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
TYPE  numAttrs);                            /*  out --  Total number of global attributes.  */
                                            /*  TYPE --  int* or "out int".  */
```

CDFgetNumgAttributes returns the total number of global attributes in a CDF.

The arguments to CDFgetNumgAttributes are defined as follows:

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

4.4.36.1. Example(s)

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

```
.
.
.
int     status;                             /*  Returned status code.  */
void* id;                                   /*  CDF identifier.  */
int numAttrs;                               /*  Number of global attributes.  */

.
.
try {
    ….
    status = CDFgetNumgAttributes (id, out numAttrs);
  Or
  . status = CDFgetNumgAttributes (id, &numAttrs);

} catch (CDFException ex) {
   …
}.
.
```

4.4.37 CDFgetNumvAttributes

```
int CDFgetNumvAttributes (                  /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
TYPE  numAttrs);                            /*  out --  Total number of variable attributes.  */
                                            /*  TYPE --  int* or "out int".  */
```

CDFgetNumvAttributes returns the total number of variable attributes in a CDF.

The arguments to CDFgetNumvAttributes are defined as follows:

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

4.4.37.1. Example(s)

The following example returns the total number of variable attributes of a CDF.

```
.
.
.
int     status;                             /*  Returned status code.  */
void* id;                                   /*  CDF identifier.  */
int numAttrs;                               /*  Number of variable attributes.  */

.
.
try {
    ….
    status = CDFgetNumvAttributes (id, out numAttrs);
  Or
    status = CDFgetNumvAttributes (id, &numAttrs);

} catch (CDFException ex) {
   …
}.
.
```

4.4.38 CDFinquireAttr

```
int CDFinquireAttr(                         /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
```

```
int  attrNum,                          /*  in --  Attribute number.  */
out string  attrName,                  /*  out --  Attribute name.  */
TYPE  attrScope,                               /*  out -- Attribute scope.  */
TYPE  maxgEntry,                       /*  out --  Maximum  gEntry number.  */
TYPE  maxrEntry,                       /*  out --  Maximum  rEntry number.  */
TYPE  maxzEntry);                      /*  out --  Maximum  zEntry number.  */
                                       /*  TYPE --  int* or "out int".  */
```

CDFinquireAttr is used to inquire information about the specified attribute. This method expands the method CDFattrInquire to provide an extra information about zEntry if the attribute has a variable scope.

The arguments to CDFinquireAttr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number to inquire. This number may be determined with a call to CDFgetAttrNum.
attrName	The attribute's name that corresponds to attrNum. This string length is limited to CDF_ATTR_NAME_LEN256.
attrScope	The scope of the attribute (GLOBAL_SCOPE or VARIABLE_SCOPE). Attribute scopes are defined in Section 2.13.
maxgEntry	For vAttributes, this value of this field is -1 as it doesn't apply to global attribute entry (gEntry). For gAttributes, this is the maximum entry (gentry) number used. This number may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.
maxrEntry	For gAttributes, this value of this field is -1 as it doesn't apply to rVariable attribute entry (rEntry). For vAttributes, this is the maximum rVariable attribute entry (rEntry) number used. This number may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.
maxzEntry	For gAttributes, this value of this field is -1 as it doesn't apply to zVariable attribute entry (zEntry). For vAttributes, this is the maximum zVariable attribute entry (zEntry) number used. This may not correspond with the number of entries (if some entry numbers were not used). If no entries exist for the attribute, then the value of -1 is returned.

4.4.38.1. Example(s)

The following example displays the name of each attribute in a CDF. The number of attributes in the CDF is first determined by calling the method CDFinquireCDF. Note that attribute numbers start at zero (0) and are consecutive.

```
.
.
.
void*          id;                     /*  CDF  identifier.  */
int    status;                         /*  Returned status code.  */
int    numDims;                        /*  Number of dimensions.  */
int[]  dimSizes;                       /*  Dimension sizes (allocate to allow the
                                           maximum number of dimensions).  */
```

```
int    encoding;                                   /*  Data encoding.  */
int    majority;                                   /*  Variable majority.  */
int    maxRec;                                     /*  Maximum record number in CDF.  */
int    numVars;                                    /*  Number of variables in CDF.  */
int    numAttrs;                                   /*  Number of attributes in CDF.  */
int    attrN;                                      /* attribute number.  */
string         attrName;                           /* attribute name.  */
int            attrScope;                          /* attribute scope.  */
int    maxgEntry, maxrEntry, maxzEntry;            /*  Maximum entry numbers.  */
.
.
try {
   ….
   status =  CDFinquireCDF  (id,  out numDims,  out dimSizes,  out encoding,  out majority, out maxRec,
                             out numVars,  out numAttrs);
   for (attrN  =  0; attrN  <  (int)numAttrs; attrN++)  {
     status = CDFinquireAttr (id, attrN, out attrName,  out attrScope,  out maxgEntry, out maxrEntry,
                             out maxzEntry);
    Or
     status = CDFinquireAttr (id, attrN, out attrName,  &attrScope,  &maxgEntry, &maxrEntry, &maxzEntry);
  }
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.39 CDFinquireAttrgEntry

```
int CDFinquireAttrgEntry (                      /*  out --  Completion status code.  */
void*  id,                                      /*  in --  CDF  identifier.  */
int  attrNum,                                   /*  in --  attribute number.  */
int  entryNum,                                  /*  in --  Entry number.  */
TYPE  dataType,                                         /*  out --  Data type.  */
TYPE  numElements);                             /*  out --  Number of elements  (of the data type).  */
                                                /*  TYPE --  int* or “out int”.  */
```

This method is identical to CDFattrEntryInquire. CDFinquireAttrgEntry is used to inquire information about a global attribute entry.

The arguments to CDFinquireAttrgEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number to inquire. This number may be determined with a call to CDFgetAttrNum.
entryNum	The entry number to inquire.
dataType	The data type of the specified entry. The data types are defined in Section 2.6.

numElements	The number of elements of the data type. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

4.4.39.1. Example(s)

The following example returns each entry for a global attribute named TITLE. Note that entry numbers need not be consecutive - not every entry number between zero (0) and the maximum entry number must exist. For this reason NO_SUCH_ENTRY is an expected error code.

```
.
.
.
void*          id;                  /* CDF identifier. */
Int    status;                      /* Returned status code. */
int    attrN;                       /* attribute number. */
int    entryN;                      /* Entry number. */
string         attrName;            /* attribute name. */
int    attrScope;                   /* attribute scope. */
int    maxEntry;                    /* Maximum entry number used. */
int    dataType;                    /* Data type. */
int    numElems;                    /* Number of elements (of the data type). */
.
.
try {
   ….
   attrN = CDFgetAttrNum (id, "TITLE");
   status = CDFattrInquire (id, attrN, attrName, out attrScope, out maxEntry);
   for (entryN = 0; entryN <= maxEntry; entryN++) {
      status = CDFinquireAttrgEntry (id, attrN, entryN, out dataType, out numElems);
     Or
      status = CDFinquireAttrgEntry (id, attrN, entryN, &dataType, &numElems);
      /* process entries */
          .
          .
   }
} catch (CDFException ex) {
  …
}.
```

4.4.40 CDFinquireAttrrEntry

```
int CDFinquireAttrrEntry (
void* id,                          /* out -- Completion status code. */
int attrNum,                       /* in -- CDF identifier. */
int entryNum,                      /* in -- Attribute number. */
TYPE dataType,                     /* in -- Entry number. */
TYPE numElements);                         /* out -- Data type. */
                                   /* out -- Number of elements (of the data type). */
                                   /* TYPE -- int* or "out int". */
```

This method is identical to the method CDFattrEntryInquire. CDFinquireAttrrEntry is used to inquire about an rVariable's attribute entry.

The arguments to CDFinquireAttrrEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number to inquire. This number may be determined with a call to CDFgetAttrNum.
entryNum	The entry number to inquire. This is the rVariable number (the rVariable being described in some way by the rEntry).
dataType	The data type of the specified entry. The data types are defined in Section 2.6.
numElements	The number of elements of the data type. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

4.4.40.1. Example(s)

The following example determines the data type of the "UNITS" attribute for the rVariable "Temperature", then retrieves and displays the value of the UNITS attribute.

```
.
.
.
void*          id;                    /* CDF identifier. */
Int    status;                        /* Returned status code. */
int    attrN;                         /* Attribute number. */
int    entryN;                        /* Entry number. */
int    dataType;                      /* Data type. */
int    numElems;                      /* Number of elements (of the data type). */
.
.
try {
   ….
   attrN = CDFgetAttrNum (id, "UNITS");
   entryN = CDFgetVarNum(id, "Temperature")
   status = CDFinquireAttrrEntry (id, attrN, entryN, out dataType, out numElems);
 Or
   status = CDFinquireAttrrEntry (id, attrN, entryN, &dataType, &numElems);
   if (dataType == CDF_CHAR) {
       string buffer;
       status = CDFgetAttrrEntry (id, attrN, entryN, out buffer);

   }
.
} catch (CDFException ex) {
  …
}.
```

.

4.4.41 CDFinquireAttrzEntry

```
int CDFinquireAttrzEntry (                  /* out -- Completion status code. */
void* id,                                   /* in -- CDF identifier. */
int attrNum,                                /* in -- (Variable) Attribute number. */
int entryNum,                               /* in -- zEntry number. */
TYPE dataType,                                    /* out -- Data type. */
TYPE numElements);                          /* out -- Number of elements (of the data type). */
                                            /* TYPE -- int* or "out int". */
```

CDFinquireAttrzEntry is used to inquire about a zVariable's attribute entry.

The arguments to CDFinquireAttrzEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The (variable) attribute number for which to inquire an entry. This number may be determined with a call to CDFgetAttrNum (see Section 4.4.24).
entryNum	The entry number to inquire. This is the zVariable number (the zVariable being described in some way by the zEntry).
dataType	The data type of the specified entry. The data types are defined in Section 2.6.
numElements	The number of elements of the data type. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

4.4.41.1. Example(s)

The following example determines the data type of the UNITS attribute for the zVariable Temperature, then retrieves and displays the value of the UNITS attribute.

```
.
.
.
void*       id;                             /* CDF identifier. */
int    status;                              /* Returned status code. */
int    attrN;                               /* attribute number. */
int    entryN;                              /* Entry number. */
int    dataType;                            /* Data type. */
int    numElems;                            /* Number of elements (of the data type). */
.
.
try {
   ....
   attrN = CDFgetAttrNum (id, "UNITS");
```

```
      entryN = CDFgetVarNum (id, "Temperature")

      status = CDFinquireAttrzEntry (id, attrN, entryN,  out dataType,  out numElems);
    Or
      status = CDFinquireAttrzEntry (id, attrN, entryN,  &dataType,  &numElems);
      if (dataType == CDF_CHAR) {
          string buffer ;
          status =  CDFgetAttrzEntry  (id, attrN, entryN,  out buffer);
      }
  } catch (CDFException ex) {
    …
  }.
.
```

4.4.42 CDFputAttrgEntry

```
int CDFputAttrgEntry(                 /*  out --  Completion status code.  */
void*  id,                            /*  in --  CDF  identifier.  */
int  attrNum,                         /*  in --  Attribute number.  */
int  entryNum,                        /*  in --  Attribute entry number.  */
int  dataType,                        /*  in --  Data type of this entry.  */
int  numElements,                     /*  in --  Number of elements in the entry (of the data type).  */
TYPE  value);                         /*  in --  Attribute entry value.  */
                                      /*  TYPE --  void*, string or object.  */
```

CDFputAttrgEntry is used to write a global attribute entry. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry. A global attribute can have one or more attribute entries.

The arguments to CDFputAttrgEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The attribute entry number.
dataType	The data type of the specified entry. Specify one of the data types defined in Section 2.6.
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.

4.4.42.1. Example(s)

The following example writes a global attribute entry to the global attribute called TITLE.

```
.
.
.
void*           id;                                  /*  CDF  identifier.  */
int    status;                                       /*  Returned status code.  */
int    entryNum;                                     /*  Attribute entry number.  */
string          title  =  "CDF  title.";             /*  Value of TITLE attribute. */

.
.
entryNum  =  0;
try {
   ….
   status =  CDFputAttrgEntry  (id,  CDFgetAttrNum(id,"TITLE"), entryNum,  CDF_CHAR, title.Length,  title);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.43 CDFputAttrrEntry

```
int CDFputAttrrEntry(                  /*  out --  Completion status code.  */
void*  id,                             /*  in --  CDF  identifier.  */
int  attrNum,                          /*  in --  Attribute number.  */
int  entryNum,                         /*  in –  Attribute entry number.  */
int  dataType,                         /*  in --  Data type.  */
int  numElems,                         /*  in --  Number of elements in the entry.  */
TYPE  value);                          /*  in --  tribute entry value.  */
                                       /*  TYPE --  void*, string or object.  */
```

This method is identical to the method CDFattrPut. CDFputAttrrEntry is used to write rVariable's attribute entry. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry.

The arguments to CDFputAttrrEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The attribute number. This number may be determined with a call to CDFgetAttrNum.
entryNum	The attribute entry number that is the rVariable number to which this attribute entry belongs.
dataType	The data type of the specified entry. Specify one of the data types defined in Section 2.6.

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.

4.4.43.1. Example(s)

The following example writes to the variable scope attribute VALIDs for the entry, of two elements, that corresponds to the rVariable TMP.

```
.
.
.
void*           id;                                   /* CDF identifier. */
int    status;                                        /* Returned status code. */
int    entryNum;                                      /* Entry number. */
int    numElements;                                   /* Number of elements (of data type). */
short[]         TMPvalids  = new short[] {15,30};     /* Value(s) of VALIDs attribute,
                                                         rEntry for rVariable TMP. */
.
numElements = 2;
try {
   ….
   status = CDFputAttrrEntry (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
                              CDF_INT2, numElements, TMPvalids);
 Or
   fixed (void* pTMPvalids = TMPvalids) {
     status = CDFputAttrrEntry (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
                                CDF_INT2, numElements, pTMPvalids);
   }

.
} catch (CDFException ex) {
  …
}.
.
```

4.4.44 CDFputAttrzEntry

```
int CDFputAttrzEntry(                 /* out -- Completion status code. */
void* id,                             /* in -- CDF identifier. */
int attrNum,                          /* in -- Attribute number. */
int entryNum,                         /* in -- Attribute entry number. */
int dataType,                         /* in -- Data type of this entry. */
int numElements,                      /* in -- Number of elements in the entry (of the data type). */
TYPE value);                          /* in -- Attribute entry value. */
                                      /* TYPE -- void*, string or object. */
```

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

The arguments to CDFputAttrzEntry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The (variable) attribute number. This number may be determined with a call to CDFgetAttrNum (see Section 4.4.24).
entryNum	The entry number that is the zVariable number to which this attribute entry belongs.
dataType	The data type of the specified entry. Specify one of the data types defined in Section 2.6.
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.

4.4.44.1. Example(s)

The following example writes a zVariable's attribute entry. The entry has two elements (that is two values for non-CDF_CHAR type). The zEntry in the variable scope attribute VALIDs corresponds to the zVariable TMP.

```
.
.
.
void*          id;                                          /* CDF identifier. */
int    status;                                              /* Returned status code. */
int    numElements;                                         /* Number of elements (of data type). */
short[]        TMPvalids = new short[] {15,30};             /* Value(s) of VALIDs attribute,
                                                               zEntry for zVariable TMP. */
.
.
numElements = 2;
try {
   ….
   status = CDFputAttrzEntry (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
                              CDF_INT2, numElements, TMPvalids);
 Or
   fixed (void* pTMPvalids = TMPvalids) {
     status = CDFputAttrzEntry (id, CDFgetAttrNum(id,"VALIDs"), CDFgetVarNum(id,"TMP"),
                                CDF_INT2, numElements, pTMPvalids);
   }
.
} catch (CDFException ex) {
  …
}.
```

.

4.4.45 CDFrenameAttr

```
int CDFrenameAttr(                          /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  attrNum,                               /*  in --  Attribute number.  */
string  attrName);                          /*  in --  New attribute name.  */
```

This method is identical to method CDFattrRename. CDFrenameAttr renames an existing attribute.

4.4.45.1. Example(s)

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

```
.
.
.
void*            id;                        /*  CDF  identifier.  */
int    status;                              /*  Returned status code.  */
.
.
try {
   ….
   status =  CDFrenameAttr (id,  CDFgetAttrNum(id,"LAT"),  "LATITUDE");
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.46 CDFsetAttrgEntryDataSpec

```
int CDFsetAttrgEntryDataSpec (              /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  attrNum,                               /*  in --  Attribute number.  */
int  entryNum,                              /*  in --  gEntry number.  */
int  dataType)                              /*  in --  Data type.  */
```

CDFsetAttrgEntryDataSpec respecifies the data type of a gEntry of a global attribute in a CDF. The new and old data type must be equivalent. Refer to the CDF User's Guide for descriptions of equivalent data types.

The arguments to CDFsetAttrgEntryDataSpec are defined as follows:

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

attrNum	The global attribute number.
entryNum	The gEntry number.
dataType	The new data type.

4.4.46.1. Example(s)

The following example modifies the third entry's (entry number 2) data type of the global attribute MY_ATTR in a CDF. It will change its original data type from CDF_INT2 to CDF_UINT2.

```
.
.
.
void* id;                                   /*  CDF identifier.  */
int    status;                              /*  Returned status code.  */
int    entryNum;                            /*  gEntry number.  */
int    dataType;                            /*  The new data type */
.
.
entryNum = 2;
dataType = CDF_UINT2;
numElems = 1;
try {
   ….
   status = CDFsetAttrgEntryDataSpec (id, CDFgetAttrNum(id, "MY_ATTR"), entryNum, dataType);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.47 CDFsetAttrrEntryDataSpec

```
int CDFsetAttrrEntryDataSpec (              /*  out --  Completion status code.  */
void*  id,                                  /*  in --  CDF  identifier.  */
int  attrNum,                               /*  in --  Attribute number.  */
int  entryNum,                              /*  in --  rEntry number.  */
int  dataType,                              /*  in --  Data type.  */
int  numElements);                          /*  in --  Number of elements.  */
```

CDFsetAttrrEntryDataSpec respecifies the data specification (data type and number of elements) of an rEntry of a variable attribute in a CDF. The new and old data type must be equivalent, and the number of elements must not be changed. Refer to the CDF User's Guide for descriptions of equivalent data types.

The arguments to CDFsetAttrrEntryDataSpec are defined as follows:

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

attrNum	The variable attribute number.
entryNum	The rEntry number.
dataType	The new data type.
numElements	The new number of elements.

4.4.47.1. Example(s)

The following example modifies the data specification for an rEntry, corresponding to rVariable "MY_VAR", in the variable attribute "MY_ATTR" in a CDF. It will change its original data type from CDF_INT2 to CDF_UINT2.

```
.
.
.
void*           id;                                   /*  CDF identifier.  */
int    status;                                        /*  Returned status code.  */
int    dataType, numElements;                         /*  Data type and number of elements.  */
.
.
dataType = CDF_UINT2;
numElems = 1;
try {
   ….
   status = CDFsetAttrrEntryDataSpec (id, CDFgetAttrNum(id, "MY_ATTR"), CDFgetVarNum(id, "MY_VAR"),
                                      dataType, numElems);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.48 CDFsetAttrScope

```
int CDFsetAttrScope (                                 /*  out --  Completion status code.  */
void*  id,                                            /*  in --  CDF  identifier.  */
int  attrNum,                                         /*  in --  Attribute number.  */
int  scope);                                          /*  in --  Attribute scope.  */
```

CDFsetAttrScope respecifies the scope of an attribute in a CDF. Specify one of the scopes described in Section 2.13. Global-scoped attributes will contain only gEntries, while variable-scoped attributes can hold rEntries and zEntries.

The arguments to CDFsetAttrScope are defined as follows:

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

scope	The new attribute scope. The value should be either VARIABLE_SCOPE or GLOBAL_SCOPE.

4.4.48.1. Example(s)

The following example changes the scope of the global attribute named MY_ATTR to a variable attribute (VARIABLE_SCOPE).

```
.
.
.
void*          id;                                   /*  CDF identifier.  */
int    status;                                       /*  Returned status code.  */
int    scope;                                        /*  New attribute scope.  */
.
.
scope = VARIABLE_SCOPE;
try {
   ….
   status = CDFsetAttrScope (id, CDFgetAttrNum(id, “MY_ATTR”), scope);
.
} catch (CDFException ex) {
  …
}.
.
```

4.4.49 CDFsetAttrzEntryDataSpec

```
int CDFsetAttrzEntryDataSpec (                      /*  out --  Completion status code.  */
void*  id,                                          /*  in --  CDF  identifier.  */
int  attrNum,                                       /*  in --  Attribute number.  */
int  entryNum,                                      /*  in --  zEntry number.  */
int  dataType)                                      /*  in --  Data type.  */
```

CDFsetAttrzEntryDataSpec modifies the data type of a zEntry of a variable attribute in a CDF. The new and old data type must be equivalent. Refer to the CDF User’s Guide for the description of equivalent data types.

The arguments to CDFsetAttrzEntryDataSpec are defined as follows:

id	The identifier of the current CDF. This identifier must have been initialized by a call to CDFcreate (or CDFcreateCDF) or CDFopenCDF.
attrNum	The variable attribute number.
entryNum	The zEntry number that is the zVariable number.
dataType	The new data type.

4.4.49.1. Example(s)

The following example respecifies the data type of the attribute entry of the attribute named MY_ATTR that is associated with the zVariable MY_VAR. It will change its original data type from CDF_INT2 to CDF_UINT2.

```
.
.
.
void*           id;                                   /*  CDF identifier.  */
int     status;                                       /*  Returned status code.  */
int     dataType;                                     /*  Data type and number of elements.  */
.
.
try {
    ….
    dataType = CDF_UINT2;
    numElems = 1;
    status = CDFsetAttrzEntryDataSpec (id, CDFgetAttrNum(id, “MY_ATTR”),
                                       CDFgetVarNum(id, “MY_VAR”), dataType);
.
. } catch (CDFException ex) {
   …
}.
```

Chapter 5

5 Interpreting CDF Status Codes

Most CDF APIs return a status code of type int. The symbolic names for these codes are defined in CDFException.cs 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 API 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 for most cases are error codes, thus an exception might be thrown.

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

```
int status;

.
try {
.
   status =  CDFfunction (...);              /* any CDF function returning  int */
.
} catch (CDFException ex)  {
    ….
}
```

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.

```
int status = ex.GetCurrentStatus();
string errorMsg = ex.GetStatusMsg(status);
```

Explanations for all CDF status codes are available to your applications through the method CDFerror. CDFerror encodes in a text string an explanation of a given status code.

Chapter 6

6 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. The Concepts chapter in the CDF User's Guide describes EPOCH values. All these APIs are defined as static methods in **CDFAPIs** class. 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.

6.1 computeEPOCH

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

double[] computeEPOCH(                  /* out -- Array of CDF_EPOCH values. */
int[] year,                             /* in -- Years (AD, e.g., 1994). */
int[] month,                            /* in -- Months (1-12). */
int[] day,                              /* in -- Days (1-31). */
int[] hour,                             /* in -- Hours (0-23). */
int[] minute,                           /* in -- Minutes (0-59). */
int[] second,                           /* in -- Seconds (0-59). */
int[] msec);                            /* in -- Milliseconds (0-999). */
```

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

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.

6.2 EPOCHbreakdown

```
void EPOCHbreakdown(
double epoch,                          /* in -- The CDF_EPOCH value. */
TYPE year,                             /* out -- Year (AD, e.g., 1994). */
TYPE month,                            /* out -- Month (1-12). */
TYPE day,                              /* out -- Day (1-31). */
TYPE hour,                             /* out -- Hour (0-23). */
TYPE minute,                           /* out -- Minute (0-59). */
TYPE second,                           /* out -- Second (0-59). */
TYPE msec);                            /* out -- Millisecond (0-999). */
                                       /* TYPE -- int* or "out int". */

void EPOCHbreakdown(
double[] epoch,                        /* in -- Array of CDF_EPOCH values. */
out int[] year,                        /* out -- Years (AD, e.g., 1994). */
out int[] month,                       /* out -- Months (1-12). */
out int[] day,                         /* out -- Days (1-31). */
out int[] hour,                        /* out -- Hours (0-23). */
out int[] minute,                      /* out -- Minutes (0-59). */
out int[] second,                      /* out -- Seconds (0-59). */
out int[] msec);                       /* out -- Milliseconds (0-999). */
```

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

6.3 encodeEPOCH

```
void encodeEPOCH(
double epoch;                          /* in -- The CDF_EPOCH value. */
out string epString;                   /* out -- The standard date/time string. */

void encodeEPOCH(
double[] epoch;                        /* in -- Array CDF_EPOCH values. */
out string[] epString;                 /* out -- The standard date/time strings. */
```

encodeEPOCH encodes a single or 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).

6.4 encodeEPOCH1

```
void  encodeEPOCH1(
double  epoch;                                  /*  in  --  The CDF_EPOCH  value.  */
out string  epString)                           /*  out  --  The alternate  date/time string.  */
```

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).

6.5 encodeEPOCH2

```
void  encodeEPOCH2(
double  epoch;                                  /*  in  --  The CDF_EPOCH  value.  */
out string  epString)                           /*  out  --  The alternate  date/time string.  */
```

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).

6.6 encodeEPOCH3

```
void  encodeEPOCH3(
double  epoch;                                  /*  in  --  The CDF_EPOCH  value.  */
out string  epString)                           /*  out  --  The alternate  date/time string.  */
```

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).

6.7 encodeEPOCH4

```
void  encodeEPOCH4(
double  epoch;                                  /*  in  --  The CDF_EPOCH  value.  */
out string  epString)                           /*  out  --  The ISO 8601  date/time string.  */
```

encodeEPOCH3 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).

6.8 encodeEPOCHx

```
void  encodeEPOCHx(
double  epoch;                          /*  in  --  The CDF_EPOCH  value.  */
string  format;                         /*  in  --  The format string.  */
out string  encoded);                   /*  out  --  The custom  date/time string.  */
```

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.

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 6.3) would be. . .

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

6.9 parseEPOCH

```
double  parseEPOCH(                                  /* out -- CDF_EPOCH  value returned. */
string  epString);                                   /* in  -- The standard  date/time string. */

double[]  parseEPOCH(                                /* out -- Array of CDF_EPOCH  values */
string[]  epString);                                 /* in  -- The standard  date/time strings. */
```

parseEPOCH parses a single or an array of standard date/time character string(s) and returns a CDF_EPOCH value(s). The format of the string is that produced by the encodeEPOCH method described in Section 6.3. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

6.10 parseEPOCH1

```
double  parseEPOCH1(                                 /* out -- CDF_EPOCH  value returned. */
string  epString);                                   /* in  -- The alternate  date/time string. */
```

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 method described in Section 6.4. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

6.11 parseEPOCH2

```
double  parseEPOCH2(                                 /* out -- CDF_EPOCH  value returned. */
string  epString);                                   /* in  -- The alternate  date/time string. */
```

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 method described in Section 6.5. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

6.12 parseEPOCH3

```
double  parseEPOCH3(                                 /* out -- CDF_EPOCH  value returned. */
string  epString);                                   /* in  -- The alternate  date/time string. */
```

parseEPOCH3 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH3 method described in Section 6.6. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

6.13 parseEPOCH4

```
double  parseEPOCH4(                                 /* out -- CDF_EPOCH  value returned. */
```

```
string  epString);                    /* in -- The alternate  date/time string.  */
```

parseEPOCH3 parses an alternate, ISO 8601 date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH3 method described in Section 6.7. If an illegal field is detected in the string the value returned will be ILLEGAL_EPOCH_VALUE.

6.14 computeEPOCH16

```
double  computeEPOCH16(               /* out -- status code returned.  */
int  year,                            /* in -- Year (AD, e.g., 1994).  */
int  month,                           /* in -- Month (1-12).  */
int  day,                             /* in -- Day (1-31).  */
int  hour,                            /* in -- Hour (0-23).  */
int  minute,                          /* in -- Minute (0-59).  */
int  second,                          /* in -- Second (0-59).  */
int  msec,                            /* in -- Millisecond (0-999).  */
int  microsec,                        /* in -- Microsecond (0-999).  */
int  nanosec,                         /* in -- Nanosecond (0-999).  */
int  picosec,                         /* in -- Picosecond (0-999).  */
TYPE  epoch);                         /* out -- CDF_EPOCH16 value returned  */
                                      /* TYPE -- double* or "out double[]".  */
```

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

6.15 EPOCH16breakdown

```
void  EPOCH16breakdown(
double[]  epoch,                      /* in -- The CDF_EPOCH16  value.  */
TYPE  year,                           /* out -- Year (AD, e.g., 1994).  */
TYPE  month,                          /* out -- Month (1-12).  */
TYPE  day,                            /* out -- Day (1-31).  */
TYPE  hour,                           /* out -- Hour (0-23).  */
TYPE  minute,                         /* out -- Minute (0-59).  */
TYPE  second,                         /* out -- Second (0-59).  */
TYPE  msec,                           /* out -- Millisecond (0-999).  */
TYPE  microsec,                               /* out -- Microsecond (0-999).  */
TYPE  nanosec,                        /* out -- Nanosecond (0-999).  */
TYPE  picosec);                       /* out -- Picosecond (0-999).  */
                                      /* TYPE -- int* or "out int".  */
```

EPOCH16breakdown decomposes a CDF_EPOCH16 value into the individual components.

6.16 encodeEPOCH16

```
void encodeEPOCH16(
double[] epoch,                          /* in -- The CDF_EPOCH16 value. */
out string epString);                    /* out -- The date/time string. */
```

encodeEPOCH16 encodes a CDF_EPOCH16 value into the standard date/time character string. The format of the string is **dd-mmm-yyyy hh:mm:ss.mmm:uuu:nnn:ppp** where dd is the day of the month (1-31), mmm is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), yyyy is the year, hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

6.17 encodeEPOCH16_1

```
void encodeEPOCH16_1(
double[] epoch,                          /* in -- The CDF_EPOCH16 value. */
out string epString);                    /* out -- The date/time string. */
```

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).

6.18 encodeEPOCH16_2

```
void encodeEPOCH16_2(
double[] epoch,                          /* in -- The CDF_EPOCH16 value. */
out string epString);                    /* out -- The date/time string. */
```

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).

6.19 encodeEPOCH16_3

```
void encodeEPOCH16_3(
double[] epoch,                          /* in -- The CDF_EPOCH16 value. */
out string epString);                    /* out -- The alternate date/time string. */
```

encodeEPOCH16_3 encodes a CDF_EPOCH16 value into an alternate date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.mmm: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), mm is the minute (0-59), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

6.20 encodeEPOCH16_4

```
void  encodeEPOCH16_4(
double[]  epoch,                                  /*  in  --  The CDF_EPOCH16  value.  */
out string  epString);                            /*  out  --  The alternate  date/time string.  */
```

encodeEPOCH16_3 encodes a CDF_EPOCH16 value into an alternate, ISO 8601 date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.mmmuuunnnppp where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

6.21 encodeEPOCH16_x

```
void  encodeEPOCH16_x(
double[]  epoch,                                  /*  in  --  The CDF_EPOCH16  value.  */
string  format;                                   /*  in  --  The format string.  */
out string  encoded);                             /*  out  --  The date/time string.  */
```

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.

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>

6.22 parseEPOCH16

```
double parseEPOCH16(                    /* out -- The status code returned. */
string epString,                        /* in -- The date/time string. */
TYPE epoch);                            /* out -- The CDF_EPOCH16 value returned */
                                        /* TYPE -- double* or "out double[]". */
```

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.

6.23 parseEPOCH16_1

```
double parseEPOCH16_1(                  /* out -- The status code returned. */
string epString,                        /* in -- The date/time string. */
TYPE epoch);                            /* out -- The CDF_EPOCH16 value returned */
                                        /* TYPE -- double* or "out double[]". */
```

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.

6.24 parseEPOCH16_2

```
double parseEPOCH16_2(                  /* out -- The status code returned. */
string epString,                        /* in -- The date/time string. */
TYPE epoch);                            /* out -- The CDF_EPOCH16 value returned */
                                        /* TYPE -- double* or "out double[]". */
```

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.

6.25 parseEPOCH16_3

```
double  parseEPOCH16_3(                 /*  out  --  The status code returned.  */
string  epString,                       /*  in  --  The date/time string.  */
TYPE  epoch);                           /*  out  --  The CDF_EPOCH16 value returned  */
                                        /*  TYPE  -- double* or “out double[]”.  */
```

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.

6.26 parseEPOCH16_4

```
double  parseEPOCH16_4(                 /*  out  --  The status code returned.  */
string  epString,                       /*  in  --  The ISO 8601 date/time string.  */
TYPE  epoch);                           /*  out  --  The CDF_EPOCH16 value returned  */
                                        /*  TYPE  -- double* or “out double[]”.  */
```

parseEPOCH16_4 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.

7 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 is included in the CDF library. The Concepts chapter in the CDF User's Guide describes TT2000 values. All these APIs are defined as static methods in **CDFAPIs** class. 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.

7.1 ComputeTT2000

compueTT2000 is a overloaded function.

```
long computeTT2000(                     /* out -- CDF_TIME_TT2000 value returned. */
TYPE year,                              /* in -- Year (AD, e.g., 1994), with no fraction. */
TYPE month,                             /* in -- Month (1-12), with no fraction. */
TYPE day);                              /* in -- Day (1-31). */

long computeTT2000(                     /* out -- CDF_TIME_TT2000 value returned. */
TYPE year,                              /* in -- Year (AD, e.g., 1994), with no fraction. */
TYPE month,                             /* in -- Month (1-12), with no fraction. */
TYPE day,                               /* in -- Day (1-31), with no fraction. */
TYPE hour);                             /* in -- Hour (0-23). */

long computeTT2000(                     /* out -- CDF_TIME_TT2000 value returned. */
TYPE year,                              /* in -- Year (AD, e.g., 1994), with no fraction. */
TYPE month,                             /* in -- Month (1-12), with no fraction. */
TYPE day,                               /* in -- Day (1-31), with no fraction. */
TYPE hour,                              /* in -- Hour (0-23), with no fraction. */
TYPE minute);                           /* in -- Minute (0-59). */

long computeTT2000(                     /* out -- CDF_TIME_TT2000 value returned. */
TYPE year,                              /* in -- Year (AD, e.g., 1994), with no fraction. */
TYPE month,                             /* in -- Month (1-12), with no fraction. */
TYPE day,                               /* in -- Day (1-31), with no fraction. */
TYPE hour,                              /* in -- Hour (0-23), with no fraction. */
TYPE minute,                            /* in -- Minute (0-59), with no fraction. */
TYPE second);                           /* in -- Second (0-59 or 0-60 if leap second). */

long computeTT2000(                     /* out -- CDF_TIME_TT2000 value returned. */
TYPE year,                              /* in -- Year (AD, e.g., 1994), with no fraction. */
TYPE month,                             /* in -- Month (1-12), with no fraction. */
TYPE day,                               /* in -- Day (1-31), with no fraction. */
TYPE hour,                              /* in -- Hour (0-23), with no fraction. */
```

```
TYPE  minute,                    /* in -- Minute  (0-59), with no fraction.  */
TYPE  second,                    /* in -- Second  (0-59 or 0-60), with no fraction.  */
TYPE  msec);                     /* in -- Millisecond  (0-999).  */

long  computeTT2000(             /* out -- CDF_TIME_TT2000  value returned.  */
TYPE  year,                      /* in -- Year  (AD,  e.g.,  1994), with no fraction.  */
TYPE  month,                     /* in -- Month  (1-12), with no fraction.  */
TYPE  day,                       /* in -- Day  (1-31), with no fraction.  */
TYPE  hour,                      /* in -- Hour  (0-23), with no fraction.  */
TYPE  minute,                    /* in -- Minute  (0-59), with no fraction.  */
TYPE  second,                    /* in -- Second  (0-59 or 0-60), with no fraction.  */
TYPE  msec,                      /* in -- Millisecond  (0-999), with no fraction.  */
TYPE  usec);                     /* in -- Microsecond  (0-999).  */

long  computeTT2000(             /* out -- CDF_TIME_TT2000  value returned.  */
TYPE  year,                      /* in -- Year  (AD,  e.g.,  1994), with no fraction.  */
TYPE  month,                     /* in -- Month  (1-12), with no fraction.  */
TYPE  day,                       /* in -- Day  (1-31), with no fraction.  */
TYPE  hour,                      /* in -- Hour  (0-23), with no fraction.  */
TYPE  minute,                    /* in -- Minute  (0-59), with no fraction  */
TYPE  second,                    /* in -- Second  (0-59 or 0-60, with no fraction.  */
TYPE  msec,                      /* in -- Millisecond  (0-999), with no fraction.  */
TYPE  usec,                      /* in -- Microsecond  (0-999), with no fraction.  */
TYPE  nsec);                     /* in -- Nanosecond  (0-999).  */
```

Another set of functions (to be added in the future):

```
long[]  computeTT2000(           /* out --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                  /* in -- Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                 /* in -- Months  (1-12), with no fraction .*/
TYPE  []  day);                  /* in -- Days  (1-31).  */

long[]  computeTT2000(           /* out --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                  /* in -- Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                 /* in -- Months  (1-12), with no fraction.  */
TYPE  []  day,                   /* in -- Days  (1-31), with no fraction.  */
TYPE  []  hour);                 /* in -- Hours  (0-23).  */

long[]  computeTT2000(           /* out --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                  /* in -- Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                 /* in -- Months  (1-12), with no fraction.  */
TYPE  []  day,                   /* in -- Days  (1-31), with no fraction.  */
TYPE  []  hour,                  /* in -- Hours  (0-23), with no fraction.  */
TYPE  []  minute);               /* in -- Minute  (0-59).  */

long[]  computeTT2000(           /* out --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                  /* in -- Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                 /* in -- Months  (1-12), with no fraction.  */
TYPE  []  day,                   /* in -- Days  (1-31), with no fraction.  */
TYPE  []  hour,                  /* in -- Hours  (0-23), with no fraction.  */
TYPE  []  minute,                /* in -- Minutes  (0-59), with no fraction.  */
TYPE  []  second);               /* in -- Seconds  (0-59 or 0-60 if leap second).  */

long[]  computeTT2000(           /* out --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                  /* in -- Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                 /* in -- Months  (1-12), with no fraction.  */
```

```
TYPE  []  day,                    /*  in  --  Days  (1-31), with no fraction.  */
TYPE  []  hour,                   /*  in  --  Hours  (0-23), with no fraction.  */
TYPE  []  minute,                 /*  in  --  Minutes  (0-59), with no fraction.  */
TYPE  []  second,                 /*  in  --  Seconds  (0-59 or 0-60) with no fraction.  */
TYPE  []  msec);                  /*  in  --  Milliseconds  (0-999).  */

long[]  computeTT2000(            /*  out  --  Array of CDF_TIME_TT2000  values.  */
TYPE  [] year,                    /*  in  --  Years  (AD,  e.g.,  1994), with no fraction.  */
TYPE  []  month,                  /*  in  --  Months  (1-12), with no fraction.  */
TYPE  []  day,                    /*  in  --  Days  (1-31), with no fraction.  */
TYPE  []  hour,                   /*  in  --  Hours  (0-23), with no fraction  */
TYPE  []  minute,                 /*  in  --  Minutes  (0-59), with no fraction.  */
TYPE  []  second,                 /*  in  --  Seconds  (0-59 or 0-60), with no fraction  */
TYPE  []  msec,                   /*  in  --  Milliseconds  (0-999), with no fraction.  */
TYPE  []  usec);                  /*  in  --  Microseconds  (0-999).  */

long[]  computeTT2000(            /*  out  --  Array of CDF_TIME_TT2000  values.  */
TYPE  []  year,                   /*  in  --  Years  (AD,  e.g.,  1994), with no fraction  */
TYPE  []  month,                  /*  in  --  Months  (1-12), with no fraction.  */
TYPE  []  day,                    /*  in  --  Days  (1-31), with no fraction.  */
TYPE  []  hour,                   /*  in  --  Hours  (0-23), with no fraction.  */
TYPE  []  minute,                 /*  in  --  Minutes  (0-59), with no fraction.  */
TYPE  []  second,                 /*  in  --  Seconds  (0-59 or 0-60). with no fraction  */
TYPE  []  msec,                   /*  in  --  Milliseconds  (0-999), with no fraction.  */
TYPE  []  usec,                   /*  in  --  Microseconds  (0-999), with no fraction.  */
TYPE  []  nsec);                  /*  in  --  Nanoseconds  (0-999), with no fraction.  */
```

Note: TYPE can be either double or long.

computeTT2000 calculates a single or an array of CDF_TIME_TT2000 value(s), from the given individual components. If an illegal component is detected, the value returned will be ILLEGAL_TT2000_VALUE. The day componment can be either in day of the month or day of the year (DOY). For DOY form, the month componment must have a value(s) of one (1).

NOTE: Even though this overloaded function uses double for all its parameter fields, all but the very last parameter can not have a non-zero fractional part for simplifying the computation. An exception will be thrown if the rule is not followed. For example, this call is allowed:

long tt2000 = computeTT2000(2010.0, 10.0, **10.5**);

But, this call will fail:

long tt2000 = computeTT2000(2010.0, 10.0, **10.5, 12.5**);

7.2 TT2000breakdown

```
void  TT2000breakdown(
long  tt2000,                     /*  in  --  The CDF_TIME_TT2000  value.  */
TYPE  year,                       /*  out  --  Year  (AD,  e.g.,  1994).  */
TYPE  month,                      /*  out  --  Month  (1-12).  */
TYPE  day,                        /*  out  --  Day  (1-31).  */
```

```
TYPE  hour,          /* out -- Hour (0-23). */
TYPE  minute,        /* out -- Minute (0-59). */
TYPE  second,        /* out -- Second (0-59 or 0-60 if leap second). */
TYPE  msec;          /* out -- Millisecond (0-999). */
TYPE  usec;          /* out -- Microsecond (0-999). */
TYPE  nsec);         /* out -- Nanosecond (0-999). */
                     /* TYPE -- double* or "out double" or
                                  long* or "out long" */
```

Another function will be added in the future:

```
void  TT2000breakdown(
long[]  tt2000,          /* in -- Array of CDF_TIME_TT2000 values. */
out double[]  year,      /* out -- Years (AD, e.g., 1994). */
out double[]  month,     /* out -- Months (1-12). */
out double[]  day,       /* out -- Days (1-31). */
out double[]  hour,      /* out -- Hours (0-23). */
out double[]  minute,    /* out -- Minutes (0-59). */
out double[]  second,    /* out -- Seconds (0-59 or 0-60 if leap second). */
out double[]  msec;      /* out -- Milliseconds (0-999). */
out double[]  usec;      /* out -- Microseconds (0-999). */
out double[]  nsec);     /* out -- Nanoseconds (0-999). */
```

TT2000breakdown decomposes a single or an array of CDF_TIME_TT2000 value(s) into the individual components.

7.3 EncodeTT2000

EncodeTT2000 is a overloaded function.

```
void  encodeTT2000(
long  tt2000;                /* in -- The CDF_TIME_TT2000 value. */
out string  epString);       /* out -- The standard date/time string. */

void  encodeTT2000(
long[]  tt2000;              /* in -- Array of TT2000 values. */
out string[]  epString);     /* out -- The standard date/time strings. */

void  encodeTT2000(
long  tt2000;                /* in -- The CDF_TIME_TT2000 value. */
out string  epString.        /* out -- The standard date/time string. */
int  format);                /* in -- The encoded string format. */

void  encodeTT2000(
long[]  tt2000;              /* in -- Array of TT2000 values. */
out string[]  epString.      /* out -- The standard date/time strings. */
int  format);                /* in -- The encoded string format. */
```

encodeTT2000 encodes a single or an array of CDF_TIME_TT2000 value(s) into the standard date/time UTC character string(s). If not provided, the default format of the string is in **ISO 8601** format: **yyyy-mm-ddT hh:mm:ss.mmmuuunnn** where yyyy is the year (1707-2292), mm is the month (01-12), dd is the day of the month

(1-31), hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59 or 0-60 if leap second), mmm is the millisecond (0-999), uuu is the microsecond (0-999) and nnn is the nanosecond (0-999).

For a format of value **0**, the encoded UTC string is **DD-Mon-YYYY hh:mm:ss.mmmuuunnn**, where DD is the day of the month (1-31), Mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), YYYY is the year, hh is the hour (0-23), mm is the minute (0-59 or 0-60 if leap second), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999). The encoded string has a length of TT2000_0_STRING_LEN (**30**).

For a format of value **1**, the encoded UTC string is **YYYYMMDD.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). The encoded string has a length of TT2000_1_STRING_LEN (**19**).

For a format of value **2**, the encoded UTC string is **YYYYMMDDhhmmss**, where YYYY is the year, MM is the month (1-12) DD is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59),and ss is the second (0-59 or 0-60 if leap second). The encoded string has a length of TT2000_2_STRING_LEN (**14**).

For a format of value **3**, the encoded UTC string is **YYYY-MM-DDThh:mm:ss.mmmuuunnn**, where YYYY is the year, MM is the month (1-12), DD is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59 or 0-60 if leap second), ss is the second (0-59), mmm is the millisecond (0-999), uuu is the microsecond (0-999), and nnn is the nanosecond (0-999). The encoded string has a length of TT2000_3_STRING_LEN (**29**).

7.4 ParseTT2000

```
long  parseTT2000(                          /*  out  --  CDF_TIME_TT2000 value returned.  */
string  epString);                          /*  in  --  The standard  date/time string.  */

long[]  parseTT2000(                        /*  out  --  Array of TT2000 values.  */
string[]  epString);                        /*  in  --  The standard  date/time strings.  */
```

parseTT2000 parses a ignle or an array of standard date/time character string(s) and returns a CDF_TIME_TT2000 value(s). The format of the string is that produced by the encodeTT2000 method described in Section 6.3. If an illegal field is detected in the string the value returned will be ILLEGAL_TT2000_VALUE.

7.5 CDFgetLastDateinLeapSecondsTable

```
void  CDFgetLastDateinLeapSecondsTable(          /
out int  year;                              /*  out  --   The year.  */
out int month;                              /*  out  --  The month.  */
out int day);                               /*  out  --  The day.  */
```

CDFgetLastDateinLeapSecondsTable returns the last entry in the leap second table used by the CDF processing. This date comes from the leap second table, either through an external text file, or the hard-coded table in the library code. This information can tell whether the leap second table is up-to-date.

8 CDF Utility Methods

Several methods are created that are mainly used to decipher the strings and their corresponding constant values or vice verse. All these APIs are defined as static methods in **CDFUtils** class. The constant values are defined in **CDFConstants** class.

8.1 CDFFileExists

```
bool  CDFFileExists(                    /*  out  --  The  file existence flag.  */
string  fileName)                       /*  in  --  The file name.  */
```

CDFFileExists method checks whether a CDF file by the given file name, with or without the .cdf extension, exists. Even the file exists, CDFFileExists will not be able to verify whether it is a valid one. (Use CDFopen to validate it).

8.2 CDFgetChecksumValue

```
int  CDFgetChecksumValue(               /*  out  --  The checksum value.  */
string  checksum)                       /*  in  --  The file checksum type string.  */
```

CDFgetChecksumValue method returns the corresponding file checksum type value, based on the passed string. The file checksum types and their values are as follows:

Type	**Value**
NONE	NO_CHECKSUM (0)
MD5	MD5_CHECKSUM (1)
OTHER	OTHER_CHECKSUM

8.3 CDFgetCompressionTypeValue

```
int  CDFgetCompressionTypeValue(        /*  out  --  The compression type.  */
string  compressionType)                /*  in  --  The compression type string.  */
```

CDFgetCompressionTypeValue method returns the corresponding compression type value, based on the passed string. The compression types and values are as follows:

Type	**Value**
NONE	NO_COMPRESSION (0)

RLE	RLE_COMPRESSION (1)
Huffman	HUFF_COMPRESSION (2)
Adaptive Huffman	AHUFF_COMPRESSION (3)
GZIP	GZIP_COMPRESSION (5)

8.4 CDFgetDataTypeValue

```
int  CDFgetDataTypeValue(                    /*  out  --  The data type.  */
string  dataType)                            /*  in  --  The data type string.  */
```

CDFgetDataTypeValue method returns the corresponding data type value, based on the passed string. The data types and their values are as follows:

Type	**Value**
CDF_BYTE	CDF_BYTE (41)
CDF_CHAR	CDF_CHAR (51)
CDF_UCHAR	CDF_UCHAR (52)
CDF_INT1	CDF_INT1 (1)
CDF_UINT1	CDF_UINT1 (11)
CDF_INT2	CDF_INT2 (2)
CDF_UINT2	CDF_UINT2 (12)
CDF_INT4	CDF_INT4 (4)
CDF_UINT4	CDF_UINT4 (14)
CDF_INT8	CDF_INT8 (8)
CDF_REAL4	CDF_REAL4 (21)
CDF_FLOAT	CDF_FLOAT (44)
CDF_REAL8	CDF_REAL8 (22)
CDF_DOUBLE	CDF_DOUBLE (45)
CDF_EPOCH	CDF_EPOCH (31)
CDF_EPOCH16	CDF_EPOCH16 (32)
CDF_TIME_TT2000	CDF_TIME_TT2000 (33)

8.5 CDFgetDecodingValue

```
int  CDFgetDecodingValue(                    /*  out  --  The decoding value.  */
string  decoding)                            /*  in  --  The data decoding string.  */
```

CDFgetDecodingValue method returns the corresponding data decoding value, based on the passed string. The data decodings and their values are as follows:

Type	**Value**
NETWORK	NETWORK_DECODING (1)
SUN	SUN_DECODING (2)
VAX	VAX_DECODING (3)
DECSTATION	DECSTATION_DECODING (4)
SGi	SGi_DECODING (5)
IBMPC	IBMPC_DECODING (6)
IBMRS	IBMRS_DECODING (7)

HOST	HOST_DECODING (8)
PPC	PPC_DECODING (9)
HP	HP_DECODING (11)
NeXT	NeXT_DECODING (12)
ALPHAOSF1	ALPHAOSF1_DECODING (13)
ALPHAVMSd	ALPHAVMSd_DECODING (14)
ALPHAVMSg	ALPHAVMSg_DECODING (15)
ALPHAVMSi	ALPHAVMSi_DECODING (16)

8.6 CDFgetEncodingValue

```
int  CDFgetEncodingValue(                    /*  out  --  The encoding value.  */
string  encoding)                            /*  in  --  The data encoding string.  */
```

CDFgetEncodingValue method returns the corresponding data encoding value, based on the passed string. The data encodings and their values are as follows:

Type	**Value**
NETWORK	NETWORK_ENCODING (1)
SUN	SUN_ENCODING (2)
VAX	VAX_ENCODING (3)
DECSTATION	DECSTATION_ENCODING (4)
SGi	SGi_ENCODING (5)
IBMPC	IBMPC_ENCODING (6)
IBMRS	IBMRS_ENCODING (7)
HOST	HOST_ENCODING (8)
PPC	PPC_ENCODING (9)
HP	HP_ENCODING (11)
NeXT	NeXT_ENCODING (12)
ALPHAOSF1	ALPHAOSF1_ENCODING (13)
ALPHAVMSd	ALPHAVMSd_ENCODING (14)
ALPHAVMSg	ALPHAVMSg_ENCODING (15)
ALPHAVMSi	ALPHAVMSi_ENCODING (16)

8.7 CDFgetFormatValue

```
int  CDFgetFormatValue(                      /*  out  --  The format value.  */
string  format)                              /*  in  --  The file format string.  */
```

CDFgetFormatValue method returns the corresponding file format value, based on the passed string. The file formats and their values are as follows:

Type	**Value**
SINGLE`	SINGLE_FILE (1)
MULTI	MULTI_FILE (2)

8.8 CDFgetMajorityValue

```
int  CDFgetMajorityValue(                    /*  out  --  The majority value.  */
string  majority)                            /*  in  --  The data majority string.  */
```

CDFgetMajorityValue method returns the corresponding file majority value, based on the passed string. The file majorities and their values are as follows:

Type	Value
ROW	ROW_MAJOR (1)
COLUMN	COLUMN_MAJOR (2)

8.9 CDFgetSparseRecordValue

```
int  CDFgetSparseRecordValue(                /*  out  --  The sparse record value.  */
string  sparseRecord)                        /*  in  --  The sparse record string.  */
```

CDFgetSparseRecordValue method returns the corresponding sparse record value, based on the passed string. The sparse records types and their values are as follows:

Type	Value
NONE	NO_SPARSERECORDS (0)
PAD	PAD_SPARSERECORDS (1)
PREV	PREV_SPARSERECORDS (2)

8.10 CDFgetStringChecksum

```
string  CDFgetStringChecksum(                /*  out  --  The checksum string.  */
int  checksum)                               /*  in  --  The file checksum type.  */
```

CDFgetStringChecksum method returns the corresponding file checksum string, based on the passed type. The file checksum types and their values are the same as those defined in CDFgetChecksumValue method.

8.11 CDFgetStringCompressionType

```
string  CDFgetStringCompressionType(         /*  out  --  The compression string.  */
int  compressionType)                        /*  in  --  The compression type.  */
```

CDFgetStringCompressionType method returns the corresponding compression type string, based on the passed type. The file checksum types and their values are the same as those defined in CDFgetCompressionTypeValue method.

8.12 CDFgetStringDataType

```
string  CDFgetStringDataType(                    /*  out  --  The data type string.  */
int  dataType)                                   /*  in  --  The data type.  */
```

CDFgetStringDataType method returns the corresponding data type string, based on the passed type. The data types and their values are the same as those in CDFgetDataTypeValue method:

8.13 CDFgetStringDecoding

```
string  CDFgetStringDecoding(                    /*  out  --  The decoding string.  */
in  decoding)                                    /*  in  --  The data decoding type.  */
```

CDFgetStringDecoding method returns the corresponding data decoding string, based on the passed type. The data decodings and their values are as same as those defined in CDFgetDecodingValue:

8.14 CDFgetStringEncoding

```
string  CDFgetStringEncoding(                    /*  out  --  The encoding string.  */
int  encoding)                                   /*  in  --  The data encoding type.  */
```

CDFgetStringEncoding method returns the corresponding data encoding string, based on the passed type. The data encodings and their values are the same as those defined in CDFgetEncodingValue method.

8.15 CDFgetStringFormat

```
string  CDFgetStringFormat(                      /*  out  --  The format string.  */
int  format)                                     /*  in  --  The file format type.  */
```

CDFgetStringFormat method returns the corresponding file format string, based on the passed type. The file formats and their values are the same as those defined in CDFgetFormatValue method.:

8.16 CDFgetStringMajority

```
string  CDFgetStringMajority(                    /*  out  --  The majority string.  */
int  majority)                                   /*  in  --  The data majority type.  */
```

CDFgetStringMajority method returns the corresponding file majority string, based on the passed type. The file majorities and their values are the same as those defined in CDFgetMajorityValue method.

8.17 CDFgetStringSparseRecord

```
string  CDFgetStringSparseRecord(                    /*  out  --  The sparse record string.  */
int  sparseRecord)                                   /*  in  --  The sparse record type.  */
```

CDFgetStringSparseRecord method returns the corresponding sparse record string, based on the passed type. The sparse records types and their values are the same as those defined in CDFgetSparseRecordValue method.:

9 CDF Exception Methods

Several methods in the CDFexception class can be used to check what happens when an exception is thrown by the CDFAPIs, and react to it if necessary. All these APIs are defined as static methods. CDFException inherits from C#'s Exception class.

9.1 CDFgetCurrentStatus

```
int  CDFgetCurrentStatus()                                  /*  out  --  The  status.  */
```

CDFgetCurrentStatus method returns the status when an exception is detected. The status value should be a negative value. Chapter 5 covers all possible status codes. Use the following CDFgetStatusMsg method to decipher what the status means.

9.2 CDFgetStatusMsg

```
string  CDFgetStatusMsg(                                    /*  out  --  The descriptive message.  */
int  status)                                                /*  in  --  The exception status.  */
```

CDFgetStatusMsg method returns the descriptive information of the passed status.

Appendix A

A.1 Introduction

A status code is returned from most CDF functions. The CDFConstants class contains the numerical values (constants) for each of the status codes (and for any other constants referred to in the explanations). The method CDFerror can be used within a program to inquire the explanation text for a given status code.

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 method 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[22]	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_INT	Unknown CDF status code received. The CDF library does not use the status code specified. [Error]
BAD_CHECKSUM	An illegal checksum mode received. It is invalid 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 CDFConstants class for C# applications. [Error]
BAD_DECODING	An unknown decoding was specified. The CDF decodings are defined in CDFConstants class for C# 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]

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

BAD_ENCODING	Unknown data encoding specified. The CDF encodings are defined in CDFConstants class for C# applications. [Error]
BAD_ENTRY_NUM	Illegal attribute entry number specified. Entry numbers must be at least zero (0) for C# 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. [Error]
BAD_FORMAT	Unknown format specified. The CDF formats are defined in CDFConstants class for C# 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 CDFConstants class for C# 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 CDFConstants class for C# 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 CDFConstants class for C# 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 CDFConstants class for C# 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. [Error]

BAD_zMODE

Illegal zMode specified. The CDF zModes are defined in CDFConstants class for C# 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	The 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 the CDF library when comparing variable names ignores trailing blanks. [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 C#-CDF APIs

The overloaded APIs will have the following **TYPE** symbols, which represent a set of possible types for parameters.

- **TYPE** -- **int*** or "**out int**" for output
- **TYPE2** -- **int*** or "**out int[]**" for output
- **TYPE3** -- **void***, **string**, or **object** for input
- **TYPE4** -- **void***, "**out string**", or "**out object**" for output
- **TYPE5** -- **void***, **string**, **string[]**, or **object** for input
- **TYPE6** -- **void***, "**out string**", "**out string[]**" or "**out object**" for output
- **TYPE7** -- **long*** or "**out long**" for output

```
int CDFattrCreate  (id, attrName, attrScope, attrNum)
void*    id;                                          /* in */
string   attrName;                                    /* in */
int      attrScope;                                   /* in */
TYPE     attrNum;                                     /* out */

int CDFattrEntryInquire  (id, attrNum, entryNum, dataType, numElements)
void*    id;                                          /* in */
int      attrNum;                                     /* in */
int      entryNum;                                    /* in */
TYPE     dataType;                                    /* out */
TYPE     numElements;                                 /* out */

int CDFattrGet  (id, attrNum, entryNum,  value)
void*    id;                                          /* in */
int      attrNum;                                     /* in */
int      entryNum;                                    /* in */
TYPE4    value;                                       /* out */

int CDFattrInquire  (id, attrNum, attrName, attrScope,  maxEntry)
void*    id;                                          /* in */
int      attrNum;                                     /* in */
out string       attrName;                           /* out */
TYPE     attrScope;                                   /* out */
TYPE     maxEntry;                                    /* out */

int CDFattrNum  (id, attrName)
void*    id;                                          /* in */
string   attrName;                                    /* in */

int CDFattrPut  (id, attrNum, entryNum, dataType,  numElements, value)
```

```
void*     id;                                   /* in */
int       attrNum;                              /* in */
int       entryNum;                             /* in */
int       dataType;                             /* in */
int       numElements;                          /* in */
TYPE3     value;                                /* in */

int CDFattrRename (id, attrNum, attrName)
void*     id;                                   /* in */
int       attrNum;                              /* in */
string    attrName;                             /* in */

int CDFclose  (id)
void*     id;                                   /* in */

int CDFcloseCDF  (id)
void*     id;                                   /* in */

int CDFcloserVar  (id, varNum)
void*     id;                                   /* in */
int       varNum;                               /* in */

int CDFclosezVar  (id, varNum)
void*     id;                                   /* in */
int       varNum;                               /* in */

int CDFconfirmAttrExistence  (id, attrName)
void*     id;                                   /* in */
string    attrName;                             /* in */

int CDFconfirmgEntryExistence  (id, attrNum, entryNum)
void*     id;                                   /* in */
int       attrNum;                              /* in */
int       entryNum;                             /* in */

int CDFconfirmrEntryExistence  (id, attrNum, entryNum)
void*     id;                                   /* in */
int       attrNum;                              /* in */
int       entryNum;                             /* in */

int CDFconfirmrVarExistence  (id, varNum)
void*     id;                                   /* in */
int       varNum;                               /* in */

int CDFconfirmrVarPadValueExistence  (id, varNum)
void*     id;                                   /* in */
int       varNum;                               /* in */

int CDFconfirmzEntryExistence  (id, attrNum, entryNum)
void*     id;                                   /* in */
int       attrNum;                              /* in */
int       entryNum;                             /* in */

int CDFconfirmzVarExistence  (id, varNum)
void*     id;                                   /* in */
int       varNum;                               /* in */
```

```
int CDFconfirmzVarPadValueExistence  (id, varNum)
void*    id;                                                    /* in */
int      varNum;                                                /* in */

int CDFcreate  (CDFname,  numDims,  dimSizes, encoding,  majority,  id)
string   CDFname;                                               /* in */
int      numDims;                                               /* in */
int[]    dimSizes;                                              /* in */
int      encoding;                                              /* in */
int      majority;                                              /* in */
void*    *id;                                                   /* out */

int CDFcreateAttr  (id, attrName,  scope, attrNum)
void*    id;                                                    /* in */
string   attrName;                                              /* in */
int      scope;                                                 /* in */
TYPE     attrNum;                                               /* out */

int CDFcreateCDF  (CDFname, id)
string   CDFname;                                               /* in */
void*    *id;                                                   /* out */

int CDFcreaterVar  (id, varName, dataType, numElements, recVary, dimVarys, varNum)
void*    id;                                                    /* in */
string   varName;                                               /* in */
int      dataType;                                              /* in */
int      numElements;                                           /* in */
int      recVary;                                               /* in */
int[]    dimVarys;                                              /* in */
TYPE     varNum;                                                /* out */

int CDFcreatezVar  (id, varName, dataType, numElements, numDims, dimSizes, recVary, dimVarys, varNum)
void*    id;                                                    /* in */
string   varName;                                               /* in */
int      dataType;                                              /* in */
int      numElements;                                           /* in */
int      numDims;                                               /* in */
int[]    dimSizes;                                              /* in */
int      recVary;                                               /* in */
int[]    dimVarys;                                              /* in */
TYPE     varNum;                                                /* out */

int CDFdelete  (id)
void*    id;                                                    /* in */

int CDFdeleteAttr  (id, attrNum)
void*    id;                                                    /* in */
int      attrNum;                                               /* in */

int CDFdeleteAttrgEntry  (id, attrNum, entryNum)
void*    id;                                                    /* in */
int      attrNum;                                               /* in */
int      entryNum;                                              /* in */

int CDFdeleteAttrrEntry  (id, attrNum, entryNum)
```

```
void*     id;                                                    /* in */
int       attrNum;                                               /* in */
int       entryNum;                                              /* in */

int CDFdeleteAttrzEntry  (id, attrNum, entryNum)
void*     id;                                                    /* in */
int       attrNum;                                               /* in */
int       entryNum;                                              /* in */

int CDFdeleteCDF  (id)
void*     id;                                                    /* in */

int CDFdeleterVar  (id, varNum)
void*     id;                                                    /* in */
int       varNum;                                                /* in */

int CDFdeleterVarRecords  (id, varNum, startRec, endRec)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       startRec;                                              /* in */
int       endRec;                                                /* in */

int CDFdeletezVar  (id, varNum)
void*     id;                                                    /* in */
int       varNum;                                                /* in */

int CDFdeletezVarRecords  (id, varNum, startRec, endRec)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       startRec;                                              /* in */
int       endRec;                                                /* in */

int CDFdeletezVarRecordsRenumber  (id, varNum, startRec, endRec)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       startRec;                                              /* in */
int       endRec;                                                /* in */

int CDFdoc  (id,  version,  release,  text)
void*     id;                                                    /* in */
TYPE      version;                                               /* out */
TYPE      release;                                               /* out */
out string          text;                                        /* out */

int CDFerror (status,  message)
int       status;                                                /* in */
out string          message;                                     /* out */

int CDFgetAttrgEntry  (id,  attrNum, entryNum, value)
void*     id;                                                    /* in  */
int       attrNum;                                               /* in  */
int       entryNum;                                              /* in  */
TYPE4     value;                                                 /* out */

int CDFgetAttrgEntryDataType  (id,  attrNum, entryNum, dataType)
void*     id;                                                    /* in  */
```

```
int         attrNum;                                        /* in  */
int         entryNum;                                       /* in  */
TYPE        dataType;                                       /* out */

int CDFgetAttrgEntryNumElements  (id,  attrNum, entryNum, numElems)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
int         entryNum;                                       /* in  */
TYPE        numElems;                                       /* out */

int CDFgetAttrMaxgEntry  (id,  attrNum, entryNum)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
TYPE        entryNum;                                       /* out  */

int CDFgetAttrMaxrEntry  (id,  attrNum, entryNum)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
TYPE        entryNum;                                       /* out  */

int CDFgetAttrMaxzEntry  (id,  attrNum, entryNum)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
TYPE        entryNum;                                       /* out  */

int CDFgetAttrName  (id,  attrNum, attrName)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
out string              attrName;                           /* out  */

int CDFgetAttrNum  (id, attrName)                           /* out */
void*       id;                                             /* in  */
string      attrName;                                       /* in  */

int CDFgetAttrrEntry  (id,  attrNum, entryNum, value)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
int         entryNum;                                       /* in  */
TYPE4       value;                                          /* out */

int CDFgetAttrrEntryDataType  (id,  attrNum, entryNum, dataType)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
int         entryNum;                                       /* in  */
TYPE        dataType;                                       /* out */

int CDFgetAttrrEntryNumElements  (id,  attrNum, entryNum, numElems)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
int         entryNum;                                       /* in  */
TYPE        numElems;                                       /* out */

int CDFgetAttrScope  (id,  attrNum, scope)
void*       id;                                             /* in  */
int         attrNum;                                        /* in  */
TYPE        scope;                                          /* out  */
```

```
int CDFgetAttrzEntry  (id,  attrNum, entryNum, value)
void*     id;                                                    /* in  */
int       attrNum;                                               /* in  */
int       entryNum;                                              /* in  */
TYPE4     value;                                                 /* out */

int CDFgetAttrzEntryDataType  (id,  attrNum, entryNum, dataType)
void*     id;                                                    /* in  */
int       attrNum;                                               /* in  */
int       entryNum;                                              /* in  */
TYPE      dataType;                                              /* out */

int CDFgetAttrzEntryNumElements  (id,  attrNum, entryNum, numElems)
void*     id;                                                    /* in  */
int       attrNum;                                               /* in  */
int       entryNum;                                              /* in  */
TYPE      numElems;                                              /* out */

int CDFgetCacheSize  (id,  numBuffers)
void*     id;                                                    /* in  */
TYPE      numBuffers;                                            /* out  */

int CDFgetChecksum  (id,  checksum)
void*     id;                                                    /* in  */
TYPE      checksum;                                              /* out  */

int CDFgetCompression  (id,  compressionType, compressionParms, compressionPercent)
void*     id;                                                    /* in  */
TYPE      compressionType;                                       /* out  */
TYPE2     compressionParms;                                      /* out  */
TYPE      compressionPercent;                                    /* out  */

int CDFgetCompressionCacheSize  (id,  numBuffers)
void*     id;                                                    /* in  */
TYPE      numBuffers;                                            /* out  */

int CDFgetCompressionInfo  (cdfName, compressionType, compressionParms, compressionSize, uncompressionSize)
string    cdfName;                                               /* in  */
TYPE      compressionType;                                       /* out  */
TYPE2     compressionParms;                                      /* out  */
TYPE7     compressionSize;                                       /* out  */
TYPE7     uncompressionSize;                                     /* out  */

int CDFgetCopyright  (id,  copyright)
void*     id;                                                    /* in  */
out string          copyright;                                   /* out  */

int CDFgetDataTypeSize  (dataType, numBytes)
int       dataType;                                              /* in  */
TYPE      numBytes;                                              /* out  */

int CDFgetDecoding  (id, decoding)
void*     id;                                                    /* in  */
TYPE      decoding;                                              /* out  */
```

```
int CDFgetEncoding  (id, encoding)
void*    id;                                                    /* in  */
TYPE     encoding;                                              /* out  */

int CDFgetFileBackward  ()

int CDFgetFormat  (id, format)
void*    id;                                                    /* in  */
TYPE     format;                                                /* out  */

int CDFgetLibraryCopyright  (copyright)
out string       copyright;                                     /* out  */

int CDFgetLibraryVersion  (version, release, increment, subIncrement)
TYPE     version;                                               /* out  */
TYPE     release;                                               /* out  */
TYPE     increment;                                             /* out  */
out string       subIncrement;                                  /* out  */

int CDFgetLeapSecondLastUpdated  (id,  lastUpdated)
void*    id;                                                    /* in  */
TYPE     lastUpdated;                                           /* out  */

int CDFgetMajority  (id,  majority)
void*    id;                                                    /* in  */
TYPE     majority;                                              /* out  */

int CDFgetMaxWrittenRecNums  (id,  maxRecrVars, maxReczVars)
void*    id;                                                    /* in  */
TYPE     maxRecrVars;                                           /* out  */
TYPE     maxReczVars;                                           /* out  */

int CDFgetName  (id,  name)
void*    id;                                                    /* in  */
out string       name;                                          /* out  */

int CDFgetNegtoPosfp0Mode  (id,  negtoPosfp0)
void*    id;                                                    /* in  */
TYPE     negtoPosfp0;                                           /* out  */

int CDFgetNumAttrgEntries  (id,  attrNum, entries)
void*    id;                                                    /* in  */
int      attrNum;                                                       /* in  */
TYPE     entries;                                               /* out  */

int CDFgetNumAttributes  (id,  numAttrs)
void*    id;                                                    /* in  */
TYPE     numAttrs;                                              /* out  */

int CDFgetNumAttrrEntries  (id,  attrNum, entries)
void*    id;                                                    /* in  */
int      attrNum;                                                       /* in  */
TYPE     entries;                                               /* out  */

int CDFgetNumAttrzEntries  (id,  attrNum, entries)
void*    id;                                                    /* in  */
```

```
int       attrNum;                                              /* in  */
TYPE      entries;                                              /* out */

int CDFgetNumgAttributes  (id,  numAttrs)
void*     id;                                                   /* in  */
TYPE      numAttrs;                                             /* out */

int CDFgetNumrVars  (id,  numVars)
void*     id;                                                   /* in  */
TYPE      numrVars;                                             /* out */

int CDFgetNumvAttributes  (id,  numAttrs)
void*     id;                                                   /* in  */
TYPE      numAttrs;                                             /* out */

int CDFgetNumzVars  (id,  numVars)
void*     id;                                                   /* in  */
TYPE      numzVars;                                             /* out */

int CDFgetReadOnlyMode  (id,  mode)
void*     id;                                                   /* in  */
TYPE      mode;                                                 /* out */

int CDFgetrVarAllocRecords  (id,  varNum, allocRecs)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
TYPE      allocRecs;                                            /* out */

int CDFgetrVarBlockingFactor  (id,  varNum, bf)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
TYPE      bf;                                                   /* out */

int CDFgetrVarCacheSize  (id,  varNum, numBuffers)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
TYPE      numBuffers;                                           /* out */

int CDFgetrVarCompression  (id,  varNum, cType, cParms, cPercent)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
TYPE      cType;                                                /* out */
TYPE2     cParms;                                               /* out */
TYPE      cPercent;                                             /* out */

int CDFgetrVarData  (id,  varNum, recNum, indices, value)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
int       recNum;                                               /* in  */
int[]     indices;                                              /* in  */
TYPE4     value;                                                /* out */

int CDFgetrVarDataType  (id,  varNum, dataType)
void*     id;                                                   /* in  */
int       varNum;                                               /* in  */
TYPE      dataType;                                             /* out */
```

```
int CDFgetrVarsDimSizes  (id,  varNum, dimSizes)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE2     dimSizes;                                              /* out  */

int CDFgetrVarDimVariances  (id,  varNum, dimVarys)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE2     dimVarys;                                              /* out  */

int CDFgetrVarInfo  (id,  varNum, dataType, numElems, numDims, dimSizes)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      dataType;                                              /* out  */
TYPE      numElems;                                              /* out  */
TYPE      numDims;                                               /* out  */
TYPE2     dimSizes;                                              /* out  */

int CDFgetrVarMaxAllocRecNum  (id,  varNum, maxRec)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      maxRec;                                                /* out  */

int CDFgetrVarMaxWrittenRecNum  (id,  varNum, maxRec)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      maxRec;                                                /* out  */

int CDFgetrVarName  (id,  varNum, varName)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
out string          varName;                                     /* out  */

int CDFgetrVarsNumDims  (id,  varNum, numDims)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      numDims;                                               /* out  */

int CDFgetrVarNumElements  (id,  varNum, numElems)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      numElems;                                              /* out  */

int CDFgetrVarNumRecsWritten  (id,  varNum, numRecs)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      numRecs;                                               /* out  */

int CDFgetrVarPadValue  (id,  varNum, padValue)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE4     padValue;                                              /* out  */

int CDFgetrVarRecordData  (id,  varNum, recNum,  buffer)
void*     id;                                                    /* in  */
```

```
int         varNum;                                    /* in  */
int         recNum;                                    /* in  */
TYPE6       buffer;                                    /* out */

int CDFgetrVarRecVariance  (id,  varNum, recVary)
void*       id;                                        /* in  */
int         varNum;                                    /* in  */
TYPE        recVary;                                   /* out  */

int CDFgetrVarReservePercent  (id,  varNum, percent)
void*       id;                                        /* in  */
int         varNum;                                    /* in  */
TYPE        percent;                                   /* out  */

int CDFgetrVarsDimSizes  (id,  dimSizes)
void*       id;                                        /* in  */
TYPE2       dimSizes;                                  /* out  */

int CDFgetrVarSeqData  (id,  varNum, value)
void*       id;                                        /* in  */
int         varNum;                                    /* in  */
TYPE4       value;                                     /* out  */

int CDFgetrVarSeqPos  (id,  varNum, recNum, indices)
void*       id;                                        /* in  */
int         varNum;                                    /* in  */
TYPE        recNum;                                    /* out  */
TYPE2       indices;                                   /* out  */

int CDFgetrVarsMaxWrittenRecNum  (id,  recNum)
void*       id;                                        /* in  */
TYPE        recNum;                                    /* out  */

int CDFgetrVarsNumDims  (id,  numDims)
void*       id;                                        /* in  */
TYPE        numDims;                                   /* out  */

int CDFgetrVarSparseRecords  (id,  varNum, sRecords)
void*       id;                                        /* in  */
int         varNum;                                    /* in  */
TYPE        sRecords;                                  /* out  */

int CDFgetStageCacheSize  (id,  numBuffers)
void*       id;                                        /* in  */
TYPE        numBuffers;                                /* out  */

int CDFgetStatusText  (status,  text)
int status;                                            /* in  */
out string          text;                              /* out  */

int CDFgetValidate  ()

int CDFgetVarNum  (id,  varName)
void*       id;                                        /* in  */
string      varName;                                   /* in  */
```

```
int CDFgetVersion  (id,  version, release, increment)
void*    id;                                          /* in  */
TYPE     version;                                     /* out  */
TYPE     release;                                     /* out  */
TYPE     increment;                                   /* out  */

int CDFgetzMode  (id,  zMode)
void*    id;                                          /* in  */
TYPE     zMode;                                       /* out  */

int CDFgetzVarAllocRecords  (id,  varNum, allocRecs)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE     allocRecs;                                   /* out  */

int CDFgetzVarBlockingFactor  (id,  varNum, bf)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE     bf;                                          /* out  */

int CDFgetzVarCacheSize  (id,  varNum, numBuffers)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE     numBuffers;                                  /* out  */

int CDFgetzVarCompression  (id,  varNum, cType, cParms, cPercent)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE     cType;                                       /* out  */
TYPE2    cParms;                                      /* out  */
TYPE     cPercent;                                    /* out  */

int CDFgetzVarData  (id,  varNum, recNum, indices, value)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
int      recNum;                                      /* in  */
int[]    indices;                                     /* in  */
TYPE4    value;                                       /* out  */

int CDFgetzVarDataType  (id,  varNum, dataType)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE     dataType;                                    /* out  */

int CDFgetzVarDimSizes  (id,  varNum, dimSizes)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE2    dimSizes;                                    /* out  */

int CDFgetzVarDimVariances  (id,  varNum, dimVarys)
void*    id;                                          /* in  */
int      varNum;                                      /* in  */
TYPE2    dimVarys;                                    /* out  */

int CDFgetzVarInfo  (id,  varNum, dataType, numElems, numDims, dimSizes)
void*    id;                                          /* in  */
```

```
int         varNum;                                   /* in  */
TYPE        dataType;                                 /* out  */
TYPE        numElems;                                 /* out  */
TYPE        numDims;                                  /* out  */
TYPE2       dimSizes;                                 /* out  */

int CDFgetzVarMaxAllocRecNum  (id,  varNum, maxRec)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        maxRec;                                   /* out  */

int CDFgetzVarMaxWrittenRecNum  (id,  varNum, maxRec)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        maxRec;                                   /* out  */

int CDFgetzVarName  (id,  varNum, varName)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
out string          varName;                          /* out  */

int CDFgetzVarNumDims  (id,  varNum, numDims)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        numDims;                                  /* out  */

int CDFgetzVarNumElements  (id,  varNum, numElems)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        numElems;                                 /* out  */

int CDFgetzVarNumRecsWritten  (id,  varNum, numRecs)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        numRecs;                                  /* out  */

int CDFgetzVarPadValue  (id,  varNum, padValue)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE4       padValue;                                 /* out  */

int CDFgetzVarRecordData  (id,  varNum, recNum,  buffer)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
int         recNum;                                   /* in  */
TYPE6       buffer;                                   /* out */

int CDFgetzVarRecVariance  (id,  varNum, recVary)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        recVary;                                  /* out  */

int CDFgetzVarReservePercent  (id,  varNum, percent)
void*       id;                                       /* in  */
int         varNum;                                   /* in  */
TYPE        percent;                                  /* out  */
```

```
int CDFgetzVarSeqData  (id,  varNum, value)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE4     value;                                                 /* out  */

int CDFgetzVarSeqPos  (id,  varNum, recNum, indices)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      recNum;                                                /* out  */
TYPE2     indices;                                               /* out  */

int CDFgetzVarsMaxWrittenRecNum  (id,  recNum)
void*     id;                                                    /* in  */
TYPE      recNum;                                                /* out  */

int CDFgetzVarSparseRecords  (id,  varNum, sRecords)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
TYPE      sRecords;                                              /* out  */

int CDFhyperGetrVarData  (id,  varNum,  recNum, recCount, recInterval, indices, counts, intervals, buffer)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       recNum;                                                /* in  */
int       recCount;                                              /* in  */
int       recInterval;                                           /* in  */
int[]     indices;                                               /* in  */
int[]     counts;                                                /* in  */
int[]     intervals;                                             /* in  */
TYPE6     buffer;                                                /* out */

int CDFhyperGetzVarData  (id,  varNum,  recNum, recCount, recInterval, indices, counts, intervals, buffer)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       recNum;                                                /* in  */
int       recCount;                                              /* in  */
int       recInterval;                                           /* in  */
int[]     indices;                                               /* in  */
int[]     counts;                                                /* in  */
int[]     intervals;                                             /* in  */
TYPE6     buffer;                                                /* out */

int CDFhyperPutrVarData  (id,  varNum,  recNum, recCount, recInterval, indices, counts, intervals, buffer)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       recNum;                                                /* in  */
int       recCount;                                              /* in  */
int       recInterval;                                           /* in  */
int[]     indices;                                               /* in  */
int[]     counts;                                                /* in  */
int[]     intervals;                                             /* in  */
TYPE5     buffer;                                                /* in */

int CDFhyperPutzVarData  (id,  varNum,  recNum, recCount, recInterval, indices, counts, intervals, buffer)
void*     id;                                                    /* in  */
```

```
int         varNum;                                                    /* in  */
int         recNum;                                                    /* in  */
int         recCount;                                                  /* in  */
int         recInterval;                                               /* in  */
int[]       indices;                                                   /* in  */
int[]       counts;                                                    /* in  */
int[]       intervals;                                                 /* in  */
TYPE5       buffer;                                                    /* in */

int CDFinquire  (id,  numDims,  dimSizes, encoding,  majority,  maxRec, numVars,  numAttrs)
void*       id;                                                        /* in */
TYPE        numDims;                                                   /* out */
TYPE2       dimSizes;                                                  /* out */
TYPE        encoding;                                                  /* out */
TYPE        majority;                                                  /* out */
TYPE        maxRec;                                                    /* out */
TYPE        numVars;                                                   /* out */
TYPE        numAttrs;                                                  /* out */

int CDFinquireAttr  (id, attrNum, attrName, attrScope,  maxgEntry, maxrEntry, maxzEntry)
void*       id;                                                        /* in */
int         attrNum;                                                   /* in */
out string          attrName;                                         /* out */
TYPE        attrScope;                                                 /* out */
TYPE        maxgEntry;                                                 /* out */
TYPE        maxrEntry;                                                 /* out */
TYPE        maxzEntry;                                                 /* out */

int CDFinquireAttrgEntry  (id,  attrNum, entryNum, dataType, numElems)
void*       id;                                                        /* in */
int         attrNum;                                                   /* in */
int         entryNum;                                                  /* in */
TYPE        dataType;                                                  /* out */
TYPE        numElems;                                                  /* out */

int CDFinquireAttrrEntry  (id,  attrNum, entryNum, dataType, numElems)
void*       id;                                                        /* in */
int         attrNum;                                                   /* in */
int         entryNum;                                                  /* in */
TYPE        dataType;                                                  /* out */
TYPE        numElems;                                                  /* out */

int CDFinquireAttrzEntry  (id,  attrNum, entryNum, dataType, numElems)
void*       id;                                                        /* in */
int         attrNum;                                                   /* in */
int         entryNum;                                                  /* in */
TYPE        dataType;                                                  /* out */
TYPE        numElems;                                                  /* out */

int CDFinquireCDF  (id,  numDims,  dimSizes, encoding,  majority, maxrRec, numrVars, maxzRec,
                    numzVars, numAttrs)
void*       id;                                                        /* in */
TYPE        numDims;                                                   /* out */
TYPE2       dimSizes;                                                  /* out */
TYPE        encoding;                                                  /* out */
TYPE        majority;                                                  /* out */
```

```
TYPE  maxrRec;                                                    /* out */
TYPE  numrVars;                                                   /* out */
TYPE  maxzRec;                                                    /* out */
TYPE  numzVars;                                                   /* out */
TYPE  numAttrs;                                                   /* out */

int CDFinquirerVar  (id, varNum, varName, dataType, numElems, numDims, dimSizes, recVary, dimVarys)
void*     id;                                                     /* in */
int       varNum;                                                 /* in */
out string          varName;                                     /* out */
TYPE  dataType;                                                   /* out */
TYPE  numElems;                                                   /* out */
TYPE  numDims;                                                    /* out */
TYPE2 dimSizes;                                                   /* out */
TYPE  recVary;                                                    /* out */
TYPE2 dimVarys;                                                   /* out */

int CDFinquirezVar  (id, varNum, varName, dataType, numElems, numDims, dimSizes, recVary, dimVarys)
void*     id;                                                     /* in */
int       varNum;                                                 /* in */
out string          varName;                                     /* out */
TYPE  dataType;                                                   /* out */
TYPE  numElems;                                                   /* out */
TYPE  numDims;                                                    /* out */
TYPE2 dimSizes;                                                   /* out */
TYPE  recVary;                                                    /* out */
TYPE2 dimVarys;                                                   /* out */

int CDFopen  (CDFname,  id)
string    CDFname;                                                /* in */
void*     *id;                                                    /* out */

int CDFopenCDF  (CDFname,  id)
string    CDFname;                                                /* in */
void*     *id;                                                    /* out */

int CDFselectCDF  (id)
void*     id;                                                     /* in */

int CDFputAttrgEntry  (id,  attrNum,  entryNum,  dataType, numElems,  value)
void*     id;                                                     /* in  */
int       attrNum;                                                /* in  */
int       entryNum;                                               /* in  */
int       dataType;                                               /* in  */
int       numElems;                                               /* in  */
TYPE3 value;                                                      /* in  */

int CDFputAttrrEntry  (id,  attrNum,  entryNum,  dataType, numElems, value)
void*     id;                                                     /* in  */
int       attrNum;                                                /* in  */
int       entryNum;                                               /* in  */
int       dataType;                                               /* in  */
int       numElems;                                               /* in  */
TYPE3 value;                                                      /* in  */

int CDFputAttrzEntry  (id,  attrNum,  entryNum,  dataType, numElems, value)
```

```
void*     id;                                          /* in  */
int       attrNum;                                     /* in  */
int       entryNum;                                    /* in  */
int       dataType;                                    /* in  */
int       numElems;                                    /* in  */
TYPE3     value;                                       /* in  */

int CDFputrVarData  (id,  varNum,  recNum,  indices, value)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
int       recNum;                                      /* in  */
int[]     indices;                                     /* in  */
TYPE3     value;                                       /* in  */

int CDFputrVarPadValue  (id, varNum, padValue)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
TYPE3     padValue;                                    /* in  */

int CDFputrVarRecordData  (id,  varNum,  recNum, values)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
int       recNum;                                      /* in  */
TYPE5     values;                                      /* in  */

int CDFputrVarSeqData  (id,  varNum, value)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
TYPE3     value;                                       /* in  */

int CDFputzVarData  (id,  varNum,  recNum,  indices, value)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
int       recNum;                                      /* in  */
int[]     indices;                                     /* in  */
TYPE3     value;                                       /* in  */

int CDFputzVarPadValue  (id, varNum, padValue)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
TYPE3     padValue;                                    /* in  */

int CDFputzVarRecordData  (id,  varNum,  recNum, values)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
int       recNum;                                      /* in  */
TYPE5     values;                                      /* in  */

int CDFputzVarSeqData  (id,  varNum, value)
void*     id;                                          /* in  */
int       varNum;                                      /* in  */
TYPE3     value;                                       /* in  */

int CDFrenameAttr  (id,  attrNum, attrName)
void*     id;                                          /* in  */
int       attrNum;                                     /* in  */
```

```
string    attrName;                                          /* in  */

int CDFrenamerVar  (id,  varNum, varName)
void*     id;                                                /* in  */
int       varNum;                                            /* in  */
string    varName;                                           /* in  */

int CDFrenamezVar  (id,  varNum, varName)
void*     id;                                                /* in  */
int       varNum;                                            /* in  */
string    varName;                                           /* in  */

int CDFselect  (id)
void*     id;                                                /* in  */

int CDFselectCDF  (id)
void*     id;                                                /* in  */

int CDFsetAttrgEntryDataSpec  (id, attrNum, entryNum, dataType)
void*     id;                                                /* in  */
int       attrNum;                                           /* in  */
int       entryNum;                                          /* in  */
int       dataType;                                          /* in  */

int CDFsetAttrrEntryDataSpec  (id, attrNum, entryNum, dataType)
void*     id;                                                /* in  */
int       attrNum;                                           /* in  */
int       entryNum;                                          /* in  */
int       dataType;                                          /* in  */

int CDFsetAttrScope  (id, attrNum, scope)
void*     id;                                                /* in  */
int       attrNum;                                           /* in  */
int       scope;                                             /* in  */

int CDFsetAttrzEntryDataSpec  (id, attrNum, entryNum, dataType)
void*     id;                                                /* in  */
int       attrNum;                                           /* in  */
int       entryNum;                                          /* in  */
int       dataType;                                          /* in  */

int CDFsetCacheSize  (id, numBuffers)
void*     id;                                                /* in  */
int       numBuffers;                                        /* in  */

int CDFsetChecksum  (id, checksum)
void*     id;                                                /* in  */
int       checksum;                                          /* in  */

int CDFsetCompression  (id, compressionType, compressionParms)
void*     id;                                                /* in  */
int       compressionType;                                   /* in  */
int[]     compressionParms;                                  /* in  */

int CDFsetCompressionCacheSize  (id, numBuffers)
void*     id;                                                /* in  */
```

```
int         numBuffers;                                                 /* in  */

int CDFsetDecoding  (id, decoding)
void*       id;                                                         /* in  */
int         decoding;                                                   /* in  */

int CDFsetEncoding  (id, encoding)
void*       id;                                                         /* in  */
int         encoding;                                                   /* in  */

void CDFsetFileBackward  (mode)
int         mode;                                                       /* in  */

int CDFsetFormat  (id, format)
void*       id;                                                         /* in  */
int         format;                                                     /* in  */

int CDFsetLeapSecondLastUpdated  (id, lastUpdated)
void*       id;                                                         /* in  */
int         lastUpdated;                                                /* in  */

int CDFsetMajority  (id, majority)
void*       id;                                                         /* in  */
int         majority;                                                   /* in  */

int CDFsetNegtoPosfp0Mode  (id, negtoPosfp0)
void*       id;                                                         /* in  */
int         negtoPosfp0;                                                /* in  */

int CDFsetReadOnlyMode (id, readOnly)
void*       id;                                                         /* in  */
int         readOnly;                                                   /* in  */

int CDFsetrVarAllocBlockRecords  (id, varNum, firstRec, lastRec)
void*       id;                                                         /* in  */
int         varNum;                                                     /* in  */
int         firstRec;                                                   /* in  */
int         lastRec;                                                    /* in  */

int CDFsetrVarAllocRecords  (id, varNum, numRecs)
void*       id;                                                         /* in  */
int         varNum;                                                     /* in  */
int         numRecs;                                                    /* in  */

int CDFsetrVarBlockingFactor  (id, varNum, bf)
void*       id;                                                         /* in  */
int         varNum;                                                     /* in  */
int         bf;                                                         /* in  */

int CDFsetrVarCacheSize  (id, varNum, numBuffers)
void*       id;                                                         /* in  */
int         varNum;                                                     /* in  */
int         numBuffers;                                                 /* in  */

int CDFsetrVarCompression  (id, varNum, compressionType,  compressionParms)
void*       id;                                                         /* in  */
```

```
int       varNum;                                          /* in  */
int       compressionType;                                 /* in  */
int[]     compressionParms;                                /* in  */

int CDFsetrVarDataSpec (id, varNum, dataType)
void*     id;                                              /* in  */
int       varNum;                                          /* in  */
int       dataType;                                        /* in  */

int CDFsetrVarDimVariances (id, varNum, dimVarys)
void*     id;                                              /* in  */
int       varNum;                                          /* in  */
int[]     dimVarys;                                        /* in  */

int CDFsetrVarInitialRecs  (id, varNum, initialRecs)
void*     id;                                              /* in  */
int       varNum;                                          /* in  */
int       initialRecs;                                     /* in  */

int CDFsetrVarRecVariance  (id, varNum, recVary)
void*     id;                                              /* in  */
int       varNum;                                          /* in  */
int       recVary;                                         /* in  */

int CDFsetrVarReservePercent  (id, varNum, reservePercent)
void*     id;                                              /* in  */
int       varNum;                                          /* in  */
int       reservePercent;                                  /* in  */

int CDFsetrVarsCacheSize  (id, numBuffers)
void*     id;                                              /* in  */
int       numBuffers;                                      /* in  */

int CDFsetrVarSeqPos (id, varNum, recNum, indices)
void*     id;                                              /* in  */
int       varNum;                                          /* in */
int       recNum;                                          /* in */
int[]     indices;                                         /* in  */

int CDFsetrVarSparseRecords (id, varNum, sRecords)
void*     id;                                              /* in  */
int       varNum;                                          /* in */
int       sRecords;                                        /* in */

int CDFsetStageCacheSize (id, numBuffers)
void*     id;                                              /* in  */
int       numBuffers;                                      /* in  */

void CDFsetValidate  (mode)
int       mode;                                            /* in  */

int CDFsetzMode (id, zMode)
void*     id;                                              /* in  */
int       zMode;                                           /* in  */

int CDFsetzVarAllocBlockRecords  (id, varNum, firstRec, lastRec)
```

```
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       firstRec;                                              /* in  */
int       lastRec;                                               /* in  */

int CDFsetzVarAllocRecords  (id, varNum, numRecs)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       numRecs;                                               /* in  */

int CDFsetzVarBlockingFactor  (id, varNum, bf)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       bf;                                                    /* in  */

int CDFsetzVarCacheSize  (id, varNum, numBuffers)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       numBuffers;                                            /* in  */

int CDFsetzVarCompression  (id, varNum, compressionType,  compressionParms)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       compressionType;                                       /* in  */
int[]     compressionParms;                                      /* in  */

int CDFsetzVarDataSpec (id, varNum, dataType)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       dataType;                                              /* in  */

int CDFsetzVarDimVariances (id, varNum, dimVarys)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int[]     dimVarys;                                              /* in  */

int CDFsetzVarInitialRecs  (id, varNum, initialRecs)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       initialRecs;                                           /* in  */

int CDFsetzVarRecVariance  (id, varNum, recVary)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       recVary;                                               /* in  */

int CDFsetzVarReservePercent  (id, varNum, reservePercent)
void*     id;                                                    /* in  */
int       varNum;                                                /* in  */
int       reservePercent;                                        /* in  */

int CDFsetzVarsCacheSize  (id, numBuffers)
void*     id;                                                    /* in  */
int       numBuffers;                                            /* in  */

int CDFsetzVarSeqPos (id, varNum, recNum, indices)
```

```
void*     id;                                                    /* in  */
int       varNum;                                                /* in */
int       recNum;                                                /* in */
int[]     indices;                                               /* in  */

int CDFsetzVarSparseRecords (id, varNum, sRecords)
void*     id;                                                    /* in  */
int       varNum;                                                /* in */
int       sRecords;                                              /* in */

int CDFvarClose  (id,  varNum)
void*     id;                                                    /* in */
int       varNum;                                                /* in */

int CDFvarCreate  (id,  varName, dataType,  numElements, recVariance,  dimVariances,  varNum)
void*     id;                                                    /* in */
string    varName;                                               /* in */
int       dataType;                                              /* in */
int       numElements;                                           /* in */
int       recVariance;                                           /* in */
int[]     dimVariances;                                          /* in */
TYPE      varNum;                                                /* out */

int CDFvarGet  (id,  varNum,  recNum, indices,  value)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       recNum;                                                /* in */
int[]     indices;                                               /* in */
TYPE4     value;                                                 /*  out */

int CDFvarHyperGet  (id,  varNum,  recStart,  recCount,  recInterval,  indices,  counts, intervals,  buffer)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       recStart;                                              /* in */
int       recCount;                                              /* in */
int       recInterval;                                           /* in */
int[]     indices;                                               /* in */
int[]     counts;                                                /* in */
int[]     intervals;                                             /* in */
TYPE6     buffer;                                                /* out */

int CDFvarHyperPut  (id,  varNum,  recStart,  recCount,  recInterval,  indices,  counts, intervals,  buffer)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
int       recStart;                                              /* in */
int       recCount;                                              /* in */
int       recInterval;                                           /* in */
int[]     indices;                                               /* in */
int[]     counts;                                                /* in */
int[]     intervals;                                             /* in */
TYPE3     buffer;                                                /* in */

int CDFvarInquire  (id,  varNum,  varName, dataType,  numElements,  recVariance,  dimVariances)
void*     id;                                                    /* in */
int       varNum;                                                /* in */
out string          varName;                                     /* out */
```

```
TYPE  dataType;                                   /* out */
TYPE  numElements;                               /* out */
TYPE  recVariance;                               /* out */
TYPE2 dimVariances;                              /* out */

int  CDFvarNum  (id,  varName)
void*     id;                                    /* in */
string    varName;                               /* in */

int CDFvarPut  (id,  varNum,  recNum, indices,  value)
void*     id;                                    /* in */
int       varNum;                                /* in */
int       recNum;                                /* in */
int[]     indices;                               /* in */
TYPE3 value;                                     /* in */

int CDFvarRename (id,  varNum,  varName)
void*     id;                                    /* in */
int       varNum;                                /* in */
string    varName;                               /* in */
```

B.2 EPOCH Utility Methods

The overloaded APIs will have the following **TYPE** symbols, which represent a set of possible types for parameters.

- **TYPE** -- **int*** or "**out int**"
- **TYPE2** -- **double*** or "**out double[]**"
- **TYPE3** -- **double** or **long**
- **TYPE4** -- **double*** or **long***

```
double computeEPOCH (year, month, day, hour, minute, second, msec)
int year;                                                    /* in */
int month;                                                   /* in */
int day;                                                     /* in */
int hour;                                                    /* in */
int minute;                                                  /* in */
int second;                                                  /* in */
int msec;                                                    /* in */

double[] computeEPOCH (year, month, day, hour, minute, second, msec)
int[] year;                                                  /* in */
int[] month;                                                 /* in */
int[] day;                                                   /* in */
int[] hour;                                                  /* in */
int[] minute;                                                /* in */
int[] second;                                                /* in */
int[] msec;                                                  /* in */

void EPOCHbreakdown (epoch, year, month, day, hour, minute, second, msec)
double epoch;                                                /* in */
TYPE year;                                                   /* out */
TYPE month;                                                  /* out */
TYPE day;                                                    /* out */
TYPE hour;                                                   /* out */
TYPE minute;                                                 /* out */
TYPE second;                                                 /* out */
TYPE msec;                                                   /* out */

void EPOCHbreakdown (epoch, year, month, day, hour, minute, second, msec)
double[] epoch;                                              /* in */
out int[] year;                                              /* out */
out int[] month;                                             /* out */
out int[] day;                                               /* out */
out int[] hour;                                              /* out */
out int[] minute;                                            /* out */
out int[] second;                                            /* out */
out int[] msec;                                              /* out */

void encodeEPOCH (epoch, epString)
double epoch;                                                /* in */
out string epString;                                         /* out */

void encodeEPOCH (epoch, epString)
double[] epoch;                                              /* in */
```

```
out string[] epString;                                                    /* out */

void encodeEPOCH1 (epoch, epString)
double epoch;                                                             /* in */
out string epString;                                                      /* out */

void encodeEPOCH2 (epoch, epString)
double epoch;                                                             /* in */
out string epString;                                                      /* out */

void encodeEPOCH3 (epoch, epString)
double epoch;                                                             /* in */
out string epString                                                       /* out */

void encodeEPOCH4 (epoch, epString)
double epoch;                                                             /* in */
out string epString                                                       /* out */

void encodeEPOCHx (epoch, format, epString)
double epoch;                                                             /* in */
string format;                                                            /* in */
out string epString;                                                      /* out */

double parseEPOCH (epString)
string epString;                                                          /* in */

double[] parseEPOCH (epString)
string[] epString;                                                        /* in */

double parseEPOCH1 (epString)
string epString;                                                          /* in */

double parseEPOCH2 (epString)
string epString;                                                          /* in */

double parseEPOCH3 (epString)
string epString;                                                          /* in */

double parseEPOCH4 (epString)
string epString;                                                          /* in */

double computeEPOCH16 (year, month, day, hour, minute, second, msec, microsec, nanosec, picosec)
int year;                                                                 /* in */
int month;                                                                /* in */
int day;                                                                  /* in */
int hour;                                                                 /* in */
int minute;                                                               /* in */
int second;                                                               /* in */
int msec;                                                                 /* in */
int microsec;                                                             /* in */
int nanosec;                                                              /* in */
int picosec;                                                              /* in */
TYPE2 epoch;                                                              /* out */

void EPOCH16breakdown (epoch, year, month, day, hour, minute, second, msec, microsec, nanosec, picosec)
double[] epoch;                                                           /* in */
```

```
TYPE  year;                                                    /*  out */
TYPE  month;                                                   /*  out */
TYPE  day;                                                     /*  out */
TYPE  hour;                                                    /*  out */
TYPE  minute;                                                  /*  out */
TYPE  second;                                                  /*  out */
TYPE  msec;                                                    /*  out */
TYPE  microsec;                                                /*  out */
TYPE  nanosec;                                                 /*  out */
TYPE  picosec;                                                 /*  out */

void encodeEPOCH16 (epoch, epString)
double[] epoch;                                                /*  in */
out string epString;                                           /*  out */

void encodeEPOCH16_1 (epoch, epString)
double[] epoch;                                                /*  in */
out string epString;                                           /*  out */

void encodeEPOCH16_2 (epoch, epString)
double[] epoch;                                                /*  in */
out string epString;                                           /*  out */

void encodeEPOCH16_3 (epoch, epString)
double[] epoch;                                                /*  in */
out string epString;                                           /*  out */

void encodeEPOCH16_4 (epoch, epString)
double[] epoch;                                                /*  in */
out string epString;                                           /*  out */

void encodeEPOCH16_x (epoch, format, epString)
double[] epoch;                                                /*  in */
string format;                                                 /*  in */
out string epString;                                           /*  out */

double parseEPOCH16 (epString, epoch)
string epString;                                               /* in */
TYPE2  epoch;                                                  /*  out */

double parseEPOCH16_1 (epString)
string epString;                                               /* in */
TYPE2  epoch;                                                  /*  out */


double parseEPOCH16_2 (epString)
string epString;                                               /* in */
TYPE2  epoch;                                                  /*  out */

double parseEPOCH16_3 (epString)
string epString;                                               /* in */
TYPE2  epoch;                                                  /*  out */

double parseEPOCH16_4 (epString)
string epString;                                               /* in */
TYPE2  epoch;                                                  /*  out */
```

```
long computeTT2000 (year, month, day)
TYPE3 year;                                                    /* in */
TYPE3 month;                                                   /* in */
TYPE3 day;                                                     /* in */

long[] computeTT2000 (year, month, day)
double[] year;                                                 /* in */
double[] month;                                                /* in */
double[] day;                                                  /* in */

long computeTT2000 (year, month, day, hour)
TYPE3 year;                                                    /* in */
TYPE3 month;                                                   /* in */
TYPE3 day;                                                     /* in */
TYPE3 hour;                                                    /* in */

long[] computeTT2000 (year, month, day, hour)
double[] year;                                                 /* in */
double[] month;                                                /* in */
double[] day;                                                  /* in */
double[] hour;                                                 /* in */

long computeTT2000 (year, month, day, hour, minute)
TYPE3 year;                                                    /* in */
TYPE3 month;                                                   /* in */
TYPE3 day;                                                     /* in */
TYPE3 hour;                                                    /* in */
TYPE3 minute;                                                  /* in */

long[] computeTT2000 (year, month, day, hour, minute)
double[] year;                                                 /* in */
double[] month;                                                /* in */
double[] day;                                                  /* in */
double[] hour;                                                 /* in */
double[] minute;                                               /* in */

long computeTT2000 (year, month, day, hour, minute, second)
TYPE3 year;                                                    /* in */
TYPE3 month;                                                   /* in */
TYPE3 day;                                                     /* in */
TYPE3 hour;                                                    /* in */
TYPE3 minute;                                                  /* in */
TYPE3 second;                                                  /* in */

long[] computeTT2000 (year, month, day, hour, minute, second)
double[] year;                                                 /* in */
double[] month;                                                /* in */
double[] day;                                                  /* in */
double[] hour;                                                 /* in */
double[] minute;                                               /* in */
double[] second;                                               /* in */

long computeTT2000 (year, month, day, hour, minute, second, msec)
TYPE3 year;                                                    /* in */
TYPE3 month;                                                   /* in */
```

TYPE3 day; /* in */
TYPE3 hour; /* in */
TYPE3 minute; /* in */
TYPE3 second; /* in */
TYPE3 msec; /* in */

long[] computeTT2000 (year, month, day, hour, minute, second, msec)
double[] year; /* in */
double[] month; /* in */
double[] day; /* in */
double[] hour; /* in */
double[] minute; /* in */
double[] second; /* in */
double[] msec; /* in */

long computeTT2000 (year, month, day, hour, minute, second, msec, usec)
TYPE3 year; /* in */
TYPE3 month; /* in */
TYPE3 day; /* in */
TYPE3 hour; /* in */
TYPE3 minute; /* in */
TYPE3 second; /* in */
TYPE3 msec; /* in */
TYPE3 usec; /* in */

long[] computeTT2000 (year, month, day, hour, minute, second, msec, usec)
double[] year; /* in */
double[] month; /* in */
double[] day; /* in */
double[] hour; /* in */
double[] minute; /* in */
double[] second; /* in */
double[] msec; /* in */
double[] usec; /* in */

long computeTT2000 (year, month, day, hour, minute, second, msec, usec, nsec)
TYPE3 year; /* in */
TYPE3 month; /* in */
TYPE3 day; /* in */
TYPE3 hour; /* in */
TYPE3 minute; /* in */
TYPE3 second; /* in */
TYPE3 msec; /* in */
TYPE3 usec; /* in */
TYPE3 nsec; /* in */

long[] computeTT2000 (year, month, day, hour, minute, second, msec, usec, nsec)
double[] year; /* in */
double[] month; /* in */
double[] day; /* in */
double[] hour; /* in */
double[] minute; /* in */
double[] second; /* in */
double[] msec; /* in */
double[] usec; /* in */
double[] nsec; /* in */

```
void TT2000breakdown (epoch, year, month, day, hour, minute, second, msec, usec, nsec)
long epoch;                                                                  /* in */
out TYPE4  year;                                                             /* out */
out TYPE4  month;                                                            /* out */
out TYPE4  day;                                                              /* out */
out TYPE4  hour;                                                             /* out */
out TYPE4  minute;                                                           /* out */
out TYPE4  second;                                                           /* out */
out TYPE4  msec;                                                             /* out */
out TYPE4  usec;                                                             /* out */
out TYPE4  nsec;                                                             /* out */

void TT2000breakdown (epoch, year, month, day, hour, minute, second, msec, usec, nsec)
long[] epoch;                                                                /* in */
out double[]  year;                                                          /* out */
out double[]  month;                                                         /* out */
out double[]  day;                                                           /* out */
out double[]  hour;                                                          /* out */
out double[]  minute;                                                        /* out */
out double[]  second;                                                        /* out */
out double[]  msec;                                                          /* out */
out double[]  usec;                                                          /* out */
out double[]  nsec;                                                          /* out */

void encodeTT2000 (epoch, epString, format)
long   epoch;                                                                /* in */
out string epString;                                                         /* out */
int    format;                                                               /* in  */

void encodeTT2000 (epoch, epString, format)
long[]   epoch;                                                              /* in */
out string[] epString;                                                       /* out */
int    format;                                                               /* in  */

long   parseTT2000 (epString)
string epString;                                                             /* in */

long[]   parseTT2000 (epString)
string[] epString;                                                           /* in */

void    CDFgetLastDateinLeapSecondsTable (year, month, day)
out int year;                                                                /* out */
out int month;                                                               /* out */
out int day;                                                                 /* out */
```

B.3 CDF Utility Methods

```
bool CDFFileExists (fileName)
string fileName;                                                    /* in */

int CDFgetChecksumValue(checksum)
string fileName;                                                    /* in */

int CDFgetCompressionTypeValue(compressionType)
string compressionType;                                             /* in */

int CDFgetDataTypeValue(dataType)
string dataType;                                                    /* in */

int CDFgetDecodingValue(decoding)
string decoding;                                                    /* in */

int CDFgetEncodingValue(encoding)
string encoding;                                                    /* in */

int CDFgetFormatValue(format)
string format;                                                      /* in */

int CDFgetMajorityValue(majority)
string majority;                                                    /* in */

int CDFgetSparseRecordValue(sparseRecord)
string sparseRecord;                                                /* in */

string CDFgetStringChecksum(checksum)
int checksum;                                                       /* in */

string CDFgetStringCompressionType(compressionType)
int compressionType;                                                /* in */

string CDFgetStringDataType(dataType)
int dataType;                                                       /* in */

string CDFgetStringDecoding(decoding)
int decoding;                                                       /* in */

string CDFgetStringEncoding(encoding)
int encoding;                                                       /* in */

string CDFgetStringFormat(format)
int format;                                                         /* in */

string CDFgetStringMajority(majority)
int majority;                                                       /* in */

string CDFgetStringSparseRecord(sparseRecord)
int sparseRecord;                                                   /* in */
```

B.4 CDF Exception Methods

```
int  CDFgetCurrentStatus  ()

string  CDFgetStatusMsg(status)
int  status;
```

Index