CDF
C Reference Manual

Version 2.6, October 1, 1998

National Space Science Data Center

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

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

Contents

Chapter 1

Compiling

Each source file that calls the CDF library or references CDF parameters must include `cdf.h`. On VMS systems a logical name, `CDF$INC`, that specifies the location of `cdf.h` is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_INC`, that serves the same purpose is defined in the definitions file `definitions.<shell-type>` where `<shell-type>` is the type of shell being used: `C` for the C-shell (`csh` and `tcsh`), `K` for the Korn (`ksh`), BASH, and POSIX shells, and `B` for the Bourne shell (`sh`). This section assumes that you are using the appropriate definitions file on those systems. On MS-DOS and Macintosh (MacOS) systems, definitions files are not available. The location of `cdf.h` is specified as described in the appropriate sections for those systems.

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

1.1 Specifying `cdf.h` Location in the Compile Command

The first method involves including the following line at/near the top of each source file:

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

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

1.1.1 VMS/OpenVMS Systems

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

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

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

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

1.1.2 UNIX Systems

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

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

where <source-name>.c is the name of the source file being compiled (the .c extension is required). The -c option specifies that only an object module is to be produced. (The link step is described in Section 2.3.) The object module created will be named <source-name>.o. Note that in a “makefile” where CDF_INC is imported, $(CDF_INC) would be specified instead of ${CDF_INC}.

1.1.3 MS-DOS Systems, Microsoft C

An example of the command to compile a source file on MS-DOS systems using Microsoft C would be as follows:

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

where <source-name>.c is the name of the source file being compiled (the .c extension is required) and <inc-path> is the file name of the directory containing cdf.h. You will need to know where on your system cdf.h has been installed. <inc-path> may be either an absolute or relative file name.

You may also need to specify the location of system include files. For Microsoft C this is usually accomplished by setting MS-DOS environment variables. Consult the Microsoft C documentation for more information.

The /c option specifies that only an object module is to be produced. (The link step and a combined compile/link step are described in Section 2.4.) The object module will be named <source-name>.obj.

The /AL option specifies that the object module is to be compiled using the large memory model. The CDF library for Microsoft C supplied with the CDF distribution is compiled using the large memory model. If you need to use the huge memory model for your application, you will also need to rebuild the CDF library for the huge memory model.

The /FPi option specifies how floating-point operations will be handled at run-time. With this option a math coprocessor will be used if it exists; otherwise, the emulation library will be called. Using this option allows your program to run on any MS-DOS system regardless of whether or not a math coprocessor exists. If you know that a math coprocessor exists, you may want to use a floating-point option that provides better performance.

You may instead want to use the Microsoft Programmer’s Workbench (PWB) development environment to compile/link your applications. The options shown above for the command line compiler are specified in the development environment. Consult the documentation for the PWB for the steps necessary to compile/link your application.

1.1.4 MS-DOS Systems, Borland C

An example of the command to compile a source file on MS-DOS systems using Borland C would be as follows:

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

where `<source-name>.c` is the name of the source file being compiled (the `.c` extension is required) and `<inc-path>` is the file name of the directory containing `cdf.h`. You will need to know where on your system `cdf.h` has been installed. `<inc-path>` may be either an absolute or relative file name.

You may also need to specify the location of system include files. For Borland C it may be necessary to also specify `-I<bc-inc-path>` where `<bc-inc-path>` is the location of the Borland C system include files. Consult the Borland C documentation for more information.

The `-c` option specifies that only an object module is to be produced. (The link step and a combined compile/link step are described in Section 2.5.) The object module will be named `<source-name>.obj`.

The `-ml` option specifies that the object module is to be compiled using the large memory model. The CDF library for Borland C supplied with the CDF distribution is compiled using the large memory model. If you need to use the huge memory model for your application, you will also need to rebuild the CDF library for the huge memory model.

You may instead want to use the Borland Integrated Developers Environment (IDE) to compile/link your applications. The options shown above for the command line compiler are specified in the development environment. Consult the documentation for the IDE for the steps necessary to compile/link your application.

1.1.5 Macintosh Systems, Symantec THINK C

Symantec THINK C has a development environment in which an application is compiled. The folder containing `cdf.h` must be in the project tree for your application (or in the THINK C project tree). You may also use an `Aliases` folder in your project tree to make known to THINK C the location of the folder containing `cdf.h`. Consult the THINK C documentation for complete details.

You should also set the following THINK C compile options:

1. Check the `#define __STDC__` check box.
2. Check the `4-byte ints` check box.[1]
3. Check the `8-byte doubles` check box.
4. Do not check the `Native floating-point format` check box.
5. Check the `Far data` check box (under `Set Project Type...`) if the link step fails due to a “data segment too big” error.

[1] Previous to CDF V2.6 the `4-byte ints` box was not checked. Because the CDF library must now be built using 4-byte `int`'s, the ANSI C run-time library with which it is linked into an application must also be built using 4-byte `int`'s. Therefore, your application code must be compiled using 4-byte `int`'s. An ANSI C run-time library compatible with the CDF library is also provided with the CDF distribution. It is described in Section 2.5.1.

1.1.6 Macintosh Systems, MPW C

Macintosh Programmer's Workshop (MPW) C uses a command line instruction to compile source files. This command may be entered either on the MPW Worksheet or in an MPW makefile. An example of the command to compile a source file using MPW C would be as follows:

```
C -i <inc-path> -model far <source-name>.c
```

where <source-name>.c is the name of the source file being compiled and <inc-path> is an absolute or relative file name of the folder containing cdf.h. You will need to know where on your system cdf.h has been installed. File names on a Macintosh are constructed by separating volume/folder names with colons and terminating the file name with a colon if it is a folder rather than a file (e.g., Disk1:cdf26-dist:include:). The name of the object module produced will be <source-name>.c.o in the current directory. Note that this example also assumes that <source-name>.c is in the current directory.

The -model far option indicates that the 32K restrictions on the size of code segments, the jump table, and the global data area are to be removed. This option is necessary in order to successfully link to the CDF library provided for MPW applications. (See Section 2.5.2.)

If your application is fairly large, you may also find it necessary to use the -s option to place your compiled source code (object modules) into separate segments when linked. Consult the MPW C documentation for more details.

1.2 Specifying cdf.h Location in the Source File

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

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

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

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

Chapter 2

Linking

Your applications must be linked with the CDF library.[1] Both the Standard and Internal interfaces for C applications are built into the CDF library. On VMS systems a logical name, CDF$LIB, which specifies the location of the CDF library, is defined in the definitions file, DEFINITIONS.COM, provided with the CDF distribution. On UNIX systems an environment variable, CDF_LIB, which serves the same purpose, is defined in the definitions file definitions.<shell-type> where <shell-type> is the type of shell being used: C for the C-shell (csh and tcsh), K for the Korn (ksh), BASH, and POSIX shells, and B for the Bourne shell (sh). This section assumes that you are using the appropriate definitions file on those systems. On MS-DOS and Macintosh (MacOS) systems, definitions files are not available. The location of the CDF library is specified as described in the appropriate sections for those systems.

2.1 VAX/VMS & VAX/OpenVMS Systems

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

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

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

It may also be necessary to specify SYS$LIBRARY:VAXCRTL/LIBRARY at the end of the LINK command if your system does not properly define LNK$LIBRARY (or LNK$LIBRARY_1, etc.).

[1] A shareable version of the CDF library is also available on VMS and some flavors of UNIX. Its use is described in Chapter 3. A dynamic link library (DLL), LIBCDF.DLL, is available on MS-DOS systems for Microsoft and Borland Windows applications. Consult the Microsoft and Borland documentation for details on using a DLL. Note that the DLL for Microsoft is created using Microsoft C 7.00.

2.2 DEC Alpha/OpenVMS Systems

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

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

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

2.3 UNIX Systems

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

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

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

2.3.1 Combining the Compile and Link

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

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

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

2.4 MS-DOS Systems, Microsoft C

An example of the command to link your application with the CDF library (LIBCDF.LIB) on MS-DOS systems using Microsoft C would be as follows:[2]

```
  > LINK /NOI /NOD /ST:<size> <objs>,<exe>,nul.map,<lib-path>libcdf+LLIBCE;
```

where <objs> is your application's object module(s) (the .obj extension is not necessary); <exe> is the name of the executable file to be created (with a default extension of .exe); and <lib-path> is the file name of the directory containing the CDF library. You will need to know where on your system the CDF library has been installed. <lib-path> may be either an absolute or relative file name.

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

The /NOI option specifies that function names are to remain case-sensitive. The /NOD option specifies that the default libraries (named in object files) should not be used. The needed libraries must instead be named in the link command. The C run-time library shown, LLIBCE, assumes the large memory model and emulated floating-point operations if a coprocessor does not exist at run-time. If Microsoft C 7.00 is being used with the CDF library built for Microsoft C 6.00, the library named OLDNAMES must also be specified to handle the function naming differences between the Microsoft C 6.00 and Microsoft C 7.00 run-time libraries.

NOTE: The same memory model must have been used to compile your application's source files and the CDF library. The CDF library for Microsoft C supplied with the CDF distribution is compiled using the large memory model. If you need to use the huge memory model for your application, you will also have to rebuild the CDF library for the huge memory model.

You may instead want to use the Microsoft Programmer's Workbench (PWB) development environment to compile/link your applications. The options shown above for the command line linker are specified in the development environment. Consult the documentation for the PWB for the steps necessary to compile/link your application.

2.5 MS-DOS Systems, Borland C

An example of the command to link your application with the CDF library (LIBCDF.LIB) on MS-DOS systems using Borland C would be as follows:

[2] This example assumes you have properly set the MS-DOS environment variables (e.g., INCLUDE and LIB) used by the Microsoft C compiler and linker. Note that there are some differences between the Microsoft C 6.00 and Microsoft C 7.00 run-time libraries (regarding system function names). The CDF distribution for msdos is supplied with CDF libraries built for both Microsoft C 6.00 and Microsoft C 7.00. It is also assumed that the appropriate CDF library was renamed to LIBCDF.LIB.

```
> TLINK /x /L<bc>\lib C0L <object-file(s)>,<executable>,,
        <lib-path>libcdf.lib CL EMU MATHL
```

where `<object-file(s)>` is your application's object module(s) (the `.obj` extension is not necessary); `<executable>` is the name of the executable file to be created (with a default extension of `.exe`); and `<lib-path>` is the file name of the directory containing the CDF library. You will need to know where on your system the CDF library has been installed. `<inc-path>` may be either an absolute or relative file name.

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

You may instead want to use the Borland C Integrated Developers Environment (IDE) to compile/link your applications. The options shown above for the command line compiler are specified in the development environment. Consult the documentation for the IDE for the steps necessary to compile/link your application.

NOTE: The same memory model must have been used to compile your application's source files and the CDF library. The CDF library for Borland C supplied with the CDF distribution is compiled using the large memory model. If you need to use the huge memory model for your application, you will also have to rebuild the CDF library for the huge memory model.

2.5.1 Macintosh Systems, Symantec THINK C

The CDF library for Symantec THINK C is distributed as eight project files named `libcdf1.`π, `libcdf2.`π, `libcdf3.`π, `libcdf4.`π, `libcdf5.`π, `libcdf6.`π, `libcdf7.`π, and `libcdf8.`π. This is necessary because the CDF library must be split into eight separate segments. Each CDF library project file must be placed in a separate segment in your application project.

Symantec THINK C has a development environment in which an application is linked. The folder containing the CDF library project files must be in the project tree for your application (or in the THINK C project tree). You may also use an `Aliases` folder in your project tree to make known to THINK C the location of the folder containing the CDF library project files. Consult the THINK C documentation for complete details.

You should also set the following THINK C link option(s):

1. Check the `Far data` check box if the link step fails due to a "data segment too big" error.

The CDF library expects to be linked with an ANSI library of C run-time functions built using 4-byte `int`'s[3] and 8-byte `float`'s. The standard ANSI C run-time library supplied with THINK C does not use 4-byte

[3] Previous to CDF V2.6 this was not the case.

int's or 8-byte float's. You will either need to build a version of the ANSI C run-time library with the above requirements or use the ANSI library supplied with the CDF distribution named ANSIcdf.π (which is the ANSI library used to link the CDF toolkit and test programs).

The CDF library does not use Macintosh resources. If your application uses resources, they must be compiled/linked as described in the THINK C documentation.

2.5.2 Macintosh Systems, MPW

Macintosh Programmer's Workshop (MPW) uses a command line instruction to link an application. This command may be entered either on the MPW Worksheet or in an MPW makefile. An example of the command to link an application with the CDF library (libcdf.o) using MPW would be as follows:

```
Link -t APPL -c '????' -model far δ
     <object-file>.c.o <object-file>.c.o ... <object-file>.c.o δ
     <lib-path>libcdf.o δ
     "{CLibraries}"<c-lib> "{CLibraries}"<c-lib> ... "{CLibraries}"<c-lib> δ
     "{Libraries}"<mac-lib> "{Libraries}"<mac-lib> ... "{Libraries}"<mac-lib> δ
     -o <appl-path>
```

where <object-file>.c.o is the name of one or more object modules being linked; <lib-path> is an absolute or relative file name of the folder containing libcdf.o; <c-lib> is the name of one or more needed C libraries; <mac-lib> is the name of one or more needed Macintosh libraries; and <appl-path> is the file name of the application being linked. You will need to know where on your system libcdf.o has been installed. File names on a Macintosh are constructed by separating volume/folder names with colons and terminating the file name with a colon if it is a folder rather than a file (e.g., Disk1:cdf26-dist:lib:). Note that this example assumes that <object-file>.c.o is in the current directory.

The C libraries that may be needed for the link are StdCLib.o, Math.o, and CSANELib.o. The Macintosh libraries that may be needed are Runtime.o and Interface.o. Note that "{CLibraries}" and "{Libraries}" are predefined by MPW.

The -model far option indicates that the 32K restrictions on the size of code segments, the jump table, and the global data area are to be removed. This option is necessary in order to successfully link to the CDF library provided for MPW applications.

The CDF library does not use Macintosh resources. If your application uses resources, they must be compiled/linked as described in the MPW documentation.

Chapter 3

Linking, Shared CDF Library

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

Machine/Operating System	Shared CDF Library
VAX (VMS & OpenVMS)	LIBCDF.EXE
DEC Alpha (OpenVMS)	LIBCDF.EXE
Sun (SunOS)	libcdf.so
Sun (SOLARIS)	libcdf.so
HP 9000 (HP-UX)	libcdf.sl
IBM RS6000 (AIX)	libcdf.o
DEC Alpha (OSF/1)	libcdf.so
SGi (IRIX 5.x & 6.x)	libcdf.so

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

3.1 VAX (VMS & OpenVMS)

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

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

NOTE: On VAX/VMS and VAX/OpenVMS systems the shareable CDF library may also be installed in `SYS$SHARE`. If that is the case, the link command would be as follows:

```
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  SYS$SHARE:LIBCDF/SHAREABLE
  SYS$SHARE:VAXCRTL/SHAREABLE
  <Control-Z>
```

3.2 DEC Alpha (OpenVMS)

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

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

NOTE: On DEC Alpha/OpenVMS systems the shareable CDF library may also be installed in `SYS$SHARE`. If that is the case, the link command would be as follows:

```
$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
  SYS$SHARE:LIBCDF/SHAREABLE
  SYS$LIBRARY:<crtl>/LIBRARY
  <Control-Z>
```

3.3 Sun (SunOS)

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

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required) and `<exe-file>` is the name of the executable file created. Note that in a "makefile" where `CDF_LIB` is imported, `$(CDF_LIB)` would be specified instead of `${CDF_LIB}`. Also, `-ldl` may not be necessary on some SunOS systems.

3.4 Sun (SOLARIS)

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

where `<object-file(s)>.o` is your application's object module(s) (the `.o` extension is required) and `<exe-file>` is the name of the executable file created. Note that in a "makefile" where `CDF_LIB` is imported, `$(CDF_LIB)` would be specified instead of `${CDF_LIB}`.

3.5 HP 9000 (HP-UX)

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

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created. Note that in a "makefile" where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

3.6 IBM RS6000 (AIX)

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

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created. Note that in a "makefile" where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

3.7 DEC Alpha (OSF/1)

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

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created. Note that in a "makefile" where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

On a DEC Alpha running OSF/1, when executing a program linked to the shareable CDF library, the environment variable LD_LIBRARY_PATH must be set to include the directory containing libcdf.so. [1]

3.8 SGi (IRIX 5.x & 6.x)

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

where <object-file(s)>.o is your application's object module(s) (the .o extension is required) and <exe-file> is the name of the executable file created. Note that in a "makefile" where CDF_LIB is imported, $(CDF_LIB) would be specified instead of ${CDF_LIB}.

[1] Other Unix boxes like SGi and Sun also require this environment variable setup.

Chapter 4

Programming Interface

The following sections describe various aspects of the C programming interface for CDF applications. These include constants and types defined for use by all CDF application programs written in C. These constants and types are defined in `cdf.h`. The file `cdf.h` should be `#include`'d in all application source files referencing CDF routines/parameters.

4.1 Item Referencing

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

4.2 Defined Types

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

`CDFstatus` All CDF functions except `CDFvarNum` and `CDFattrNum` are of type `CDFstatus`. They return a status code indicating the completion status of the function. The `CDFerror` function can be used to inquire the meaning of any status code. Appendix A lists the possible status codes along with their explanations. Chapter 7 describes how to interpret status codes.

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

4.3 CDFstatus Constants

These constants are of type `CDFstatus`.

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

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

Chapter 7 describes how to use these constants to interpret status codes.

4.4 CDF Formats

`SINGLE_FILE` The CDF consists of only one file.

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

4.5 CDF Data Types

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

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

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

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

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

4.6 Data Encodings

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

HOST_ENCODING Indicates host machine data representation (native). This encoding will provide the greatest performance when reading/writing on a machine of the same type.

NETWORK_ENCODING Indicates network transportable data representation (XDR).

VAX_ENCODING Indicates VAX data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.

ALPHAVMSd_ENCODING Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.

ALPHAVMSg_ENCODING Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's G_FLOAT representation.

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

ALPHAOSF1_ENCODING Indicates DEC Alpha running OSF/1 data representation.

SUN_ENCODING Indicates SUN data representation.

SGi_ENCODING Indicates Silicon Graphics Iris and Power Series data representation.

DECSTATION_ENCODING Indicates DECstation data representation.

IBMRS_ENCODING Indicates IBMRS data representation (IBM RS6000 series).

HP_ENCODING Indicates HP data representation (HP 9000 series).

PC_ENCODING Indicates PC data representation.

NeXT_ENCODING Indicates NeXT data representation.

MAC_ENCODING Indicates Macintosh data representation.

When creating a CDF (via the Standard Interface) or respecifying a CDF's encoding (via the Internal Interface), you may specify any of the encodings listed above. Specifying the host machine's encoding explicitly has the same effect as specifying HOST_ENCODING.

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

4.7 Data Decodings

A CDF's decoding affects how its attribute entry and variable data values are passed out to a calling application. The decoding for a CDF may be selected and reselected any number of times while the CDF is open. Selecting a decoding does not affect how the values are stored in the CDF file(s) — only how 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 Internal Interface with the <SELECT_,CDF_DECODING_> operation. The Concepts chapter in the CDF User's Guide describes those situations in which a decoding other than HOST_DECODING may be desired.

4.8 Variable Majorities

A CDF's variable majority determines the order in which variable values (within the variable arrays) are stored in the CDF file(s). The majority is the same for rVariable and zVariables.

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

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

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

4.9 Record/Dimension Variances

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

`VARY` True record or dimension variance.

`NOVARY` False record or dimension variance.

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

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

4.10 Compressions

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

`NO_COMPRESSION` No compression.

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

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

`HUFF_COMPRESSION` Huffman compression. There is one parameter.

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

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

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

`GZIP_COMPRESSION` [1] Gnu's "zip" compression. There is one parameter.

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

4.11 Sparseness

4.11.1 Sparse Records

The following types of sparse records for variables are supported.

`NO_SPARSERECORDS` No sparse records.

`PAD_SPARSERECORDS` Sparse records — the variable's pad value is used when reading values from a missing record.

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

`PREV_SPARSERECORDS` Sparse records — values from the previous existing record are used when reading values from a missing record. If there is no previous existing record the variable's pad value is used.

4.11.2 Sparse Arrays

The following types of sparse arrays for variables are supported.[2]

`NO_SPARSEARRAYS` No sparse arrays.

4.12 Attribute Scopes

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

`GLOBAL_SCOPE` Indicates that an attribute's scope is global (applies to the CDF as a whole).

`VARIABLE_SCOPE` Indicates that an attribute's scope is by-variable. (Each rEntry or zEntry corresponds to an rVariable or zVariable, respectively.)

4.13 Read-Only Modes

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

`READONLYon` Turns on read-only mode.

`READONLYoff` Turns off read-only mode.

4.14 zModes

Once a CDF has been opened, it may be placed into one of two variations of zMode. zMode is fully explained in the Concepts chapter in the CDF User's Guide. A zMode is selected for a CDF via the Internal Interface using the `<SELECT_,CDF_zMODE_>` operation.

`zMODEoff` Turns off zMode.

[2] Obviously, sparse arrays are not yet supported.

`zMODEon1` Turns on zMode/1.

`zMODEon2` Turns on zMode/2.

4.15 `-0.0` to `0.0` Modes

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

`NEGtoPOSfp0on` Convert `-0.0` to `0.0` when read from or written to a CDF.

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

4.16 Operational Limits

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

`CDF_MAX_DIMS` Maximum number of dimensions for the rVariables or a zVariable.

`CDF_MAX_PARMS` Maximum number of compression or sparseness parameters.

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

4.17 Limits of Names and Other Character Strings

`CDF_PATHNAME_LEN` Maximum length of a CDF file name (excluding the `NUL`[3] terminator and the `.cdf` or `.vnn` appended by the CDF library to construct file names). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating systems being used (including logical names on VMS systems and environment variables on UNIX systems).

`CDF_VAR_NAME_LEN` Maximum length of a variable name (excluding the `NUL` terminator).

`CDF_ATTR_NAME_LEN` Maximum length of an attribute name (excluding the `NUL` terminator).

`CDF_COPYRIGHT_LEN` Maximum length of the CDF copyright text (excluding the `NUL` terminator).

`CDF_STATUSTEXT_LEN` Maximum length of the explanation text for a status code (excluding the `NUL` terminator).

[3] The ASCII null character, `0x0`.

Chapter 5

Standard Interface

The following sections describe the Standard Interface routines callable from C applications. Most functions return a status code of type CDFstatus (see Chapter 7). The Internal Interface is described in Chapter 6. An application can use both interfaces when necessary. Note that zVariables and vAttribute zEntries are only accessible via the Internal Interface.

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

5.1 CDFcreate

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

CDFcreate creates a CDF as defined by the arguments. A CDF cannot be created if it already exists. (The existing CDF will not be overwritten.) If you want to overwrite an existing CDF, you must first open it with CDFopen, delete it with CDFdelete, and then recreate it with CDFcreate. If the existing CDF is corrupted, the call to CDFopen will fail. (An error code will be returned.) In this case you must delete the CDF at the command line. Delete the dotCDF file (having an extension of .cdf), and if the CDF has the multi-file format, delete all of the variable files (having extensions of .v0,.v1,... and .z0,.z1,...).

The arguments to CDFcreate are defined as follows:

CDFname The file name of the CDF to create. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). A

CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS 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). If there are zero (0) dimensions, this argument is ignored (but must be present).

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

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

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

When a CDF is created, both read and write access are allowed. The default format for a CDF created with `CDFcreate` is specified in the configuration file of your CDF distribution. Consult your system manager for this default. The `CDFlib` function (Internal Interface) may be used to change a CDF's format.

NOTE: `CDFclose` must be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 5.5).

5.1.1 Example(s)

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

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

```
.
```

ROW_MAJOR and NETWORK_ENCODING are defined in cdf.h.

5.2 CDFopen

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

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

The arguments to CDFopen are defined as follows:

CDFname The file name of the CDF to open. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters (excluding the NUL terminator). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS 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 (see Section 5.5).

5.2.1 Example(s)

The following example will open a CDF named NOAA1.

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

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

5.3 CDFdoc

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

CDFdoc is used to inquire general documentation about a CDF. The version/release of the CDF library that created the CDF is provided (e.g., CDF V2.4 is version 2, release 4) along with the CDF copyright notice.

The arguments to CDFdoc are defined as follows:

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

version
The version number of the CDF library that created the CDF.

release
The release number of the CDF library that created the CDF.

copyRight
The copyright notice of the CDF library that created the CDF. This character string must be large enough to hold CDF_COPYRIGHT_LEN + 1 characters (including the NUL terminator). This string will contain a newline character after each line of the copyright notice.

The copyright notice is formatted for printing without modification. The version and release are used together (e.g., CDF V2.4 is version 2, release 4).

5.3.1 Example(s)

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

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

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

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

5.4 CDFinquire

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

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

The arguments to `CDFinquire` are defined as follows:

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

`numDims` The number of dimensions for the rVariables in the CDF.

`dimSizes` The dimension sizes of the rVariables in the CDF. `dimSizes` is a 1-dimensional array containing one element per dimension. Each element of `dimSizes` receives the corresponding dimension size. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary).

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

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

`maxRec` The maximum record number written to an rVariable in the CDF. Note that the maximum record number written is also kept separately for each rVariable in the CDF. The value of `maxRec` is the largest of these. Some rVariables may have fewer records actually written. `CDFlib` (Internal Interface) may be used to inquire the maximum record written for an individual rVariable (see Section 6).

`numVars` The number of rVariables in the CDF.

`numAttrs` The number of attributes in the CDF.

5.4.1 Example(s)

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

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

5.5 CDFclose

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

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

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

The arguments to CDFclose are defined as follows:

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

5.5.1 Example(s)

The following example will close an open CDF.

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

5.6 CDFdelete

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

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

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

The arguments to CDFdelete are defined as follows:

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

5.6.1 Example(s)

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

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

5.7 CDFerror

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

`CDFerror` is used to inquire the explanation of a given status code (not just error codes). Chapter 7 explains how to interpret status codes and Appendix A lists all of the possible status codes.

The arguments to `CDFerror` are defined as follows:

`status` The status code to check.

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

5.7.1 Example(s)

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

```
.
```

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

5.8 CDFattrCreate

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

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

The arguments to CDFattrCreate are defined as follows:

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

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

attrScope The scope of the new attribute. Specify one of the scopes described in Section 4.12.

attrNum The number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may be determined with the CDFattrNum function.

5.8.1 Example(s)

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

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

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

5.9 CDFattrNum

```
long CDFattrNum(   /* out -- Attribute number. */
CDFid id,          /* in  -- CDF id */
char *attrName);   /* in  -- Attribute name */
```

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

The arguments to CDFattrNum are defined as follows:

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

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

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

5.9.1 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. CDFattrRename is described in Section 5.10.

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

5.10 CDFattrRename

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

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

The arguments to CDFattrRename are defined as follows:

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

attrNum — The number of the attribute to rename. This number may be determined with a call to CDFattrNum (see Section 5.9).

attrName — The new attribute name. This may be at most CDF_ATTR_NAME_LEN characters (excluding the NUL terminator). Attribute names are case-sensitive.

5.10.1 Example(s)

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

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

5.11 CDFattrInquire

```
CDFstatus CDFattrInquire(   /* out -- Completion status code. */
CDFid id,                   /* in  -- CDF identifier. */
long attrNum,               /* in  -- Attribute number. */
char *attrName,             /* out -- Attribute name. */
long *attrScope,            /* out -- Attribute scope. */
long *maxEntry);            /* out -- Maximum gEntry or rEntry number. */
```

CDFattrInquire is used to inquire about the specified attribute. To inquire about a specific attribute entry, use CDFattrEntryInquire (Section 5.12).

The arguments to CDFattrInquire are defined as follows:

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

attrNum
The number of the attribute to inquire. This number may be determined with a call to CDFattrNum (see Section 5.9).

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

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

maxEntry
For gAttributes this is the maximum gEntry number used. For vAttributes this is the maximum rEntry number used. In either case this may not correspond with the number of entries (if some entry numbers were not used). The number of entries actually used may be inquired with the CDFlib function (see Section 6). If no entries exist for the attribute, then a value of -1 will be passed back.

5.11.1 Example(s)

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

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

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

5.12 CDFattrEntryInquire

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

CDFattrEntryInquire is used to inquire about a specific attribute entry. To inquire about the attribute in general, use CDFattrInquire (see Section 5.11). CDFattrEntryInquire would normally be called before calling CDFattrGet in order to determine the data type and number of elements (of that data type) for an entry. This would be necessary to correctly allocate enough memory to receive the value read by CDFattrGet.

The arguments to CDFattrEntryInquire are defined as follows:

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

attrNum The attribute number for which to inquire an entry. This number may be determined with a call to CDFattrNum (see Section 5.9).

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

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

numElements The number of elements of the data type. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

5.12.1 Example(s)

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

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

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

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

5.13 CDFattrPut

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

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

The arguments to CDFattrPut are defined as follows:

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

attrNum — The attribute number. This number may be determined with a call to CDFattrNum (see Section 5.9).

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

`dataType` The data type of the specified entry. Specify one of the data types defined in Section 4.5.

`numElements` The number of elements of the data type. For character data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

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

5.13.1 Example(s)

The following example writes two attribute entries. The first is to gEntry number zero (0) of the gAttribute `TITLE`. The second is to the variable scope attribute `VALIDs` for the rEntry that corresponds to the rVariable `TMP`.

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

```
   numElements = 2;
   status = CDFattrPut (id, CDFattrNum(id,"VALIDs"), CDFvarNum(id,"TMP"),
                        CDF_INT2, numElements, TMPvalids);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
```

5.14 CDFattrGet

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

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

The arguments to CDFattrGet are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate or CDFopen.
attrNum	The attribute number. This number may be determined with a call to CDFattrNum (see Section 5.9).
entryNum	The entry number. If the attribute is global in scope, this is simply the gEntry number and has meaning only to the application. If the attribute is variable in scope, this is the number of the associated rVariable (the rVariable being described in some way by the rEntry).
value	The value read. This buffer must be large enough to hold the value. The function CDFattrEntryInquire would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.

5.14.1 Example(s)

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

```
   .
   .
```

```
#include "cdf.h"
.
.
CDFid     id;              /* CDF identifier. */
CDFstatus status;          /* Returned status code. */
long      attrN;           /* Attribute number. */
long      entryN;          /* Entry number. */
long      dataType;        /* Data type. */
long      numElems;        /* Number of elements (of data type). */
void      *buffer;         /* Buffer to receive value. */
.
.
attrN = CDFattrNum (id, "UNITS");
if (attrN < 0) UserStatusHandler (attrN);  /* If less than zero (0), then
                                              it must be a warning/error
                                              code. */

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

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

if (dataType == CDF_CHAR) {
  buffer = (char *) malloc (numElems + 1);
  if (buffer == NULL)...

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

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

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

  free (buffer);
}
.
.
```

5.15 CDFvarCreate

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

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

The arguments to CDFvarCreate are defined as follows:

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

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

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

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

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

dimVariances
: The rVariable's dimension variances. Each element of dimVariances specifies the corresponding dimension variance. For each dimension specify one of the variances defined in Section 4.9. If there are zero (0) dimensions, 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 rVariables's number may be determined with the CDFvarNum function.

5.15.1 Example(s)

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

```
   .
   .
   #include "cdf.h"
   .
   .
   CDFid     id;                                 /* CDF identifier. */
   CDFstatus status;                             /* Returned status code. */
   static long EPOCHrecVary = {VARY};            /* EPOCH record variance. */
```

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

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

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

status = CDFvarCreate (id, "TEMPERATURE", CDF_REAL4, 1,
                       TMPrecVary, TMPdimVarys, &TMPvarNum);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

5.16 CDFvarNum

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

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

The arguments to CDFvarNum are defined as follows:

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

varName — The name of the rVariable for which to search. This may be at most CDF_VAR_NAME_LEN characters (excluding the NUL terminator). Variable names are case-sensitive.

CDFvarNum may be used as an embedded function call when an rVariable number is needed.

5.16.1 Example(s)

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

```
.
.
#include "cdf.h"
.
.
CDFid     id;                          /* CDF identifier. */
CDFstatus status;                      /* Returned status code. */
char      varName[CDF_VAR_NAME_LEN+1]; /* rVariable name. */
long      dataType;                    /* Data type of the rVariable. */
long      numElements;                 /* Number of elements (of the
                                          data type). */
long      recVariance;                 /* Record variance. */
long      dimVariances[CDF_MAX_DIMS];  /* Dimension variances. */
.
.
status = CDFvarInquire (id, CDFvarNum(id,"LATITUDE"), varName, &dataType,
                        &numElements, &recVariance, dimVariances);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

In this example the rVariable named LATITUDE was inquired. Note that if LATITUDE did not exist in the CDF, the call to CDFvarNum would have returned an error code. Passing that error code to CDFvarInquire as an rVariable number would have resulted in CDFvarInquire also returning an error code. Also note that the name written into varName is already known (LATITUDE). In some cases the rVariable names will be unknown — CDFvarInquire would be used to determine them. CDFvarInquire is described in Section 5.18.

5.17 CDFvarRename

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

CDFvarRename is used to rename an existing rVariable. A variable (rVariable or zVariable) with the same name must not already exist in the CDF.

The arguments to CDFvarRename are defined as follows:

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

varNum — The number of the rVariable to rename. This number may be determined with a call to CDFvarNum (see Section 5.16).

varName — The new rVariable name. This may be at most CDF_VAR_NAME_LEN characters (excluding the NUL terminator). Variable names are case-sensitive.

5.17.1 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.

```
.
.
#include "cdf.h"
.
.
CDFid     id;       /* CDF identifier. */
CDFstatus status;   /* Returned status code. */
long      varNum;   /* rVariable number. */
.
.
varNum = CDFvarNum (id, "TEMPERATURE");
if (varNum < 0) {
  if (varNum != NO_SUCH_VAR) UserStatusHandler (varNum);
}
else {
  status = CDFvarRename (id, varNum, "TMP");
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

5.18 CDFvarInquire

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

CDFvarInquire is used to inquire about the specified rVariable. This function would normally be used before reading rVariable values (with CDFvarGet or CDFvarHyperGet) to determine the data type and number of elements (of that data type).

The arguments to CDFvarInquire are defined as follows:

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

varNum
: The number of the rVariable to inquire. This number may be determined with a call to CDFvarNum (see Section 5.16).

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

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

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

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

dimVariances
: The dimension variances. Each element of dimVariances receives the corresponding dimension variance. The dimension variances are defined in Section 4.9. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary).

5.18.1 Example(s)

The following example inquires about an rVariable named HEAT_FLUX in a CDF. Note that the rVariable name returned by CDFvarInquire will be the same as that passed in to CDFvarNum.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                          /* CDF identifier. */
CDFstatus status;                      /* Returned status code. */
char      varName[CDF_VAR_NAME_LEN+1]; /* rVariable name, +1
                                          for NUL terminator. */
long      dataType;                    /* Data type of the rVariable. */
long      numElems;                    /* Number of elements (of data
                                          type). */
```

```
   long      recVary;                     /* Record variance. */
   long      dimVarys[CDF_MAX_DIMS];      /* Dimension variances (allocate
                                             to allow the maximum number of
                                             dimensions). */
   .
   .
   status = CDFvarInquire (id, CDFvarNum(id,"HEAT_FLUX"), varName, &dataType,
                           &numElems, &recVary, dimVarys);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
```

5.19 CDFvarPut

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

CDFvarPut is used to write a single value to an rVariable. CDFvarHyperPut may be used to write more than one rVariable value with a single call (see Section 5.21).

The arguments to CDFvarPut are defined as follows:

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

varNum
The number of the rVariable to which to write. This number may be determined with a call to CDFvarNum (see Section 5.16).

recNum
The record number at which to write.

indices
The array indices within the specified record at which to write. Each element of indices specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

value
The value to write. The value is written to the CDF from memory address value.

5.19.1 Example(s)

The following example writes values to the rVariable named LATITUDE in a CDF whose rVariables are 2-dimensional with dimension sizes [360,181]. For LATITUDE the record variance is NOVARY, the dimension variances are [NOVARY,VARY], and the data type is CDF_INT2.

```
   .
   .
   #include "cdf.h"
   .
   .
   CDFid       id;                      /* CDF identifier. */
   CDFstatus   status;                  /* Returned status code. */
   short       lat;                     /* Latitude value. */
   long        varN;                    /* rVariable number. */
   static long recNum = 0;              /* Record number. */
   static long indices[2] = {0,0};      /* Dimension indices. */
   .
   .
   varN = CDFvarNum (id, "LATITUDE");
   if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), not a
                                               rVariable number but
                                               rather a warning/error code. */

   for (lat = -90; lat <= 90; lat ++) {
      indices[1] = 90 + lat;
      status = CDFvarPut (id, varN, recNum, indices, &lat);
      if (status != CDF_OK) UserStatusHandler (status);
   }
   .
   .
```

Since the record variance is `NOVARY`, the record number (`recNum`) is set to zero (0). Also note that because the dimension variances are `[NOVARY,VARY]`, only the second dimension is varied as values are written. (The values are "virtually" the same at each index of the first dimension.)

5.20 CDFvarGet

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

`CDFvarGet` is used to read a single value from an rVariable. `CDFvarHyperGet` may be used to read more than one rVariable value with a single call (see Section 5.22).

The arguments to `CDFvarGet` are defined as follows:

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

`varNum` The number of the rVariable from which to read. This number may be determined with a call to `CDFvarNum` (see Section 5.16).

`recNum` The record number at which to read.

`indices` The array indices within the specified record at which to read. Each element of `indices` specifies the corresponding dimension index. If there are zero (0) dimensions, this argument is ignored (but must be present).

`value` The value read. This buffer must be large enough to hold the value. `CDFvarInquire` would be used to determine the rVariable's data type and number of elements (of that data type) at each value. The value is read from the CDF and placed at memory address `value`.

5.20.1 Example(s)

The following example will read and hold an entire record of data from an rVariable. The CDF's rVariables are 3-dimensional with sizes `[180,91,10]`. For this rVariable the record variance is `VARY`, the dimension variances are `[VARY,VARY,VARY]`, and the data type is `CDF_REAL4`.

```
.
.
#include "cdf.h"
.
.
CDFid     id;                /* CDF identifier. */
CDFstatus status;            /* Returned status code. */
float     tmp[180][91][10];  /* Temperature values. */
long      indices[3];        /* Dimension indices. */
long      varN;              /* rVariable number. */
long      recNum;            /* Record number. */
long      d0, d1, d2;        /* Dimension index values. */
.
.
varN = CDFvarNum (id, "Temperature");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), then
                                            it is actually a warning/error
                                            code. */

recNum = 13;

for (d0 = 0; d0 < 180; d0++) {
   indices[0] = d0;
   for (d1 = 0; d1 < 91; d1++) {
      indices[1] = d1;
      for (d2 = 0; d2 < 10; d2++) {
         indices[2] = d2;
         status = CDFvarGet (id, varN, recNum, indices, &tmp[d0][d1][d2]);
         if (status != CDF_OK) UserStatusHandler (status);
      }
   }
}
.
```

.

5.21 CDFvarHyperPut

```
CDFstatus CDFvarHyperPut(   /* out -- Completion status code. */
CDFid id,                   /* in  -- CDF identifier. */
long varNum,                /* in  -- rVariable number. */
long recStart,              /* in  -- Starting record number. */
long recCount,              /* in  -- Number of records. */
long recInterval,           /* in  -- Interval between records. */
long indices[],             /* in  -- Dimension indices of starting value. */
long counts[],              /* in  -- Number of values along each dimension. */
long intervals[],           /* in  -- Interval between values along each
                                      dimension. */
void *buffer);              /* in  -- Buffer of values. */
```

`CDFvarHyperPut` is used to write a buffer of one or more values to an rVariable. It is important to know the variable majority of the CDF before using `CDFvarHyperPut` because the values in the buffer to be written must be in the same majority. `CDFinquire` 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 arguments to `CDFvarHyperPut` are defined as follows:

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

`varNum` The number of the rVariable to which to write. This number may be determined with a call to `CDFvarNum` (see Section 5.16).

`recStart` The record number at which to start writing.

`recCount` The number of records to write.

`recInterval` The interval between records for subsampling[1] (e.g., an interval of 2 means write to every other record).

`indices` The indices (within each record) at which to start writing. Each element of `indices` specifies the corresponding dimension index. If there are zero (0) dimensions, this argument is ignored (but must be present).

`counts` The number of values along each dimension to write. Each element of `count` specifies the corresponding dimension count. If there are zero (0) dimensions, this argument is ignored (but must be present).

`intervals` For each dimension the interval between values for subsampling[2] (e.g., an interval of 2 means write to every other value). `intervals` is a 1-dimensional array containing one element per rVariable dimension. Each element of `intervals` specifies the corresponding dimension interval. If there are zero (0) dimensions, this argument is ignored (but a place holder is necessary).

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

[2] Again, not the best term.

`buffer` The buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address `buffer` are written to the CDF.

5.21.1 Example(s)

The following example writes values to the rVariable `LATITUDE` of a CDF whose rVariables are 2-dimensional with dimension sizes `[360,181]`. For `LATITUDE` the record variance is `NOVARY`, the dimension variances are `[NOVARY,VARY]`, and the data type is `CDF_INT2`. This example is similar to the example in Section 5.19 except that it uses a single call to `CDFvarHyperPut` rather than numerous calls to `CDFvarPut`.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                         /* CDF identifier. */
CDFstatus   status;                     /* Returned status code. */
short       lat;                        /* Latitude value. */
short       lats[181];                  /* Buffer of latitude values. */
long        varN;                       /* rVariable number. */
long recStart = 0;                      /* Record number. */
long recCount = 1;                      /* Record counts. */
long recInterval = 1;                   /* Record interval. */
static long indices[2] = {0,0};         /* Dimension indices. */
static long counts[2] = {1,181};        /* Dimension counts. */
static long intervals[2] = {1,1};       /* Dimension intervals. */
.
.
varN = CDFvarNum (id, "LATITUDE");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), not an
                                            rVariable number but rather a
                                            warning/error code. */

for (lat = -90; lat <= 90; lat ++) lats[90+lat] = lat;

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

5.22 CDFvarHyperGet

```
CDFstatus CDFvarHyperGet(   /* out -- Completion status code. */
CDFid id,                   /* in  -- CDF identifier. */
long varNum,                /* in  -- rVariable number. */
long recStart,              /* in  -- Starting record number. */
long recCount,              /* in  -- Number of records. */
long recInterval,           /* in  -- Subsampling interval between records. */
long indices[],             /* in  -- Dimension indices of starting value. */
long counts[],              /* in  -- Number of values along each dimension. */
long intervals[],           /* in  -- Subsampling intervals along each
                                      dimension. */
void *buffer);              /* out -- Buffer of values. */
```

`CDFvarHyperGet` is used to read a buffer of one or more values from an rVariable. It is important to know the variable majority of the CDF before using `CDFvarHyperGet` because the values placed into the buffer will be in that majority. `CDFinquire` 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 arguments to `CDFvarHyperGet` are defined as follows:

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

`varNum` The number of the rVariable from which to read. This number may be determined with a call to `CDFvarNum` (see Section 5.16).

`recStart` The record number at which to start reading.

`recCount` The number of records to read.

`recInterval` The interval between records for subsampling (e.g., an interval of 2 means read every other record).

`indices` The indices (within each record) at which to start reading. Each element of `indices` specifies the corresponding dimension index. If there are zero (0) dimensions, 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. If there are zero (0) dimensions, this argument is ignored (but must be present).

`intervals` For each dimension, the interval between values for subsampling (e.g., an interval of 2 means read every other value). Each element of `intervals` specifies the corresponding dimension interval. If there are zero (0) dimensions, this argument is ignored (but must be present).

`buffer` The buffer of values read. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large to hold the values. `CDFvarInquire` would be used to determine the rVariable's data type and number of elements (of that data type) at each value. The values are read from the CDF and placed into memory starting at address `buffer`.

5.22.1 Example(s)

The following example will read an entire record of data from an rVariable. The CDF's rVariables are 3-dimensional with sizes [180,91,10] and CDF's variable majority is ROW_MAJOR. For the rVariable the record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF_REAL4. This example is similar to the example in Section 5.20 except that it uses a single call to CDFvarHyperGet rather than numerous calls to CDFvarGet.

```
.
.
#include "cdf.h"
.
.
CDFid       id;                          /* CDF identifier. */
CDFstatus   status;                      /* Returned status code. */
float       tmp[180][91][10];            /* Temperature values. */
long        varN;                        /* rVariable number. */
long recStart = 13;                      /* Record number. */
long recCount = 1;                       /* Record counts. */
long recInterval = 1;                    /* Record interval. */
static long indices[3] = {0,0,0};        /* Dimension indices. */
static long counts[3] = {180,91,10};     /* Dimension counts. */
static long intervals[3] = {1,1,1};      /* Dimension intervals. */
.
.
varN = CDFvarNum (id, "Temperature");
if (varN < 0) UserStatusHandler (varN);  /* If less than zero (0), then
                                            it is actually a warning/error
                                            code. */

status = CDFvarHyperGet (id, varN, recStart, recCount, recInterval,
                         indices, counts, intervals, tmp);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

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

5.23 CDFvarClose

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

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

The arguments to `CDFvarClose` are defined as follows:

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

`varNum` The number of the rVariable to close. This number may be determined with a call to `CDFvarNum` (see Section 5.16).

5.23.1 Example(s)

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

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

Chapter 6

Internal Interface — CDFlib

The Internal Interface consists of only one routine, CDFlib. CDFlib can be used to perform all possible operations on a CDF. In fact, all of the Standard Interface functions are implemented using the Internal Interface. CDFlib must be used to perform operations not possible with the Standard Interface functions. These operations would involve CDF features added after the Standard Interface functions had been defined (e.g., specifying a single-file format for a CDF, accessing zVariables, or specifying a pad value for an rVariable or zVariable). Note that CDFlib can also be used to perform certain operations more efficiently than with the Standard Interface functions.

CDFlib takes a variable number of arguments that specify one or more operations to be performed (e.g., opening a CDF, creating an attribute, or writing a variable value). The operations are performed according to the order of the arguments. Each operation consists of a function being performed on an item. An item may be either an object (e.g., a CDF, variable, or attribute) or a state (e.g., a CDF's format, a variable's data specification, or a CDF's current attribute). The possible functions and corresponding items (on which to perform those functions) are described in Section 6.6. The function prototype for CDFlib is as follows:

```
CDFstatus CDFlib (long function, ...);
```

This function prototype is found in the include file cdf.h.

6.1 Example(s)

The easiest way to explain how to use CDFlib would be to start with a few examples. The following example shows how a CDF would be created with the single-file format (assuming multi-file is the default).

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

```
CDFid       id;                           /* CDF identifier (handle). */
CDFstatus   status;                       /* Status returned from CDF
                                             library. */
static char CDFname[] = {"test1"};        /* File name of the CDF. */
long        numDims = 2;                  /* Number of dimensions. */
static long dimSizes[2] = {100,200};      /* Dimension sizes. */
long        encoding = HOST_ENCODING;     /* Data encoding. */
long        majority = ROW_MAJOR;         /* Variable data majority. */
long        format = SINGLE_FILE;         /* Format of CDF. */
.
.
status = CDFcreate (CDFname, numDims, dimSizes, encoding, majority, &id);
if (status != CDF_OK) UserStatusHandler (status);

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

The call to `CDFcreate` created the CDF as expected but with a format of multi-file (assuming that is the default). The call to `CDFlib` is then used to change the format to single-file (which must be done before any variables are created in the CDF).

The arguments to `CDFlib` in this example are explained as follows:

`PUT_` The first function to be performed. In this case an item is going to be put to the "current" CDF (a new format). `PUT_` is defined in `cdf.h` (as are all CDF constants). It was not necessary to select a current CDF since the call to `CDFcreate` implicitly selected the CDF created as the current CDF.[1] This is the case since all of the Standard Interface functions actually call the Internal Interface to perform their operations.

`CDF_FORMAT_` The item to be put. In this case it is the CDF's format.

`format` The actual format for the CDF. Depending on the item being put, one or more arguments would have been necessary. In this case only one argument is necessary.

`NULL_` This argument could have been one of two things. It could have been another item to put (followed by the arguments required for that item) or it could have been a new function to perform. In this case it is a new function to perform — the `NULL_` function. `NULL_` indicates the end of the call to `CDFlib`. Specifying `NULL_` at the end of the argument list is required because not all compilers/operating systems provide the ability for a called function to determine how many arguments were passed in by the calling function.

The next example shows how the same CDF could have been created using only one call to `CDFlib`. (The declarations would be the same.)

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

```
.
.
status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
                 PUT_, CDF_ENCODING_, encoding,
                       CDF_MAJORITY_, majority,
                       CDF_FORMAT_, format,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

The purpose of each argument is as follows:

`CREATE_` The first function to be performed. In this case something will be created.

`CDF_` The item to be created — a CDF in this case. There are four required arguments that must follow. When a CDF is created (with `CDFlib`), the format, encoding, and majority default to values specified when your CDF distribution was built and installed. Consult your system manager for these defaults.

`CDFname` The file name of the CDF.

`numDims` The number of dimensions in the CDF.

`dimSizes` The dimension sizes.

`id` The identifier to be used when referencing the created CDF in subsequent operations.

`PUT_` This argument could have been one of two things. Another item to create or a new function to perform. In this case it is another function to perform — something will be put to the CDF.

`CDF_ENCODING_` The item to be put — in this case the CDF's encoding. Note that the CDF did not have to be selected. It was implicitly selected as the current CDF when it was created.

`encoding` The encoding to be put to the CDF.

`CDF_MAJORITY_` This argument could have been one of two things. Another item to put or a new function to perform. In this case it is another item to put — the CDF's majority.

`majority` The majority to be put to the CDF.

`CDF_FORMAT_` Once again this argument could have been either another item to put or a new function to perform. It is another item to put — the CDF's format.

`format` The format to be put to the CDF.

`NULL_` This argument could have been either another item to put or a new function to perform. Here it is another function to perform — the `NULL_` function that ends the call to `CDFlib`.

Note that the operations are performed in the order that they appear in the argument list. The CDF had to be created before the encoding, majority, and format could be specified (put).

6.2 Current Objects/States (Items)

The use of `CDFlib` requires that an application be aware of the current objects/states maintained by the CDF library. The following current objects/states are used by the CDF library when performing operations.

CDF (object)

> A CDF operation is always performed on the current CDF. The current CDF is implicitly selected whenever a CDF is opened or created. The current CDF may be explicitly selected using the `<SELECT_,CDF_>`[2] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.[3]

rVariable (object)

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

zVariable (object)

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

attribute (object)

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

gEntry number (state)

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

rEntry number (state)

> A vAttribute rEntry operation is always performed on the current rEntry number in the current CDF for the current attribute in that CDF. For each open CDF a current rEntry number is maintained.

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

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

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

zEntry number (state)

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

record number, rVariables (state)

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

record count, rVariables (state)

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

record interval, rVariables (state)

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

dimension indices, rVariables (state)

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

dimension counts, rVariables (state)

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

dimension intervals, rVariables (state)

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

sequential value, rVariable (state)

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

record number, zVariable (state)

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

record count, zVariable (state)

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

record interval, zVariable (state)

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

dimension indices, zVariable (state)

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

dimension counts, zVariable (state)

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

dimension intervals, zVariable (state)

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

sequential value, zVariable (state)

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

status code (state)

When inquiring the explanation of a CDF status code, the text returned is always for the current status code. One current status code is maintained for the entire CDF library (regardless of the number of open CDFs). The current status code may be selected using the <SELECT_,CDF_STATUS_> operation. There is no default current status code. Note that the current status code is NOT the status code from the last operation performed.[4]

6.3 Returned Status

CDFlib returns a status code of type CDFstatus. Since more than one operation may be performed with a single call to CDFlib, the following rules apply:

1. The first error detected aborts the call to CDFlib, and the corresponding status code is returned.
2. In the absence of any errors, the status code for the last warning detected is returned.
3. In the absence of any errors or warnings, the status code for the last informational condition is returned.
4. In the absence of any errors, warnings, or informational conditions, CDF_OK is returned.

Chapter 7 explains how to interpret status codes. Appendix A lists the possible status codes and the type of each: error, warning, or informational.

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

6.4 Indentation/Style

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

```
   status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id,
                    PUT_, CDF_FORMAT_, format,
                          CDF_MAJORITY_, majority,
                    CREATE_, ATTR_, attrName, scope, &attrNum,
                             rVAR_, varName, dataType, numElements,
                                    recVary, dimVarys, &varNum,
                    NULL_);
```

Note that the functions (CREATE_, PUT_, and NULL_) are indented the same and that the items (CDF_, CDF_FORMAT_, CDF_MAJORITY_, ATTR_, and rVAR_) are indented the same under their corresponding functions.

The following example shows the same call to CDFlib without the proper indentation.

```
   status = CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, &id, PUT_,
                    CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
                    ATTR_, attrName, scope, &attrNum, rVAR_, varName, dataType,
                    numElements, recVary, dimVarys, &varNum, NULL_);
```

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

6.5 Syntax

CDFlib takes a variable number of arguments. There must always be at least one argument. The maximum number of arguments is not limited by CDF but rather the C compiler and operating system being used. Under normal circumstances that limit would never be reached (or even approached). Note also that a call to CDFlib with a large number of arguments can always be broken up into two or more calls to CDFlib with fewer arguments.

The syntax for CDFlib is as follows:

```
   status = CDFlib (fnc1, item1, arg1, arg2, ...argN,
                          item2, arg1, arg2, ...argN,
                          .
                          .
                          itemN, arg1, arg2, ...argN,
                    fnc2, item1, arg1, arg2, ...argN,
                          item2, arg1, arg2, ...argN,
                          .
                          .
                          itemN, arg1, arg2, ...argN,
.
.
.
fncN, item1, arg1, arg2, ...argN,
      item2, arg1, arg2, ...argN,
      .
      .
      itemN, arg1, arg2, ...argN,
NULL_);
```

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

6.6 Operations...

An operation consists of a function being performed on an item. The supported functions are as follows:

`CLOSE_`	Used to close an item.
`CONFIRM_`	Used to confirm the value of an item.
`CREATE_`	Used to create an item.
`DELETE_`	Used to delete an item.
`GET_`	Used to get (read) something from an item.
`NULL_`	Used to signal the end of the argument list of an internal interface call.
`OPEN_`	Used to open an item.
`PUT_`	Used to put (write) something to an item.
`SELECT_`	Used to select the value of an item.

For each function the supported items, required arguments, and required preselected objects/states are listed below. The required preselected objects/states are those objects/states that must be selected (typically with the `SELECT_` function) before a particular operation may be performed. Note that some of the required preselected objects/states have default values as described beginning on page 58.

`<CLOSE_,CDF_>`

Closes the current CDF. When the CDF is closed, there is no longer a current CDF. A CDF must be closed to ensure that it will be properly written to disk.

There are no required arguments.

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

`<CLOSE_,rVAR_>`

Closes the current rVariable (in the current CDF). This operation is only applicable to multi-file CDFs.

There are no required arguments.

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

`<CLOSE_,zVAR_>`

Closes the current zVariable (in the current CDF). This operation is only applicable to multi-file CDFs.

There are no required arguments.

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

`<CONFIRM_,ATTR_>`

Confirms the current attribute (in the current CDF). Required arguments are as follows:

out: `long *attrNum`

Attribute number.

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

`<CONFIRM_,ATTR_EXISTENCE_>`

Confirms the existence of the named attribute (in the current CDF). If the attribute does not exist, an error code will be returned. In any case the current attribute is not affected. Required arguments are as follows:

in: `char *attrName`

The attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<CONFIRM_,CDF_>`

Confirms the current CDF. Required arguments are as follows:

out: `CDFid *id`

The current CDF.

There are no required preselected objects/states.

`<CONFIRM_,CDF_ACCESS_>`

Confirms the accessability of the current CDF. If a fatal error occurred while accessing the CDF the error code `NO_MORE_ACCESS` will be returned. If this is the case, the CDF should still be closed.

There are no required arguments.

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

`<CONFIRM_,CDF_CACHESIZE_>`

Confirms the number of cache buffers being used for the dotCDF file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: `long *numBuffers`

The number of cache buffers being used.

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

`<CONFIRM_,CDF_DECODING_>`

Confirms the decoding for the current CDF. Required arguments are as follows:

out: `long *decoding`

The decoding. The decodings are described in Section 4.7.

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

`<CONFIRM_,CDF_NAME_>`

Confirms the file name of the current CDF. Required arguments are as follows:

out: `char CDFname[CDF_PATHNAME_LEN+1]`

File name of the CDF.

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

`<CONFIRM_,CDF_NEGtoPOSfp0_MODE_>`

Confirms the `-0.0` to `0.0` mode for the current CDF. Required arguments are as follows:

out: `long *mode`

The `-0.0` to `0.0` mode. The `-0.0` to `0.0` modes are described in Section 4.15.

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

`<CONFIRM_,CDF_READONLY_MODE_>`

Confirms the read-only mode for the current CDF. Required arguments are as follows:

out: `long *mode`

The read-only mode. The read-only modes are described in Section 4.13.

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

`<CONFIRM_,CDF_STATUS_>`

Confirms the current status code. Note that this is not the most recently returned status code but rather the most recently selected status code (see the `<SELECT_,CDF_STATUS_>` operation). Required arguments are as follows:

out: `CDFstatus *status`

The status code.

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

`<CONFIRM_,zMODE_>`

Confirms the zMode for the current CDF. Required arguments are as follows:

out: `long *mode`

The zMode. The zModes are described in Section 4.14.

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

<CONFIRM_,COMPRESS_CACHESIZE_>

Confirms the number of cache buffers being used for the compression scratch file file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: `long *numBuffers`

The number of cache buffers being used.

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

<CONFIRM_,CURgENTRY_EXISTENCE_>

Confirms the existence of the gEntry at the current gEntry number for the current attribute (in the current CDF). If the gEntry does not exist, an error code will be returned.

There are no required arguments.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<CONFIRM_,CURrENTRY_EXISTENCE_>

Confirms the existence of the rEntry at the current rEntry number for the current attribute (in the current CDF). If the rEntry does not exist, an error code will be returned.

There are no required arguments.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_,CURzENTRY_EXISTENCE_>

Confirms the existence of the zEntry at the current zEntry number for the current attribute (in the current CDF). If the zEntry does not exist, an error code will be returned.

There are no required arguments.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<CONFIRM_,gENTRY_>

Confirms the current gEntry number for all attributes in the current CDF. Required arguments are as follows:

out: `long *entryNum`

The gEntry number.

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

<CONFIRM_,gENTRY_EXISTENCE_>

Confirms the existence of the specified gEntry for the current attribute (in the current CDF). If the gEntry does not exist, an error code will be returned. In any case the current gEntry number is not affected. Required arguments are as follows:

in: `long entryNum`

The gEntry number.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<CONFIRM_,rENTRY_>`

Confirms the current rEntry number for all attributes in the current CDF. Required arguments are as follows:

out: `long *entryNum`

The rEntry number.

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

`<CONFIRM_,rENTRY_EXISTENCE_>`

Confirms the existence of the specified rEntry for the current attribute (in the current CDF). If the rEntry does not exist, an error code will be returned. In any case the current rEntry number is not affected. Required arguments are as follows:

in: `long entryNum`

The rEntry number.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<CONFIRM_,rVAR_>`

Confirms the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *varNum`

rVariable number.

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

`<CONFIRM_,rVAR_CACHESIZE_>`

Confirms the number of cache buffers being used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: `long *numBuffers`

The number of cache buffers being used.

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

<CONFIRM_,rVAR_EXISTENCE_>

Confirms the existence of the named rVariable (in the current CDF). If the rVariable does not exist, an error code will be returned. In any case the current rVariable is not affected. Required arguments are as follows:

in: `char *varName`

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

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

<CONFIRM_,rVAR_PADVALUE_>

Confirms the existence of an explicitly specified pad value for the current rVariable (in the current CDF). If an explicit pad value has not been specified, the informational status code `NO_PADVALUE_SPECIFIED_` will be returned.

There are no required arguments.

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

<CONFIRM_,rVAR_RESERVEPERCENT_>

Confirms the reserve percentage being used for the current rVariable (of the current CDF). This operation is only applicable to compressed rVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

out: `long *percent`

The reserve percentage.

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

<CONFIRM_,rVAR_SEQPOS_>

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

out: `long *recNum`

Record number.

out: `long indices[CDF_MAX_DIMS]`

Dimension indices. Each element of `indices` receives the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

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

<CONFIRM_,rVARs_DIMCOUNTS_>

Confirms the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long counts[CDF_MAX_DIMS]`

Dimension counts. Each element of `counts` receives the corresponding dimension count.

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

`<CONFIRM_,rVARs_DIMINDICES_>`

Confirms the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long indices[CDF_MAX_DIMS]`

Dimension indices. Each element of `indices` receives the corresponding dimension index.

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

`<CONFIRM_,rVARs_DIMINTERVALS_>`

Confirms the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long intervals[CDF_MAX_DIMS]`

Dimension intervals. Each element of `intervals` receives the corresponding dimension interval.

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

`<CONFIRM_,rVARs_RECCOUNT_>`

Confirms the current record count for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recCount`

Record count.

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

`<CONFIRM_,rVARs_RECINTERVAL_>`

Confirms the current record interval for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recInterval`

Record interval.

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

`<CONFIRM_,rVARs_RECNUMBER_>`

Confirms the current record number for all rVariables in the current CDF. Required arguments are as follows:

out: `long *recNum`

Record number.

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

`<CONFIRM_,STAGE_CACHESIZE_>`

Confirms the number of cache buffers being used for the staging scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: `long *numBuffers`

The number of cache buffers being used.

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

`<CONFIRM_,zENTRY_>`

Confirms the current zEntry number for all attributes in the current CDF. Required arguments are as follows:

out: `long *entryNum`

The zEntry number.

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

`<CONFIRM_,zENTRY_EXISTENCE_>`

Confirms the existence of the specified zEntry for the current attribute (in the current CDF). If the zEntry does not exist, an error code will be returned. In any case the current zEntry number is not affected. Required arguments are as follows:

in: `long entryNum`

The zEntry number.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<CONFIRM_,zVAR_>`

Confirms the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *varNum`

zVariable number.

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

`<CONFIRM_,zVAR_CACHESIZE_>`

Confirms the number of cache buffers being used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

out: `long *numBuffers`

The number of cache buffers being used.

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

`<CONFIRM_,zVAR_DIMCOUNTS_>`

Confirms the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long counts[CDF_MAX_DIMS]`

Dimension counts. Each element of `counts` receives the corresponding dimension count.

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

`<CONFIRM_,zVAR_DIMINDICES_>`

Confirms the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long indices[CDF_MAX_DIMS]`

Dimension indices. Each element of `indices` receives the corresponding dimension index.

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

`<CONFIRM_,zVAR_DIMINTERVALS_>`

Confirms the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long intervals[CDF_MAX_DIMS]`

Dimension intervals. Each element of `intervals` receives the corresponding dimension interval.

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

`<CONFIRM_,zVAR_EXISTENCE_>`

Confirms the existence of the named zVariable (in the current CDF). If the zVariable does not exist, an error code will be returned. In any case the current zVariable is not affected. Required arguments are as follows:

in: `char *varName`

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

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

`<CONFIRM_,zVAR_PADVALUE_>`

Confirms the existence of an explicitly specified pad value for the current zVariable (in the current CDF). If an explicit pad value has not been specified, the informational status code `NO_PADVALUE_SPECIFIED_` will be returned.

There are no required arguments.

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

`<CONFIRM_,zVAR_RECCOUNT_>`

Confirms the current record count for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recCount`

Record count.

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

`<CONFIRM_,zVAR_RECINTERVAL_>`

Confirms the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recInterval`

Record interval.

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

`<CONFIRM_,zVAR_RECNUMBER_>`

Confirms the current record number for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *recNum`

Record number.

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

`<CONFIRM_,zVAR_RESERVEPERCENT_>`

Confirms the reserve percentage being used for the current zVariable (of the current CDF). This operation is only applicable to compressed zVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

out: `long *percent`

The reserve percentage.

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

`<CONFIRM_,zVAR_SEQPOS_>`

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

out: `long *recNum`

Record number.

out: `long indices[CDF_MAX_DIMS]`

Dimension indices. Each element of `indices` receives the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but must be present).

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

`<CREATE_,ATTR_>`

A new attribute will be created in the current CDF. An attribute with the same name must not already exist in the CDF. The created attribute implicitly becomes the current attribute (in the current CDF). Required arguments are as follows:

in: `char *attrName`

Name of the attribute to be created. This can be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator). Attribute names are case-sensitive.

in: `long scope`

Scope of the new attribute. Specify one of the scopes described in Section 4.12.

out: `long *attrNum`

Number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may also be determined with the `<GET_,ATTR_NUMBER_>` operation.

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

`<CREATE_,CDF_>`

A new CDF will be created. It is illegal to create a CDF that already exists. The created CDF implicitly becomes the current CDF. Required arguments are as follows:

in: `char *CDFname`

File name of the CDF to be created. (Do not append an extension.) This can be at most `CDF_PATHNAME_LEN` characters (excluding the `NUL` terminator). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

UNIX: File names are case-sensitive.

in: `long numDims`

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

in: `long dimSizes[]`

Dimension sizes for the rVariables. Each element of `dimSizes` specifies the corresponding dimension size. Each dimension size must be greater than zero (0). For 0-dimensional rVariables this argument is ignored (but must be present). Note that this must be specified even if the CDF will contain only zVariables.

out: `CDFid *id`

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

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

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

There are no required preselected objects/states.

<CREATE_,rVAR_>

A new rVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created rVariable implicitly becomes the current rVariable (in the current CDF). Required arguments are as follows:

in: `char *varName`

Name of the rVariable to be created. This can be at most CDF_VAR_NAME_LEN characters (excluding the NUL). Variable names are case-sensitive.

in: `long dataType`

Data type of the new rVariable. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in each string (an array of characters). A string exists at each value of the variable. For the non-character data types this must be one (1) — multiple elements are not allowed for non-character data types.

in: `long recVary`

Record variance. Specify one of the variances described in Section 4.9.

in: `long dimVarys[]`

Dimension variances. Each element of dimVarys specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9. For 0-dimensional rVariables this argument is ignored (but must be present).

out: `long *varNum`

Number assigned to the new rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariable's number may also be determined with the <GET_,rVAR_NUMBER_> operation.

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

<CREATE_,zVAR_>

A new zVariable will be created in the current CDF. A variable (rVariable or zVariable) with the same name must not already exist in the CDF. The created zVariable implicitly becomes the current zVariable (in the current CDF). Required arguments are as follows:

in: `char *varName`

Name of the zVariable to be created. This can be at most CDF_VAR_NAME_LEN characters (excluding the NUL terminator). Variable names are case-sensitive.

in: `long dataType`

Data type of the new zVariable. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in each string (an array of characters).

A string exists at each value of the variable. For the non-character data types this must be one (1) — multiple elements are not allowed for non-character data types.

in: `long numDims`

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

in: `long dimSizes[]`

The dimension sizes. Each element of `dimSizes` specifies the corresponding dimension size. Each dimension size must be greater than zero (0). For a 0-dimensional zVariable this argument is ignored (but must be present).

in: `long recVary`

Record variance. Specify one of the variances described in Section 4.9.

in: `long dimVarys[]`

Dimension variances. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9. For a 0-dimensional zVariable this argument is ignored (but must be present).

out: `long *varNum`

Number assigned to the new zVariable. This number must be used in subsequent CDF function calls when referring to this zVariable. An existing zVariable's number may also be determined with the `<GET_,zVAR_NUMBER_>` operation.

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

`<DELETE_,ATTR_>`

Deletes the current attribute (in the current CDF). Note that the attribute's entries are also deleted. The attributes which numerically follow the attribute being deleted are immediately renumbered. When the attribute is deleted, there is no longer a current attribute.

There are no required arguments.

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

`<DELETE_,CDF_>`

Deletes the current CDF. A CDF must be opened before it can be deleted. When the CDF is deleted, there is no longer a current CDF.

There are no required arguments.

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

`<DELETE_,gENTRY_>`

Deletes the gEntry at the current gEntry number of the current attribute (in the current CDF). Note that this does not affect the current gEntry number.

There are no required arguments.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<DELETE_,rENTRY_>`

Deletes the rEntry at the current rEntry number of the current attribute (in the current CDF). Note that this does not affect the current rEntry number.

There are no required arguments.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<DELETE_,rVAR_>`

Deletes the current rVariable (in the current CDF). Note that the rVariable's corresponding rEntries are also deleted (from each vAttribute). The rVariables which numerically follow the rVariable being deleted are immediately renumbered. The rEntries which numerically follow the rEntries being deleted are also immediately renumbered. When the rVariable is deleted, there is no longer a current rVariable. **NOTE:** This operation is only allowed on single-file CDFs.

There are no required arguments.

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

`<DELETE_,rVAR_RECORDS_>`

Deletes the specified range of records from the current rVariable (in the current CDF). If the rVariable has sparse records a gap of missing records will be created. If the rVariable does not have sparse records, the records following the range of deleted records are immediately renumbered beginning with the number of the first deleted record. **NOTE:** This operation is only allowed on single-file CDFs. Required arguments are as follows:

in: `long firstRecord`

The record number of the first record to be deleted.

in: `long lastRecord`

The record number of the last record to be deleted.

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

`<DELETE_,zENTRY_>`

Deletes the zEntry at the current zEntry number of the current attribute (in the current CDF). Note that this does not affect the current zEntry number.

There are no required arguments.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<DELETE_,zVAR_>`

Deletes the current zVariable (in the current CDF). Note that the zVariable's corresponding zEntries are also deleted (from each vAttribute). The zVariables which numerically follow the zVariable being deleted are immediately renumbered. The rEntries which numerically follow the rEntries being deleted are also immediately renumbered. When the zVariable is deleted, there is no longer a current zVariable. **NOTE:** This operation is only allowed on single-file CDFs.

There are no required arguments.

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

<DELETE_,zVAR_RECORDS_>

Deletes the specified range of records from the current zVariable (in the current CDF). If the zVariable has sparse records a gap of missing records will be created. If the zVariable does not have sparse records, the records following the range of deleted records are immediately renumbered beginning with the number of the first deleted record. **NOTE:** This operation is only allowed on single-file CDFs. Required arguments are as follows:

in: `long firstRecord`

The record number of the first record to be deleted.

in: `long lastRecord`

The record number of the last record to be deleted.

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

<GET_,ATTR_MAXgENTRY_>

Inquires the maximum gEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of gEntries for the attribute. Required arguments are as follows:

out: `long *maxEntry`

The maximum gEntry number for the attribute. If no gEntries exist, then a value of `-1` will be passed back.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<GET_,ATTR_MAXrENTRY_>

Inquires the maximum rEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of rEntries for the attribute. Required arguments are as follows:

out: `long *maxEntry`

The maximum rEntry number for the attribute. If no rEntries exist, then a value of `-1` will be passed back.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

<GET_,ATTR_MAXzENTRY_>

Inquires the maximum zEntry number used for the current attribute (in the current CDF). This does not necessarily correspond with the number of zEntries for the attribute. Required arguments are as follows:

out: `long *maxEntry`

The maximum zEntry number for the attribute. If no zEntries exist, then a value of `-1` will be passed back.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,ATTR_NAME_>`

Inquires the name of the current attribute (in the current CDF). Required arguments are as follows:

out: `char attrName[CDF_ATTR_NAME_LEN+1]`

Attribute name.

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

`<GET_,ATTR_NUMBER_>`

Gets the number of the named attribute (in the current CDF). Note that this operation does not select the current attribute. Required arguments are as follows:

in: `char *attrName`

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

out: `long *attrNum`

The attribute number.

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

`<GET_,ATTR_NUMgENTRIES_>`

Inquires the number of gEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum gEntry number used. Required arguments are as follows:

out: `long *numEntries`

The number of gEntries for the attribute.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<GET_,ATTR_NUMrENTRIES_>`

Inquires the number of rEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum rEntry number used. Required arguments are as follows:

out: `long *numEntries`

The number of rEntries for the attribute.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,ATTR_NUMzENTRIES_>`

Inquires the number of zEntries for the current attribute (in the current CDF). This does not necessarily correspond with the maximum zEntry number used. Required arguments are as follows:

out: `long *numEntries`

The number of zEntries for the attribute.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,ATTR_SCOPE_>`

Inquires the scope of the current attribute (in the current CDF). Required arguments are as follows:

out: `long *scope`

Attribute scope. The scopes are described in Section 4.12.

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

`<GET_,CDF_COMPRESSION_>`

Inquires the compression type/parameters of the current CDF. This refers to the compression of the CDF — not of any compressed variables. Required arguments are as follows:

out: `long *cType`

The compression type. The types of compressions are described in Section 4.10.

out: `long cParms[CDF_MAX_PARMS]`

The compression parameters. The compression parameters are described in Section 4.10.

out: `long *cPct`

If compressed, the percentage of the uncompressed size of the CDF needed to store the compressed CDF.

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

`<GET_,CDF_COPYRIGHT_>`

Reads the copyright notice for the CDF library that created the current CDF. Required arguments are as follows:

out: `char copyRight[CDF_COPYRIGHT_LEN+1`

CDF copyright text.

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

`<GET_,CDF_ENCODING_>`

Inquires the data encoding of the current CDF. Required arguments are as follows:

out: `long *encoding`

Data encoding. The encodings are described in Section 4.6.

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

<GET_,CDF_FORMAT_>

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

out: `long *format`

CDF format. The formats are described in Section 4.4.

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

<GET_,CDF_INCREMENT_>

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

out: `long *increment`

Incremental number.

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

<GET_,CDF_INFO_>

Inquires the compression type/parameters of a CDF without having to open the CDF. This refers to the compression of the CDF — not of any compressed variables. Required arguments are as follows:

in: `char *CDFname`

File name of the CDF to be inquired. (Do not append an extension.) This can be at most `CDF_PATHNAME_LEN` characters (excluding the `NUL` terminator). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

UNIX: File names are case-sensitive.

out: `long *cType`

The CDF compression type. The types of compressions are described in Section 4.10.

out: `long cParms[CDF_MAX_PARMS]`

The compression parameters. The compression parameters are described in Section 4.10.

out: `long *cSize`

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

out: `long *uSize`

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

There are no required preselected objects/states.

<GET_,CDF_MAJORITY_>

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

out: `long *majority`

Variable majority. The majorities are described in Section 4.8.

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

`<GET_,CDF_NUMATTRS_>`

Inquires the number of attributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of attributes.

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

`<GET_,CDF_NUMgATTRS_>`

Inquires the number of gAttributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of gAttributes.

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

`<GET_,CDF_NUMrVARS_>`

Inquires the number of rVariables in the current CDF. Required arguments are as follows:

out: `long *numVars`

Number of rVariables.

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

`<GET_,CDF_NUMvATTRS_>`

Inquires the number of vAttributes in the current CDF. Required arguments are as follows:

out: `long *numAttrs`

Number of vAttributes.

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

`<GET_,CDF_NUMzVARS_>`

Inquires the number of zVariables in the current CDF. Required arguments are as follows:

out: `long *numVars`

Number of zVariables.

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

`<GET_,CDF_RELEASE_>`

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

out: `long *release`

Release number.

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

`<GET_,CDF_VERSION_>`

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

out: `long *version`

Version number.

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

`<GET_,DATATYPE_SIZE_>`

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

in: `long dataType`

Data type.

out: `long *numBytes`

Number of bytes per element.

There are no required preselected objects/states.

`<GET_,gENTRY_DATA_>`

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

out: `void *value`

Value. This buffer must be large to hold the value. The value is read from the CDF and placed into memory at address `value`.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<GET_,gENTRY_DATATYPE_>`

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

out: `long *dataType`

Data type. The data types are described in Section 4.5.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<GET_,gENTRY_NUMELEMS_>`

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

out: `long *numElements`

Number of elements of the data type. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<GET_,LIB_COPYRIGHT_>`

Reads the copyright notice of the CDF library being used. Required arguments are as follows:

out: `char copyRight[CDF_COPYRIGHT_LEN+1`

CDF library copyright text.

There are no required preselected objects/states.

`<GET_,LIB_INCREMENT_>`

Inquires the incremental number of the CDF library being used. Required arguments are as follows:

out: `long *increment`

Incremental number.

There are no required preselected objects/states.

`<GET_,LIB_RELEASE_>`

Inquires the release number of the CDF library being used. Required arguments are as follows:

out: `long *release`

Release number.

There are no required preselected objects/states.

`<GET_,LIB_subINCREMENT_>`

Inquires the subincremental character of the CDF library being used. Required arguments are as follows:

out: `char *subincrement`

Subincremental character.

There are no required preselected objects/states.

`<GET_,LIB_VERSION_>`

Inquires the version number of the CDF library being used. Required arguments are as follows:

out: `long *version`

Version number.

There are no required preselected objects/states.

`<GET_,rENTRY_DATA_>`

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

out: `void *value`

Value. This buffer must be large to hold the value. The value is read from the CDF and placed into memory at address `value`.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,rENTRY_DATATYPE_>`

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

out: `long *dataType`

Data type. The data types are described in Section 4.5.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,rENTRY_NUMELEMS_>`

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

out: `long *numElements`

Number of elements of the data type. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,rVAR_ALLOCATEDFROM_>`

Inquires the next allocated record at or after a given record for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long startRecord`

The record number at which to begin searching for the next allocated record. If this record exists, it will be considered the next allocated record.

out: `long *nextRecord`

The number of the next allocated record.

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

`<GET_,rVAR_ALLOCATEDTO_>`

Inquires the last allocated record (before the next unallocated record) at or after a given record for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long startRecord`

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

out: `long *nextRecord`

The number of the last allocated record.

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

`<GET_,rVAR_BLOCKINGFACTOR_>`[5]

Inquires the blocking factor for the current rVariable (in the current CDF). Blocking factors are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

out: `long *blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor is being used.

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

`<GET_,rVAR_COMPRESSION_>`

Inquires the compression type/parameters of the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *cType`

The compression type. The types of compressions are described in Section 4.10.

out: `long cParms[CDF_MAX_PARMS]`

The compression parameters. The compression parameters are described in Section 4.10.

out: `long *cPct`

If compressed, the percentage of the uncompressed size of the rVariable's data values needed to store the compressed values.

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

`<GET_,rVAR_DATA_>`

Reads a value from the current rVariable (in the current CDF). The value is read at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments are as follows:

[5] The item `rVAR_BLOCKINGFACTOR_` was previously named `rVAR_EXTENDRECS_`.

out: `void *value`

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`<GET_,rVAR_DATATYPE_>`

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

out: `long *dataType`

Data type. The data types are described in Section 4.5.

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

`<GET_,rVAR_DIMVARYS_>`

Inquires the dimension variances of the current rVariable (in the current CDF). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long dimVarys[CDF_MAX_DIMS]`

Dimension variances. Each element of `dimVarys` receives the corresponding dimension variance. The variances are described in Section 4.9.

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

`<GET_,rVAR_HYPERDATA_>`

Reads one or more values from the current rVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

out: `void *buffer`

Values. This buffer must be large enough to hold the values. The values are read from the CDF and placed into memory starting at address `buffer`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`<GET_,rVAR_MAXallocREC_>`

Inquires the maximum record number allocated for the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *varMaxRecAlloc`

Maximum record number allocated.

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

`<GET_,rVAR_MAXREC_>`

Inquires the maximum record number for the current rVariable (in the current CDF). For rVariables with a record variance of NOVARY, this will be at most zero (0). A value of negative one (-1) indicates that no records have been written. Required arguments are as follows:

out: `long *varMaxRec`

Maximum record number.

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

`<GET_,rVAR_NAME_>`

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

out: `char varName[CDF_VAR_NAME_LEN+1`

Name of the rVariable.

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

`<GET_,rVAR_nINDEXENTRIES_>`

Inquires the number of index entries for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numEntries`

Number of index entries.

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

`<GET_,rVAR_nINDEXLEVELS_>`

Inquires the number of index levels for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numLevels`

Number of index levels.

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

`<GET_,rVAR_nINDEXRECORDS_>`

Inquires the number of index records for the current rVariable (in the current CDF). This only has significance for rVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of index records.

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

`<GET_,rVAR_NUMallocRECS_>`

Inquires the number of records allocated for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes the allocation of variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of allocated records.

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

`<GET_,rVAR_NUMBER_>`

Gets the number of the named rVariable (in the current CDF). Note that this operation does not select the current rVariable. Required arguments are as follows:

in: `char *varName`

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

out: `long *varNum`

The rVariable number.

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

`<GET_,rVAR_NUMELEMS_>`

Inquires the number of elements (of the data type) for the current rVariable (in the current CDF). Required arguments are as follows:

out: `long *numElements`

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

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

`<GET_,rVAR_NUMRECS_>`

Inquires the number of records written for the current rVariable (in the current CDF). This may not correspond to the maximum record written (see `<GET_,rVAR_MAXREC_>`) if the rVariable has sparse records. Required arguments are as follows:

out: `long *numRecords`

Number of records written.

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

`<GET_,rVAR_PADVALUE_>`

Inquires the pad value of the current rVariable (in the current CDF). If a pad value has not been explicitly specified for the rVariable (see `<PUT_,rVAR_PADVALUE_>`), the informational status code `NO_PADVALUE_SPECIFIED` will be returned and the default pad value for the rVariable's data type will be placed in the pad value buffer provided. Required arguments are as follows:

out: void *value

Pad value. This buffer must be large enough to hold the pad value. The pad value is read from the CDF and placed in memory at address value.

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

<GET_,rVAR_RECVARY_>

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

out: long *recVary

Record variance. The variances are described in Section 4.9.

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

<GET_,rVAR_SEQDATA_>

Reads one value from the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the rVariable. Required arguments are as follows:

out: void *value

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address value.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are read.

<GET_,rVAR_SPARSEARRAYS_>

Inquires the sparse arrays type/parameters of the current rVariable (in the current CDF). Required arguments are as follows:

out: long *sArraysType

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

out: long sArraysParms[CDF_MAX_PARMS]

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

out: long *sArraysPct

If sparse arrays, the percentage of the non-sparse size of the rVariable's data values needed to store the sparse values.

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

<GET_,rVAR_SPARSERECORDS_>

Inquires the sparse records type of the current rVariable (in the current CDF). Required arguments are as follows:

out: long *sRecordsType

The sparse records type. The types of sparse records are described in Section 4.11.

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

<GET_,rVARs_DIMSIZES_>

Inquires the size of each dimension for the rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

out: `long dimSizes[CDF_MAX_DIMS]`

Dimension sizes. Each element of `dimSizes` receives the corresponding dimension size.

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

<GET_,rVARs_MAXREC_>

Inquires the maximum record number of the rVariables in the current CDF. Note that this is not the number of records but rather the maximum record number (which is one less than the number of records). A value of negative one (-1) indicates that the rVariables contain no records. The maximum record number for an individual rVariable may be inquired using the <GET_,rVAR_MAXREC_> operation. Required arguments are as follows:

out: `long *maxRec`

Maximum record number.

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

<GET_,rVARs_NUMDIMS_>

Inquires the number of dimensions for the rVariables in the current CDF. Required arguments are as follows:

out: `long *numDims`

Number of dimensions.

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

<GET_,rVARs_RECDATA_>

Reads full-physical records from one or more rVariables (in the current CDF). The full-physical records are read at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

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

in: `long varNums[]`

The rVariables from which to read. This array, whose size is determined by the value of `numVars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `void *buffer`

The buffer into which the full-physical rVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. The order of the full-physical rVariable records in this buffer will correspond to the rVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical rVariable records. Be careful if using C `struct` objects to receive multiple full-physical rVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to allocate this buffer.

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

`<GET_,STATUS_TEXT_>`

Inquires the explanation text for the current status code. Note that the current status code is NOT the status from the last operation performed. Required arguments are as follows:

out: `char text[CDF_STATUSTEXT_LEN+1`

Text explaining the status code.

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

`<GET_,zENTRY_DATA_>`

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

out: `void *value`

Value. This buffer must be large to hold the value. The value is read from the CDF and placed into memory at address `value`.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,zENTRY_DATATYPE_>`

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

out: `long *dataType`

Data type. The data types are described in Section 4.5.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,zENTRY_NUMELEMS_>`

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

out: `long *numElements`

Number of elements of the data type. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<GET_,zVAR_ALLOCATEDFROM_>`

Inquires the next allocated record at or after a given record for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long startRecord`

The record number at which to begin searching for the next allocated record. If this record exists, it will be considered the next allocated record.

out: `long *nextRecord`

The number of the next allocated record.

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

`<GET_,zVAR_ALLOCATEDTO_>`

Inquires the last allocated record (before the next unallocated record) at or after a given record for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long startRecord`

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

out: `long *nextRecord`

The number of the last allocated record.

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

`<GET_,zVAR_BLOCKINGFACTOR_>`[6]

Inquires the blocking factor for the current zVariable (in the current CDF). Blocking factors are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

out: `long *blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor is being used.

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

`<GET_,zVAR_COMPRESSION_>`

Inquires the compression type/parameters of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *cType`

The compression type. The types of compressions are described in Section 4.10.

[6] The item `zVAR_BLOCKINGFACTOR_` was previously named `zVAR_EXTENDRECS_`.

out: `long cParms[CDF_MAX_PARMS]`

The compression parameters. The compression parameters are described in Section 4.10.

out: `long *cPct`

If compressed, the percentage of the uncompressed size of the zVariable's data values needed to store the compressed values.

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

`<GET_,zVAR_DATA_>`

Reads a value from the current zVariable (in the current CDF). The value is read at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

out: `void *value`

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`<GET_,zVAR_DATATYPE_>`

Inquires the data type of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *dataType`

Data type. The data types are described in Section 4.5.

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

`<GET_,zVAR_DIMSIZES_>`

Inquires the size of each dimension for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long dimSizes[CDF_MAX_DIMS]`

Dimension sizes. Each element of `dimSizes` receives the corresponding dimension size.

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

`<GET_,zVAR_DIMVARYS_>`

Inquires the dimension variances of the current zVariable (in the current CDF). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

out: `long dimVarys[CDF_MAX_DIMS]`

Dimension variances. Each element of `dimVarys` receives the corresponding dimension variance. The variances are described in Section 4.9.

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

`<GET_,zVAR_HYPERDATA_>`

Reads one or more values from the current zVariable (in the current CDF). The values are read based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

out: `void *buffer`

Values. This buffer must be large enough to hold the values. The values are read from the CDF and placed into memory starting at address `buffer`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`<GET_,zVAR_MAXallocREC_>`

Inquires the maximum record number allocated for the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *varMaxRecAlloc`

Maximum record number allocated.

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

`<GET_,zVAR_MAXREC_>`

Inquires the maximum record number for the current zVariable (in the current CDF). For zVariables with a record variance of `NOVARY`, this will be at most zero (0). A value of negative one (-1) indicates that no records have been written. Required arguments are as follows:

out: `long *varMaxRec`

Maximum record number.

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

`<GET_,zVAR_NAME_>`

Inquires the name of the current zVariable (in the current CDF). Required arguments are as follows:

out: `char varName[CDF_VAR_NAME_LEN+1`

Name of the zVariable.

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

`<GET_,zVAR_nINDEXENTRIES_>`

Inquires the number of index entries for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numEntries`

Number of index entries.

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

<GET_,zVAR_nINDEXLEVELS_>

Inquires the number of index levels for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numLevels`

Number of index levels.

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

<GET_,zVAR_nINDEXRECORDS_>

Inquires the number of index records for the current zVariable (in the current CDF). This only has significance for zVariables that are in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the indexing scheme used for variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of index records.

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

<GET_,zVAR_NUMallocRECS_>

Inquires the number of records allocated for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes the allocation of variable records in a single-file CDF. Required arguments are as follows:

out: `long *numRecords`

Number of allocated records.

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

<GET_,zVAR_NUMBER_>

Gets the number of the named zVariable (in the current CDF). Note that this operation does not select the current zVariable. Required arguments are as follows:

in: `char *varName`

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

out: `long *varNum`

The zVariable number.

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

<GET_,zVAR_NUMDIMS_>

Inquires the number of dimensions for the current zVariable in the current CDF. Required arguments are as follows:

out: `long *numDims`

Number of dimensions.

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

`<GET_,zVAR_NUMELEMS_>`

Inquires the number of elements (of the data type) for the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *numElements`

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

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

`<GET_,zVAR_NUMRECS_>`

Inquires the number of records written for the current zVariable (in the current CDF). This may not correspond to the maximum record written (see `<GET_,zVAR_MAXREC_>`) if the zVariable has sparse records. Required arguments are as follows:

out: `long *numRecords`

Number of records written.

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

`<GET_,zVAR_PADVALUE_>`

Inquires the pad value of the current zVariable (in the current CDF). If a pad value has not been explicitly specified for the zVariable (see `<PUT_,zVAR_PADVALUE_>`), the informational status code `NO_PADVALUE_SPECIFIED` will be returned and the default pad value for the zVariable's data type will be placed in the pad value buffer provided. Required arguments are as follows:

out: `void *value`

Pad value. This buffer must be large enough to hold the pad value. The pad value is read from the CDF and placed in memory at address `value`.

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

`<GET_,zVAR_RECVARY_>`

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

out: `long *recVary`

Record variance. The variances are described in Section 4.9.

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

`<GET_,zVAR_SEQDATA_>`

Reads one value from the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the zVariable. Required arguments are as follows:

out: `void *value`

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are read.

`<GET_,zVAR_SPARSEARRAYS_>`

Inquires the sparse arrays type/parameters of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *sArraysType`

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

out: `long sArraysParms[CDF_MAX_PARMS]`

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

out: `long *sArraysPct`

If sparse arrays, the percentage of the non-sparse size of the zVariable's data values needed to store the sparse values.

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

`<GET_,zVAR_SPARSERECORDS_>`

Inquires the sparse records type of the current zVariable (in the current CDF). Required arguments are as follows:

out: `long *sRecordsType`

The sparse records type. The types of sparse records are described in Section 4.11.

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

`<GET_,zVARs_MAXREC_>`

Inquires the maximum record number of the zVariables in the current CDF. Note that this is not the number of records but rather the maximum record number (which is one less than the number of records). A value of negative one (-1) indicates that the zVariables contain no records. The maximum record number for an individual zVariable may be inquired using the `<GET_,zVAR_MAXREC_>` operation. Required arguments are as follows:

out: `long *maxRec`

Maximum record number.

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

`<GET_,zVARs_RECDATA_>`

Reads full-physical records from one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is read at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

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

in: `long varNums[]`

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

in: `void *buffer`

The buffer into which the full-physical zVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. The order of the full-physical zVariable records in this buffer will correspond to the zVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to receive multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to allocate this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`<NULL_>`

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

`<OPEN_,CDF_>`

Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: `char *CDFname`

File name of the CDF to be opened. (Do not append an extension.) This can be at most `CDF_PATHNAME_LEN` characters (excluding the `NUL` terminator). A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

UNIX: File names are case-sensitive.

out: `CDFid *id`

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

There are no required preselected objects/states.

<PUT_,ATTR_NAME_>

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

in: `char *attrName`

New attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

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

<PUT_,ATTR_SCOPE_>

Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

in: `long scope`

New attribute scope. Specify one of the scopes described in Section 4.12.

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

<PUT_,CDF_COMPRESSION_>

Specifies the compression type/parameters for the current CDF. This refers to the compression of the CDF — not of any variables. Required arguments are as follows:

in: `long cType`

The compression type. The types of compressions are described in Section 4.10.

in: `long cParms[]`

The compression parameters. The compression parameters are described in Section 4.10.

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

<PUT_,CDF_ENCODING_>

Respecifies the data encoding of the current CDF. A CDF's data encoding may not be changed after any variable values (including the pad value) or attribute entries have been written. Required arguments are as follows:

in: `long encoding`

New data encoding. Specify one of the encodings described in Section 4.6.

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

<PUT_,CDF_FORMAT_>

Respecifies the format of the current CDF. A CDF's format may not be changed after any variables have been created. Required arguments are as follows:

in: `long format`

New CDF format. Specify one of the formats described in Section 4.4.

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

<PUT_,CDF_MAJORITY_>

Respecifies the variable majority of the current CDF. A CDF's variable majority may not be changed after any variable values have been written. Required arguments are as follows:

in: `long majority`

New variable majority. Specify one of the majorities described in Section 4.8.

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

<PUT_,gENTRY_DATA_>

Writes a gEntry to the current attribute at the current gEntry number (in the current CDF). An existing gEntry may be overwritten with a new gEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the gEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

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

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<PUT_,gENTRY_DATASPEC_>

Modifies the data specification (data type and number of elements) of the gEntry at the current gEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the gEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

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

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<PUT_,rENTRY_DATA_>

Writes an rEntry to the current attribute at the current rEntry number (in the current CDF). An existing rEntry may be overwritten with a new rEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the rEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

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

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,rENTRY_DATASPEC_>`

Modifies the data specification (data type and number of elements) of the rEntry at the current rEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the rEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,rVAR_ALLOCATEBLOCK_>`

Specifies a range of records to allocate for the current rVariable (in the current CDF). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long firstRecord`

The first record number to allocate.

in: `long lastRecord`

The last record number to allocate.

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

<PUT_,rVAR_ALLOCATERECS_>

Specifies the number of records to allocate for the current rVariable (in the current CDF). The records are allocated beginning at record number 0 (zero). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long nRecords`

Number of records to allocate.

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

<PUT_,rVAR_BLOCKINGFACTOR_>[7]

Specifies the blocking factor for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

in: `long blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

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

<PUT_,rVAR_COMPRESSION_>

Specifies the compression type/parameters for the current rVariable (in current CDF). Required arguments are as follows:

in: `long cType`

The compression type. The types of compressions are described in Section 4.10.

in: `long cParms[]`

The compression parameters. The compression parameters are described in Section 4.10.

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

<PUT_,rVAR_DATA_>

Writes one value to the current rVariable (in the current CDF). The value is written at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

<PUT_,rVAR_DATASPEC_>

Respecifies the data specification (data type and number of elements) of the current rVariable (in the current CDF). An rVariable's data specification may not be changed if the new data

[7] The item `rVAR_BLOCKINGFACTOR_` was previously named `rVAR_EXTENDRECS_`.

specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (an array of characters). A string exists at each value. For the non-character data types this must be one (1) — arrays of values are not allowed for non-character data types.

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

`<PUT_,rVAR_DIMVARYS_>`

Respecifies the dimension variances of the current rVariable (in the current CDF). An rVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

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

`<PUT_,rVAR_HYPERDATA_>`

Writes one or more values to the current rVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

`<PUT_,rVAR_INITIALRECS_>`

Specifies the number of records to initially write to the current rVariable (in the current CDF). The records are written beginning at record number `0` (zero). This may be specified only once per rVariable and before any other records have been written to that rVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: `long nRecords`

Number of records to write.

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

<PUT_,rVAR_NAME_>

Renames the current rVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the rVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

<PUT_,rVAR_PADVALUE_>

Specifies the pad value for the current rVariable (in the current CDF). An rVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

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

<PUT_,rVAR_RECVARY_>

Respecifies the record variance of the current rVariable (in the current CDF). An rVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.9.

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

<PUT_,rVAR_SEQDATA_>

Writes one value to the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the rVariable, the rVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are written.

<PUT_,rVAR_SPARSEARRAYS_>

Specifies the sparse arrays type/parameters for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long sArraysType`

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

in: `long sArraysParms[]`

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

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

`<PUT_,rVAR_SPARSERECORDS_>`

Specifies the sparse records type for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long sRecordsType`

The sparse records type. The types of sparse records are described in Section 4.11.

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

`<PUT_,rVARs_RECDATA_>`

Writes full-physical records to one or more rVariables (in the current CDF). The full-physical records are written at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

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

in: `long varNums[]`

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

in: `void *buffer`

The buffer of full-physical rVariable records to be written. The order of the full-physical rVariable records in this buffer must agree with the rVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical rVariable records. Be careful if using C `struct` objects to store multiple full-physical rVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `sturct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to create this buffer.

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

`<PUT_,zENTRY_DATA_>`

Writes a zEntry to the current attribute at the current zEntry number (in the current CDF). An existing zEntry may be overwritten with a new zEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the zEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

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

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,zENTRY_DATASPEC_>`

Modifies the data specification (data type and number of elements) of the zEntry at the current zEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the zEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

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

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,zVAR_ALLOCATEBLOCK_>`

Specifies a range of records to allocate for the current zVariable (in the current CDF). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long firstRecord`

The first record number to allocate.

in: `long lastRecord`

The last record number to allocate.

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

`<PUT_,zVAR_ALLOCATERECS_>`

Specifies the number of records to allocate for the current zVariable (in the current CDF). The records are allocated beginning at record number `0` (zero). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long nRecords`

Number of records to allocate.

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

`<PUT_,zVAR_BLOCKINGFACTOR_>`[8]

Specifies the blocking factor for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

in: `long blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

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

`<PUT_,zVAR_COMPRESSION_>`

Specifies the compression type/parameters for the current zVariable (in current CDF). Required arguments are as follows:

in: `long cType`

The compression type. The types of compressions are described in Section 4.10.

in: `long cParms[]`

The compression parameters. The compression parameters are described in Section 4.10.

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

`<PUT_,zVAR_DATA_>`

Writes one value to the current zVariable (in the current CDF). The value is written at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`<PUT_,zVAR_DATASPEC_>`

Respecifies the data specification (data type and number of elements) of the current zVariable (in the current CDF). A zVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are

[8] The item `zVAR_BLOCKINGFACTOR_` was previously named `zVAR_EXTENDRECS_`.

equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (an array of characters). A string exists at each value. For the non-character data types this must be one (1) — arrays of values are not allowed for non-character data types.

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

`<PUT_,zVAR_DIMVARYS_>`

Respecifies the dimension variances of the current zVariable (in the current CDF). A zVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

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

`<PUT_,zVAR_INITIALRECS_>`

Specifies the number of records to initially write to the current zVariable (in the current CDF). The records are written beginning at record number `0` (zero). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: `long nRecords`

Number of records to write.

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

`<PUT_,zVAR_HYPERDATA_>`

Writes one or more values to the current zVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

`<PUT_,zVAR_NAME_>`

Renames the current zVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the zVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<PUT_,zVAR_PADVALUE_>`

Specifies the pad value for the current zVariable (in the current CDF). A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

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

`<PUT_,zVAR_RECVARY_>`

Respecifies the record variance of the current zVariable (in the current CDF). A zVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.9.

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

`<PUT_,zVAR_SEQDATA_>`

Writes one value to the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the zVariable, the zVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are written.

`<PUT_,zVAR_SPARSEARRAYS_>`

Specifies the sparse arrays type/parameters for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long sArraysType`

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

in: `long sArraysParms[]`

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

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

`<PUT_,zVAR_SPARSERECORDS_>`

Specifies the sparse records type for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long sRecordsType`

The sparse records type. The types of sparse records are described in Section 4.11.

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

`<PUT_,zVARs_RECDATA_>`

Writes full-physical records to one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is written at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

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

in: `long varNums[]`

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

in: `void *buffer`

The buffer of full-physical zVariable records to be written. The order of the full-physical zVariable records in this buffer must agree with the zVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to store multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to create this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`<SELECT_,ATTR_>`

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

in: `long attrNum`

Attribute number.

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

`<SELECT_,ATTR_NAME_>`

Explicitly selects the current attribute (in the current CDF) by name. **NOTE:** Selecting the current attribute by number (see `<SELECT_,ATTR_>`) is more efficient. Required arguments are as follows:

in: `char *attrName`

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

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

`<SELECT_,CDF_>`

Explicitly selects the current CDF. Required arguments are as follows:

in: `CDFid id`

Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

There are no required preselected objects/states.

`<SELECT_,CDF_CACHESIZE_>`

Selects the number of cache buffers to be used for the dotCDF file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,CDF_DECODING_>`

Selects a decoding (for the current CDF). Required arguments are as follows:

in: `long decoding`

The decoding. Specify one of the decodings described in Section 4.7.

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

`<SELECT_,CDF_NEGtoPOSfp0_MODE_>`

Selects a `-0.0` to `0.0` mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The `-0.0` to `0.0` mode. Specify one of the `-0.0` to `0.0` modes described in Section 4.15.

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

`<SELECT_,CDF_READONLY_MODE_>`

Selects a read-only mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The read-only mode. Specify one of the read-only modes described in Section 4.13.

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

`<SELECT_,CDF_SCRATCHDIR_>`

Selects a directory to be used for scratch files (by the CDF library) for the current CDF. The Concepts chapter in the CDF User's Guide describes how the CDF library uses scratch files. This scratch directory will override the directory specified by the the `CDF$TMP` logical name (on VMS systems) or `CDF_TMP` environment variable (on UNIX and MS-DOS systems). Required arguments are as follows:

in: `char *scratchDir`

The directory to be used for scratch files. The length of this directory specification is limited only by the operating system being used.

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

`<SELECT_,CDF_STATUS_>`

Selects the current status code. Required arguments are as follows:

in: `CDFstatus status`

CDF status code.

There are no required preselected objects/states.

`<SELECT_,CDF_zMODE_>`

Selects a zMode (for the current CDF). Required arguments are as follows:

in: `long mode`

The zMode. Specify one of the zModes described in Section 4.14.

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

`<SELECT_,COMPRESS_CACHESIZE_>`

Selects the number of cache buffers to be used for the compression scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,gENTRY_>`

Selects the current gEntry number for all gAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

gEntry number.

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

`<SELECT_,rENTRY_>`

Selects the current rEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

rEntry number.

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

`<SELECT_,rENTRY_NAME_>`

Selects the current rEntry number for all vAttributes (in the current CDF) by rVariable name. The number of the named rVariable becomes the current rEntry number. (The current rVariable is not changed.) **NOTE:** Selecting the current rEntry by number (see `<SELECT_,rENTRY_>`) is more efficient. Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<SELECT_,rVAR_>`

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

in: `long varNum`

rVariable number.

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

`<SELECT_,rVAR_CACHESIZE_>`

Selects the number of cache buffers to be used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,rVAR_NAME_>`

Explicitly selects the current rVariable (in the current CDF) by name. **NOTE:** Selecting the current rVariable by number (see `<SELECT_,rVAR_>`) is more efficient. Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<SELECT_,rVAR_RESERVEPERCENT_>`

Selects the reserve percentage to be used for the current rVariable (in the current CDF). This operation is only applicable to compressed rVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

in: `long percent`

The reserve percentage.

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

`<SELECT_,rVAR_SEQPOS_>`

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

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

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

`<SELECT_,rVARs_CACHESIZE_>`

Selects the number of cache buffers to be used for all of the rVariable files (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,rVARs_DIMCOUNTS_>`

Selects the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: long counts[]

Dimension counts. Each element of counts specifies the corresponding dimension count.

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

<SELECT_,rVARs_DIMINDICES_>

Selects the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: long indices[]

Dimension indices. Each element of indices specifies the corresponding dimension index.

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

<SELECT_,rVARs_DIMINTERVALS_>

Selects the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: long intervals[]

Dimension intervals. Each element of intervals specifies the corresponding dimension interval.

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

<SELECT_,rVARs_RECCOUNT_>

Selects the current record count for all rVariables in the current CDF. Required arguments are as follows:

in: long recCount

Record count.

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

<SELECT_,rVARs_RECINTERVAL_>

Selects the current record interval for all rVariables in the current CDF. Required arguments are as follows:

in: long recInterval

Record interval.

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

<SELECT_,rVARs_RECNUMBER_>

Selects the current record number for all rVariables in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

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

`<SELECT_,STAGE_CACHESIZE_>`

Selects the number of cache buffers to be used for the staging scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,zENTRY_>`

Selects the current zEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

zEntry number.

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

`<SELECT_,zENTRY_NAME_>`

Selects the current zEntry number for all vAttributes (in the current CDF) by zVariable name. The number of the named zVariable becomes the current zEntry number. (The current zVariable is not changed.) **NOTE:** Selecting the current zEntry by number (see `<SELECT_,zENTRY_>`) is more efficient. Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<SELECT_,zVAR_>`

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

in: `long varNum`

zVariable number.

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

`<SELECT_,zVAR_CACHESIZE_>`

Selects the number of cache buffers to be used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,zVAR_DIMCOUNTS_>`

Selects the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long counts[]`

Dimension counts. Each element of `counts` specifies the corresponding dimension count.

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

`<SELECT_,zVAR_DIMINDICES_>`

Selects the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long indices[]`

Dimension indices. Each element of `indices` specifies the corresponding dimension index.

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

`<SELECT_,zVAR_DIMINTERVALS_>`

Selects the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long intervals[]`

Dimension intervals. Each element of `intervals` specifies the corresponding dimension interval.

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

`<SELECT_,zVAR_NAME_>`

Explicitly selects the current zVariable (in the current CDF) by name. **NOTE:** Selecting the current zVariable by number (see `<SELECT_,zVAR_>`) is more efficient. Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

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

`<SELECT_,zVAR_RECCOUNT_>`

Selects the current record count for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recCount`

Record count.

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

`<SELECT_,zVAR_RECINTERVAL_>`

Selects the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recInterval`

Record interval.

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

`<SELECT_,zVAR_RECNUMBER_>`

Selects the current record number for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

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

`<SELECT_,zVAR_RESERVEPERCENT_>`

Selects the reserve percentage to be used for the current zVariable (in the current CDF). This operation is only applicable to compressed zVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

in: `long percent`

The reserve percentage.

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

`<SELECT_,zVAR_SEQPOS_>`

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

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but must be present).

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

`<SELECT_,zVARs_CACHESIZE_>`

Selects the number of cache buffers to be used for all of the zVariable files (of the current

CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

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

`<SELECT_,zVARs_RECNUMBER_>`

Selects the current record number for each zVariable in the current CDF. This operation is provided to simplify the selection of the current record numbers for the zVariables involved in a multiple variable access operation (see the Concepts chapter in the CDF User's Guide). Required arguments are as follows:

in: `long recNum`

Record number.

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

6.7 More Examples

Several more examples of the use of `CDFlib` follow. In each example it is assumed that the current CDF has already been selected (either implicitly by creating/opening the CDF or explicitly with `<SELECT_,CDF_>`).

6.7.1 rVariable Creation.

In this example an rVariable will be created with a pad value being specified; initial records will be written; and the rVariable's blocking factor will be specified. Note that the pad value was specified before the initial records. This results in the specified pad value being written. Had the pad value not been specified first, the initial records would have been written with the default pad value. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;                   /* Status returned from CDF
                                         library. */
long        dimVarys[2];              /* Dimension variances. */
long        varNum;                   /* rVariable number. */
float       padValue = -999.9;        /* Pad value. */
.
.
dimVarys[0] = VARY;
```

```
dimVarys[1] = VARY;
status = CDFlib (CREATE_, rVAR_, "HUMIDITY", CDF_REAL4, 1, VARY,
                                 dimVarys, &varNum,
                 PUT_, rVAR_PADVALUE_, &padValue,
                       rVAR_INITIALRECS_, (long) 500,
                       rVAR_BLOCKINGFACTOR_, (long) 50,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.2 zVariable Creation (Character Data Type).

In this example a zVariable with a character data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;                     /* Status returned from CDF
                                           library. */
long        dimVarys[1];                /* Dimension variances. */
long        varNum;                     /* zVariable number. */
long        numDims = 1;                /* Number of dimensions. */
static long dimSizes[1] = { 20 };       /* Dimension sizes. */
long        numElems = 10;              /* Number of elements (characters
                                           in this case). */
static char padValue = "**********";    /* Pad value. */
.
.
dimVarys[0] = VARY;
status = CDFlib (CREATE_, zVAR_, "Station", CDF_CHAR, numElems, numDims,
                                 dimSizes, NOVARY, dimVarys, &varNum,
                 PUT_, zVAR_PADVALUE_, padValue,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.3 Hyper Read with Subsampling.

In this example an rVariable will be subsampled in a CDF whose rVariables are 2-dimensional and have dimension sizes [100,200]. The CDF is row major, and the data type of the rVariable is CDF_UINT2. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;               /* Status returned from CDF library. */
unsigned short values[50][100];      /* Buffer to receive values. */
long           recCount = 1;         /* Record count, one record per hyper
                                        get. */
long           recInterval = 1;      /* Record interval, set to one to
                                        indicate contiguous records (really
                                        meaningless since record count is
                                        one). */
static long    indices[2] = {0,0};   /* Dimension indices, start each read
                                        at 0,0 of the array. */
static long    counts[2] = {50,100}; /* Dimension counts, half of the
                                        values along each dimension will
                                        be read. */
static long    intervals[2] = {2,2}; /* Dimension intervals, every other
                                        value along each dimension will be
                                        read. */
long           recNum;               /* Record number. */
long           maxRec;               /* Maximum rVariable record
                                        number in the CDF - this was
                                        determined with a call to
                                        CDFinquire. */
.
.
status = CDFlib (SELECT_, rVAR_NAME_, "BRIGHTNESS",
                          rVARs_RECCOUNT_, recCount,
                          rVARs_RECINTERVAL_, recInterval,
                          rVARs_DIMINDICES_, indices,
                          rVARs_DIMCOUNTS_, counts,
                          rVARs_DIMINTERVALS_, intervals,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);

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

6.7.4 Attribute Renaming.

In this example the attribute named Tmp will be renamed to TMP. It is assumed that the current CDF has already been selected.

```
   .
   .
   #include "cdf.h"
   .
   .
   CDFstatus      status;               /* Status returned from CDF library. */
   .
   .
   status = CDFlib (SELECT_, ATTR_NAME_, "Tmp",
                    PUT_, ATTR_NAME, "TMP",
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
```

6.7.5 Sequential Access.

In this example the values for a zVariable will be averaged. The values will be read using the sequential access method (see the Concepts chapter in the CDF User's Guide). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be CDF_REAL4. It is assumed that the current CDF has already been selected.

```
   .
   .
   #include "cdf.h"
   .
   .
   CDFstatus   status;                /* Status returned from CDF library. */
   long        varNum;                /* zVariable number. */
   long        recNum = 0;            /* Record number, start at first record. */
   static long indices[2] = {0,0};  /* Dimension indices. */
   float       value;                 /* Value read. */
   double      sum = 0.0;             /* Sum of all values. */
   long        count = 0;             /* Number of values. */
   float       ave;                   /* Average value. */
   .
   .
   status = CDFlib (GET_, zVAR_NUMBER_, "FLUX", &varNum,
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);

   status = CDFlib (SELECT_, zVAR_, varNum,
                             zVAR_SEQPOS_, recNum, indices,
                 GET_, zVAR_SEQDATA_, &value,
                 NULL_);
while (status >= CDF_OK) {
  sum += value;
  count++;
  status = CDFlib (GET_, zVAR_SEQDATA_, &value,
                   NULL_);
}

if (status != END_OF_VAR) UserStatusHandler (status);

ave = sum / count;
.
.
```

6.7.6 Attribute rEntry Writes.

In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;                  /* Status returned from CDF
                                           library. */
static float   scale[2] = {-90.0,90.0};  /* Scale, minimum/maximum. */
.
.
status = CDFlib (SELECT_, rENTRY_NAME_, "LATITUDE",
                          ATTR_NAME_, "FIELDNAM",
                 PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                     "Latitude            ",
                 SELECT_, ATTR_NAME_, "SCALE",
                 PUT_, rENTRY_DATA_, CDF_REAL4, (long) 2, scale,
                 SELECT_, ATTR_NAME_, "UNITS",
                 PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                     "Degrees north       ",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.7 Multiple zVariable Write.

In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see the Concepts chapter in the CDF User's Guide). It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;              /* Status returned from CDF library. */
short       time;                /* 'Time' value. */
char        vectorA[3];          /* 'vectorA' values. */
double      vectorB[5];          /* 'vectorB' values. */
long        recNumber;           /* Record number. */
char        buffer[45];          /* Buffer of full-physical records. */
long        varNumbers[3];       /* Variable numbers. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "vectorB", &varNumbers[0],
                       zVAR_NUMBER_, "time", &varNumbers[1],
                       zVAR_NUMBER_, "vectorA", &varNumbers[2],
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
for (recNumber = 0; recNumber < 100; recNumber++) {
   .
   /* read values from input file */
   .
   memmove (&buffer[0], vectorB, 40);
   memmove (&buffer[40], &time, 2);
   memmove (&buffer[42], vectorA, 3);
   status = CDFlib (SELECT_, zVARs_RECNUMBER_, recNumber,
                    PUT_, zVARs_RECDATA_, 3L, varNumbers, buffer,
                    NULL_);
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

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

6.8 A Potential Mistake We Don't Want You to Make

The following example illustrates one of the most common mistakes made when using the Internal Interface in a C application. Please don't do something like the following:

```
.
.
#include "cdf.h"
.
.
CDFid          id;                        /* CDF identifier (handle). */
CDFstatus      status;                    /* Status returned from CDF
                                             library. */
long           varNum;                    /* zVariable number. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                 GET_, zVAR_NUMBER_, "EPOCH", &varNum,
                 SELECT_, zVAR_, varNum,                  /* ERROR! */
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

It looks like the current zVariable will be selected based on the zVariable number determined by using the <GET_,zVAR_NUMBER_> operation. What actually happens is that the zVariable number passed to the <SELECT_,zVAR_> operation is undefined. This is because the C compiler is passing varNum by value rather than reference.[9] Since the argument list passed to CDFlib is created before CDFlib is called, varNum does not yet have a value. Only after the <GET_,zVAR_NUMBER_> operation is performed does varNum have a valid value. But at that point it's too late since the argument list has already been created. In this type of situation you would have to make two calls to CDFlib. The first would inquire the zVariable number and the second would select the current zVariable.

6.9 Custom C Functions

Most of the Standard Interface functions callable from C applications are implemented as C macros that call CDFlib (Internal Interface). For example, the CDFcreate function is actually defined as the following C macro:

```
#define CDFcreate(CDFname,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, id, \
        PUT_, CDF_ENCODING_, encoding, \
              CDF_MAJORITY_, majority, \
        NULL_)
```

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

These macros are defined in cdf.h. Where your application calls CDFcreate, the C compiler (preprocessor) expands the macro into the corresponding call to CDFlib.

The flexibility of CDFlib allows you to define your own custom CDF functions using C macros. For instance, a function that inquires the format of a CDF could be defined as follows:

```
#define CDFinquireFormat(id,format) \
CDFlib (SELECT_, CDF_, id, \
        GET_, CDF_FORMAT_, format, \
        NULL_)
```

Your application would call the function as follows:

```
.
.
CDFid     id;            /* CDF identifier. */
CDFstatus status;        /* Returned status code. */
long      format;        /* Format of CDF. */
.
.
status = CDFinquireFormat (id, &format);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Chapter 7

Interpreting CDF Status Codes

Most CDF functions return a status code of type `CDFstatus`. The symbolic names for these codes are defined in `cdf.h` and should be used in your applications rather than using the true numeric values. Appendix A explains each status code. When the status code returned from a CDF function is tested, the following rules apply.

`status > CDF_OK` Indicates successful completion but some additional information is provided. These are informational codes.

`status = CDF_OK` Indicates successful completion.

`CDF_WARN < status < CDF_OK` Indicates that the function completed but probably not as expected. These are warning codes.

`status < CDF_WARN` Indicates that the function did not complete. These are error codes.

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

```
CDFstatus status;
.
.
status = CDFfunction (...);   /* any CDF function returning CDFstatus */
if (status != CDF_OK) {
  UserStatusHandler (status, ...);
  .
  .
}
```

In your own status handler you can take whatever action is appropriate to the application. An example status handler follows. Note that no action is taken in the status handler if the status is `CDF_OK`.

```
#include <stdio.h>
```

```
#include "cdf.h"

void UserStatusHandler (status)
CDFstatus status;
{
  char message[CDF_STATUSTEXT_LEN+1];
  if (status < CDF_WARN) {
    printf ("An error has occurred, halting...\n");
    CDFerror (status, message);
    printf ("%s\n", message);
    exit (status);
  }
  else {
    if (status < CDF_OK) {
      printf ("Warning, function may not have completed as expected...\n");
      CDFerror (status, message);
      printf ("%s\n", message);
    }
    else {
      if (status > CDF_OK) {
        printf ("Function completed successfully, but be advised that...\n");
        CDFerror (status, message);
        printf ("%s\n", message);
      }
    }
  }
  return;
}
```

Explanations for all CDF status codes are available to your applications through the function `CDFerror`. `CDFerror` encodes in a text string an explanation of a given status code.

Chapter 8

EPOCH Utility Routines

Several functions exist that compute, decompose, parse, and encode CDF_EPOCH values. These functions may be called by applications using the CDF_EPOCH data type and are included in the CDF library. Function prototypes for these functions may be found in the include file cdf.h. The Concepts chapter in the CDF User's Guide describes EPOCH values.

8.1 computeEPOCH

computeEPOCH calculates a CDF_EPOCH value given the individual components. If an illegal component is detected, the value returned will be -1.0.

```
double computeEPOCH(     /* Out -- CDF_EPOCH value returned. */
long year,               /* In  -- Year (AD, e.g., 1994). */
long month,              /* In  -- Month (1-12). */
long day,                /* In  -- Day (1-31). */
long hour,               /* In  -- Hour (0-23). */
long minute,             /* In  -- Minute (0-59). */
long second,             /* In  -- Second (0-59). */
long msec);              /* In  -- Millisecond (0-999). */
```

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.

8.2 EPOCHbreakdown

EPOCHbreakdown decomposes a CDF_EPOCH value into the individual components.

```
void EPOCHbreakdown(
double epoch,           /* In  -- The CDF_EPOCH value. */
long   *year,           /* Out -- Year (AD, e.g., 1994). */
long   *month,          /* Out -- Month (1-12). */
long   *day,            /* Out -- Day (1-31). */
long   *hour,           /* Out -- Hour (0-23). */
long   *minute,         /* Out -- Minute (0-59). */
long   *second,         /* Out -- Second (0-59). */
long   *msec);          /* Out -- Millisecond (0-999). */
```

8.3 encodeEPOCH

encodeEPOCH encodes a CDF_EPOCH value into the standard date/time character string. The format of the string is dd-mmm-yyyy hh:mm:ss.ccc where dd is the day of the month (1-31), mmm is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), yyyy is the year, hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), and ccc is the millisecond (0-999).

```
void encodeEPOCH(
double epoch;                          /* In  -- The CDF_EPOCH value. */
char epString[EPOCH_STRING_LEN+1]);    /* Out -- The standard date/time
                                                 character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

8.4 encodeEPOCH1

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

```
void encodeEPOCH1(
double epoch;                          /* In  -- The CDF_EPOCH value. */
char epString[EPOCH1_STRING_LEN+1]);   /* Out -- The alternate date/time
                                                 character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

8.5 encodeEPOCH2

encodeEPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymoddhhmmss where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), and ss is the second (0-59).

```
void encodeEPOCH2(
double epoch;                           /* In  -- The CDF_EPOCH value. */
char epString[EPOCH2_STRING_LEN+1]);    /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

8.6 encodeEPOCH3

encodeEPOCH3 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.cccZ where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), and ccc is the millisecond (0-999).

```
void encodeEPOCH3(
double epoch;                           /* In  -- The CDF_EPOCH value. */
char epString[EPOCH3_STRING_LEN+1]);    /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH3_STRING_LEN is defined in cdf.h.

8.7 encodeEPOCHx

encodeEPOCHx encodes a CDF_EPOCH value into a custom date/time character string. The format of the encoded string is specified by a format string.

```
void encodeEPOCHx(
double epoch;                           /* In  -- The CDF_EPOCH value. */
char format[EPOCHx_FORMAT_MAX];         /* In  ---The format string. */
char encoded[EPOCHx_STRING_MAX]);       /* Out -- The custom date/time
                                                  character string. */
```

The format string consists of EPOCH components which are encoded and text which 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 8.3) would be...

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

EPOCHx_FORMAT_LEN and EPOCHx_STRING_MAX are defined in cdf.h.

8.8 parseEPOCH

parseEPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH function described in Section 8.3. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH(                    /* Out -- CDF_EPOCH value
                                                returned. */
char epString[EPOCH_STRING_LEN+1]);   /* In  -- The standard date/time
                                                character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

8.9 parseEPOCH1

parseEPOCH1 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH1 function described in Section 8.4. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH1(                   /* Out -- CDF_EPOCH value
                                                returned. */
char epString[EPOCH1_STRING_LEN+1]);  /* In  -- The alternate date/time
                                                character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

8.10 parseEPOCH2

parseEPOCH2 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH2 function described in Section 8.5. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH2(                   /* Out -- CDF_EPOCH value
                                                returned. */
char epString[EPOCH2_STRING_LEN+1]);  /* In  -- The alternate date/time
                                                character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

8.11 parseEPOCH3

parseEPOCH3 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH3 function described in Section 8.6. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH3(                   /* Out -- CDF_EPOCH value
                                                returned. */
char epString[EPOCH3_STRING_LEN+1]);  /* In  -- The alternate date/time
                                                character string. */
```

EPOCH3_STRING_LEN is defined in cdf.h.

Appendix A

Status Codes

A.1 Introduction

A status code is returned from most CDF functions. The `cdf.h` (for C) and `CDF.INC` (for Fortran) include files contain the numerical values (constants) for each of the status codes (and for any other constants referred to in the explanations). The CDF library Standard Interface functions `CDFerror` (for C) and `CDF_error` (for Fortran) can be used within a program to inquire the explanation text for a given status code. The Internal Interface can also be used to inquire explanation text.

There are three classes of status codes: informational, warning, and error. The purpose of each is as follows:

Informational	Indicates success but provides some additional information that may be of interest to an application.
Warning	Indicates that the function completed but possibly not as expected.
Error	Indicates that a fatal error occurred and the function aborted.

Status codes fall into classes as follows:

Error codes < `CDF_WARN` < Warning codes < `CDF_OK` < Informational codes

`CDF_OK` indicates an unqualified success (it should be the most commonly returned status code). `CDF_WARN` is simply used to distinguish between warning and error status codes.

A.2 Status Codes and Messages

The following list contains an explanation for each possible status code. Whether a particular status code is considered informational, a warning, or an error is also indicated.

`ATTR_EXISTS`	Named attribute already exists — cannot create or rename.

Each attribute in a CDF must have a unique name. Note that trailing blanks are ignored by the CDF library when comparing attribute names. [Error]

ATTR_NAME_TRUNC
Attribute name truncated to CDF_ATTR_NAME_LEN 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[1]
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 which has been renamed with a different file extension or no file extension. [Error]

BAD_CDF_ID
CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF. [Error]

BAD_CDF_NAME
Illegal CDF name specified. CDF names must contain at least one character, and each character must be printable. Trailing blanks are allowed but will be ignored. [Error]

BAD_CDFSTATUS
Unknown CDF status code received. The status code specified is not used by the CDF library. [Error]

BAD_COMPRESSION_PARM
An illegal compression parameter was specified. [Error]

BAD_DATA_TYPE
An unknown data type was specified or encountered. The CDF data types are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

BAD_DECODING
An unknown decoding was specified. The CDF decodings are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

BAD_DIM_COUNT
Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension.

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

[Error]

`BAD_DIM_INDEX` One or more dimension index is out of range. A valid value must be specified regardless of the dimension variance. Note also that the combination of dimension index, count, and interval must not specify an element beyond the end of the dimension. [Error]

`BAD_DIM_INTERVAL` Illegal dimension interval specified. Dimension intervals must be at least one (1). [Error]

`BAD_DIM_SIZE` Illegal dimension size specified. A dimension size must be at least one (1). [Error]

`BAD_ENCODING` Unknown data encoding specified. The CDF encodings are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_ENTRY_NUM` Illegal attribute entry number specified. Entry numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. [Error]

`BAD_FNC_OR_ITEM` The specified function or item is illegal. Check that the proper number of arguments are specified for each operation being performed. Also make sure that `NULL_` is specified as the last operation. [Error]

`BAD_FORMAT` Unknown format specified. The CDF formats are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_INITIAL_RECS` An illegal number of records to initially write has been specified. The number of initial records must be at least one (1). [Error]

`BAD_MAJORITY` Unknown variable majority specified. The CDF variable majorities are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_MALLOC` Unable to allocate dynamic memory — system limit reached. Contact CDF User Support if this error occurs. [Error]

`BAD_NEGtoPOSfp0_MODE` An illegal `-0.0` to `0.0` mode was specified. The `-0.0` to `0.0` modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_NUM_DIMS` The number of dimensions specified is out of the allowed range. Zero (0) through `CDF_MAX_DIMS` dimensions are allowed. If more are needed, contact CDF User Support. [Error]

`BAD_NUM_ELEMS` The number of elements of the data type is illegal. The number of elements must be at least one (1). For variables with a non-character data type, the number of elements must always be one (1). [Error]

`BAD_NUM_VARS` Illegal number of variables in a record access operation. [Error]

`BAD_READONLY_MODE`
Illegal read-only mode specified. The CDF read-only modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_REC_COUNT`
Illegal record count specified. A record count must be at least one (1). [Error]

`BAD_REC_INTERVAL`
Illegal record interval specified. A record interval must be at least one (1). [Error]

`BAD_REC_NUM`
Record number is out of range. Record numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. Note that a valid value must be specified regardless of the record variance. [Error]

`BAD_SCOPE`
Unknown attribute scope specified. The attribute scopes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`BAD_SCRATCH_DIR`
An illegal scratch directory was specified. The scratch directory must be writeable and accessable (if a relative path was specified) from the directory in which the application has been executed. [Error]

`BAD_SPARSEARRAYS_PARM`
An illegal sparse arrays parameter was specified. [Error]

`BAD_VAR_NAME`
Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]

`BAD_VAR_NUM`
Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]

`BAD_zMODE`
Illegal zMode specified. The CDF zModes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]

`CANNOT_ALLOCATE_RECORDS`
Records cannot be allocated for the given type of variable (e.g., a compressed variable). [Error]

`CANNOT_CHANGE`
Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow:

1. Changing a CDF's data encoding after a variable value (including a pad value) or an attribute entry has been written.
2. Changing a CDF's format after a variable has been created or if a compressed single-file CDF.
3. Changing a CDF's variable majority after a variable value (excluding a pad value) has been written.
4. Changing a variable's data specification after a value (including the pad value) has been written to

that variable or after records have been allocated for that variable.

5. Changing a variable's record variance after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
6. Changing a variable's dimension variances after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
7. Writing "initial" records to a variable after a value (excluding the pad value) has already been written to that variable.
8. Changing a variable's blocking factor when a compressed variable and a value (excluding the pad value) has been written or when a variable with sparse records and a value has been accessed.
9. Changing an attribute entry's data specification where the new specification is not equivalent to the old specification.

CANNOT_COMPRESS
The CDF or variable cannot be compressed. For CDFs, this occurs if the CDF has the multi-file format. For variables, this occurs if the variable is in a multi-file CDF, values have been written to the variable, or if sparse arrays have already been specified for the variable. [Error]

CANNOT_SPARSEARRAYS
Sparse arrays cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, records have been allocated for the variable, or if compression has already been specified for the variable. [Error]

CANNOT_SPARSERECORDS
Sparse records cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, or records have been allocated for the variable. [Error]

CDF_CLOSE_ERROR
Error detected while trying to close CDF. Check that sufficient disk space exists for the dotCDF file and that it has not been corrupted. [Error]

CDF_CREATE_ERROR
Cannot create the CDF specified — error from file system. Make sure 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. Unsufficient 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 pathname truncated to CDF_PATHNAME_LEN characters. The CDF was created but with a truncated name. [Warning]

CDF_OK
Function completed successfully.

CDF_OPEN_ERROR
Cannot open 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]

COMPRESSION_ERROR
An error occured 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 occured 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 choosen 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]

PC_OVERFLOW
An operation involving a buffer greater than 64k bytes in size has been specified. [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 pathname 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. This can also 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 preceeding 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 writable. [Error]

SCRATCH_DELETE_ERROR

Cannot delete a scratch file — error from file system. [Error]

SCRATCH_READ_ERROR

Cannot read from a scratch file — error from file system. [Error]

SCRATCH_WRITE_ERROR

Cannot write to a scratch file — error from file system. [Error]

SINGLE_FILE_FORMAT

The specified operation is not applicable to CDFs with the single-file format. For example, it does not make sense to close a variable in a single-file CDF. [Informational]

SOME_ALREADY_ALLOCATED

Some of the records being allocated were already allocated. [Informational]

TOO_MANY_PARMS

A type of sparse arrays or compression was encountered having too many parameters. This could be causes by a corrupted CDF or if the CDF was created/modified by a CDF distribution more recent than the one being used. [Error]

TOO_MANY_VARS

A multi-file CDF on a PC may contain only a limited number of variables because of the 8.3 file naming convention of MS-DOS. This consists of 100 rVariables and 100 zVariables. [Error]

UNKNOWN_COMPRESSION

An unknown type of compression was specified or encountered. [Error]

UNKNOWN_SPARSENESS

An unknown type of sparseness was specified or encountered. [Error]

`UNSUPPORTED_OPERATION` The attempted operation is not supported at this time. [Error]

`VAR_ALREADY_CLOSED` The specified variable is already closed. [Informational]

`VAR_CLOSE_ERROR` Error detected while trying to close variable file. Check that sufficient disk space exists for the variable file and that it has not been corrupted. [Error]

`VAR_CREATE_ERROR` An error occurred while creating a variable file in a multi-file CDF. Check that a file quota has not been reached. [Error]

`VAR_DELETE_ERROR` An error occurred while deleting a variable file in a multi-file CDF. Check that sufficient privilege exist to delete the CDF files. [Error]

`VAR_EXISTS` Named variable already exists - cannot create or rename. Each variable in a CDF must have a unique name (rVariables and zVariables can not share names). Note that trailing blanks are ignored by the CDF library when comparing variable names. [Error]

`VAR_NAME_TRUNC` Variable name truncated to `CDF_VAR_NAME_LEN` characters. The variable was created but with a truncated name. [Warning]

`VAR_OPEN_ERROR` An error occurred while opening variable file. Check that sufficient privilege exists to open the variable file. Also make sure that the associated variable file exists. [Error]

`VAR_READ_ERROR` Failed to read variable as requested — error from file system. Check that the associated file is not corrupted. [Error]

`VAR_WRITE_ERROR` Failed to write variable as requested — error from file system. Check that the associated file is not corrupted. [Error]

`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

C Programming Summary

B.1 Standard Interface

```
CDFstatus CDFcreate (CDFname, numDims, dimSizes, encoding, majority, id)
char  *CDFname;                                                  /* in  */
long  numDims;                                                   /* in  */
long  dimSizes[];                                                /* in  */
long  encoding;                                                  /* in  */
long  majority;                                                  /* in  */
CDFid *id;                                                       /* out */

CDFstatus CDFopen (CDFname, id)
char  *CDFname;                                                  /* in  */
CDFid *id;                                                       /* out */

CDFstatus CDFdoc (id, version, release, text)
CDFid id;                                                        /* in  */
long  *version;                                                  /* out */
long  *release;                                                  /* out */
char  text[CDF_DOCUMENT_LEN+1];                                  /* out */

CDFstatus CDFinquire (id, numDims, dimSizes, encoding, majority, maxRec,
                      numVars, numAttrs)
CDFid id;                                                        /* in  */
long  *numDims;                                                  /* out */
long  dimSizes[CDF_MAX_DIMS];                                    /* out */
long  *encoding;                                                 /* out */
long  *majority;                                                 /* out */
long  *maxRec;                                                   /* out */
long  *numVars;                                                  /* out */
long  *numAttrs;                                                 /* out */

CDFstatus CDFclose (id)
```

```
CDFid id;                                                      /* in  */

CDFstatus CDFdelete (id)
CDFid id;                                                      /* in  */

CDFstatus CDFerror (status, message)
CDFstatus status;                                              /* in  */
char message[CDF_STATUSTEXT_LEN+1];                            /* out */

CDFstatus CDFattrCreate (id, attrName, attrScope, attrNum)
CDFid id;                                                      /* in  */
char  *attrName;                                               /* in  */
long  attrScope;                                               /* in  */
long  *attrNum;                                                /* out */

long CDFattrNum (id, attrName)
CDFid id;                                                      /* in  */
char  *attrName;                                               /* in  */

CDFstatus CDFattrRename (id, attrNum, attrName)
CDFid id;                                                      /* in  */
long  attrNum;                                                 /* in  */
char  *attrName;                                               /* in  */

CDFstatus CDFattrInquire (id, attrNum, attrName, attrScope, maxEntry)
CDFid id;                                                      /* in  */
long  attrNum;                                                 /* in  */
char  *attrName;                                               /* out */
long  *attrScope;                                              /* out */
long  *maxEntry;                                               /* out */

CDFstatus CDFattrEntryInquire (id, attrNum, entryNum, dataType,
                               numElements)
CDFid id;                                                      /* in  */
long  attrNum;                                                 /* in  */
long  entryNum;                                                /* in  */
long  *dataType;                                               /* out */
long  *numElements;                                            /* out */

CDFstatus CDFattrPut (id, attrNum, entryNum, dataType, numElements,
                      value)
CDFid id;                                                      /* in  */
long  attrNum;                                                 /* in  */
long  entryNum;                                                /* in  */
long  dataType;                                                /* in  */
long  numElements;                                             /* in  */
void  *value;                                                  /* in  */

CDFstatus CDFattrGet (id, attrNum, entryNum, value)
CDFid id;                                                      /* in  */
long  attrNum;                                                 /* in  */
long  entryNum;                                                       /* in  */
void  *value;                                                         /* out */

CDFstatus CDFvarCreate (id, varName, dataType, numElements,
                        recVariances, dimVariances, varNum)
CDFid id;                                                             /* in  */
char  *varName;                                                       /* in  */
long  dataType;                                                       /* in  */
long  numElements;                                                    /* in  */
long  recVariance;                                                    /* in  */
long  dimVariances[];                                                 /* in  */
long  *varNum;                                                        /* out */

long CDFvarNum (id, varName)
CDFid id;                                                             /* in  */
char  *varName;                                                       /* in  */

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

CDFstatus CDFvarInquire (id, varNum, varName, dataType, numElements,
                         recVariance, dimVariances)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
char  *varName;                                                       /* out */
long  *dataType;                                                      /* out */
long  *numElements;                                                   /* out */
long  *recVariance;                                                   /* out */
long  dimVariances[CDF_MAX_DIMS];                                     /* out */

CDFstatus CDFvarPut (id, varNum, recNum, indices, value)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
long  recNum;                                                         /* in  */
long  indices[];                                                      /* in  */
void  *value;                                                         /* in  */

CDFstatus CDFvarGet (id, varNum, recNum, indices, value)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
long  recNum;                                                         /* in  */
long  indices[];                                                      /* in  */
void  *value;                                                         /* out */

CDFstatus CDFvarHyperPut (id, varNum, recStart, recCount, recInterval,
                          indices, counts, intervals, buffer)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
long  recStart;                                                       /* in  */
long  recCount;                                                       /* in  */
long  recInterval;                                                    /* in  */
long  indices[];                                                      /* in  */
long  counts[];                                                       /* in  */
long  intervals[];                                                    /* in  */
void  *buffer;                                                        /* in  */

CDFstatus CDFvarHyperGet (id, varNum, recStart, recCount, recInterval,
                          indices, counts, intervals, buffer)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
long  recStart;                                                       /* in  */
long  recCount;                                                       /* in  */
long  recInterval;                                                    /* in  */
long  indices[];                                                      /* in  */
long  counts[];                                                       /* in  */
long  intervals[];                                                    /* in  */
void  *buffer;                                                        /* out */

CDFstatus CDFvarClose (id, varNum)
CDFid id;                                                             /* in  */
long  varNum;                                                         /* in  */
```

B.2 Internal Interface

```
CDFstatus CDFlib (op, ...)
long  op;                                                             /* in  */

   CLOSE_
              CDF_
              rVAR_
              zVAR_

   CONFIRM_
              ATTR_                    long *attrNum                  /* out */
              ATTR_EXISTENCE_          char *attrName                 /* in  */
              CDF_                     CDFid *id                      /* out */
              CDF_ACCESS_
              CDF_CACHESIZE_           long *numBuffers               /* out */
              CDF_DECODING_            long *decoding                 /* out */
              CDF_NAME_                char CDFname[CDF_PATHNAME_LEN+1]
                                                                      /* out */
              CDF_NEGtoPOSfp0_MODE_    long *mode                     /* out */
              CDF_READONLY_MODE_       long *mode                     /* out */
              CDF_STATUS_              CDFstatus *status              /* out */
              CDF_zMODE_               long *mode                     /* out */
              COMPRESS_CACHESIZE_      long *numBuffers               /* out */
              CURgENTRY_EXISTENCE_
              CURrENTRY_EXISTENCE_
```

```
           CURzENTRY_EXISTENCE_
           gENTRY_                  long *entryNum                 /* out */
           gENTRY_EXISTENCE_        long entryNum                  /* in  */
           rENTRY_                  long *entryNum                 /* out */
           rENTRY_EXISTENCE_        long entryNum                  /* in  */
           rVAR_                    long *varNum                   /* out */
           rVAR_CACHESIZE_          long *numBuffers               /* out */
           rVAR_EXISTENCE_          char *varName                  /* in  */
           rVAR_PADVALUE_
           rVAR_RESERVEPERCENT_     long *percent                  /* out */
           rVAR_SEQPOS_             long *recNum                   /* out */
                                    long indices[CDF_MAX_DIMS]     /* out */
           rVARs_DIMCOUNTS_         long counts[CDF_MAX_DIMS]      /* out */
           rVARs_DIMINDICES_        long indices[CDF_MAX_DIMS]     /* out */
           rVARs_DIMINTERVALS_      long intervals[CDF_MAX_DIMS]   /* out */
           rVARs_RECCOUNT_          long *recCount                 /* out */
           rVARs_RECINTERVAL_       long *recInterval              /* out */
           rVARs_RECNUMBER_         long *recNum                   /* out */
           STAGE_CACHESIZE_         long *numBuffers               /* out */
           zENTRY_                  long *entryNum                 /* out */
           zENTRY_EXISTENCE_        long entryNum                  /* in  */
           zVAR_                    long *varNum                   /* out */
           zVAR_CACHESIZE_          long *numBuffers               /* out */
           zVAR_DIMCOUNTS_          long counts[CDF_MAX_DIMS]      /* out */
           zVAR_DIMINDICES_         long indices[CDF_MAX_DIMS]     /* out */
           zVAR_DIMINTERVALS_       long intervals[CDF_MAX_DIMS]   /* out */
           zVAR_EXISTENCE_          char *varName                  /* in  */
           zVAR_PADVALUE_
           zVAR_RECCOUNT_           long *recCount                 /* out */
           zVAR_RECINTERVAL_        long *recInterval              /* out */
           zVAR_RECNUMBER_          long *recNum                   /* out */
           zVAR_RESERVEPERCENT_     long *percent                  /* out */
           zVAR_SEQPOS_             long *recNum                   /* out */
                                    long indices[CDF_MAX_DIMS]     /* out */

CREATE_
           ATTR_                    char *attrName                 /* in  */
                                    long scope                     /* in  */
                                    long *attrNum                  /* out */

           CDF_                     char *CDFname                  /* in  */
                                    long numDims                   /* in  */
                                    long dimSizes[]                /* in  */
                                    CDFid *id                      /* out */

           rVAR_                    char *varName                  /* in  */
                                    long dataType                  /* in  */
                                    long numElements               /* in  */
                                    long recVary                   /* in  */
                                    long dimVarys                  /* in  */
                                    long *varNum                   /* out */
           zVAR_                    char *varName                  /* in  */
                                    long dataType                  /* in  */
                                    long numElements               /* in  */
                                    long numDims                   /* in  */
                                    long dimSizes[]                /* in  */
                                    long recVary                   /* in  */
                                    long dimVarys                  /* in  */
                                    long *varNum                   /* out */

DELETE_
           ATTR_
           CDF_
           gENTRY_
           rENTRY_
           rVAR_
           rVAR_RECORDS_            long firstRecord               /* in  */
                                    long lastRecord                /* in  */
           zENTRY_
           zVAR_
           zVAR_RECORDS_            long firstRecord               /* in  */
                                    long lastRecord                /* in  */

GET_
           ATTR_MAXgENTRY_          long *maxEntry                 /* out */
           ATTR_MAXrENTRY_          long *maxEntry                 /* out */
           ATTR_MAXzENTRY_          long *maxEntry                 /* out */
           ATTR_NAME_               char attrName[CDF_ATTR_NAME_LEN+1]
                                                                   /* out */
           ATTR_NUMBER_             char *attrName                 /* in  */
                                    long *attrNum                  /* out */
           ATTR_NUMgENTRIES_        long *numEntries               /* out */
           ATTR_NUMrENTRIES_        long *numEntries               /* out */
           ATTR_NUMzENTRIES_        long *numEntries               /* out */
           ATTR_SCOPE_              long *scope                    /* out */
           CDF_COMPRESSION_         long *cType                    /* out */
                                    long cParms[CDF_MAX_PARMS]     /* out */
                                    long *cPct                     /* out */
           CDF_COPYRIGHT_           char copyRight[CDF_COPYRIGHT_LEN+1]
                                                                   /* out */
           CDF_ENCODING_            long *encoding                 /* out */
           CDF_FORMAT_              long *format                   /* out */
           CDF_INCREMENT_           long *increment                /* out */
           CDF_INFO_                char *name                     /* in  */
                                    long *cType                    /* out */
                                    long cParms[CDF_MAX_PARMS]     /* out */
                                    long *cSize                    /* out */
                                    long *uSize                    /* out */
           CDF_MAJORITY_            long *majority                 /* out */
           CDF_NUMATTRS_            long *numAttrs                 /* out */
           CDF_NUMgATTRS_           long *numAttrs                 /* out */
```

```
CDF_NUMrVARS_            long *numVars                  /* out */
CDF_NUMvATTRS_           long *numAttrs                 /* out */
CDF_NUMzVARS_            long *numVars                  /* out */
CDF_RELEASE_             long *release                  /* out */
CDF_VERSION_             long *version                  /* out */
DATATYPE_SIZE_           long dataType                  /* in  */
                         long *numBytes                 /* out */
gENTRY_DATA_             void *value                    /* out */
gENTRY_DATATYPE_         long *dataType                 /* out */
gENTRY_NUMELEMS_         long *numElements              /* out */
LIB_COPYRIGHT_           char copyRight[CDF_COPYRIGHT_LEN+1]
                                                        /* out */
LIB_INCREMENT_           long *increment                /* out */
LIB_RELEASE_             long *release                  /* out */
LIB_subINCREMENT_        char *subincrement             /* out */
LIB_VERSION_             long *version                  /* out */
rENTRY_DATA_             void *value                    /* out */
rENTRY_DATATYPE_         long *dataType                 /* out */
rENTRY_NUMELEMS_         long *numElements              /* out */
rVAR_ALLOCATEDFROM_      long startRecord               /* in  */
                         long *nextRecord               /* out */
rVAR_ALLOCATEDTO_        long startRecord               /* in  */
                         long *lastRecord               /* out */
rVAR_BLOCKINGFACTOR_     long *blockingFactor           /* out */
rVAR_COMPRESSION_        long *cType                    /* out */
                         long cParms[CDF_MAX_PARMS]     /* out */
                         long *cPct                     /* out */
rVAR_DATA_               void *value                    /* out */
rVAR_DATATYPE_           long *dataType                 /* out */
rVAR_DIMVARYS_           long dimVarys[CDF_MAX_DIMS]    /* out */
rVAR_HYPERDATA_          void *buffer                   /* out */
rVAR_MAXallocREC_        long *maxRec                   /* out */
rVAR_MAXREC_             long *maxRec                   /* out */
rVAR_NAME_               char varName[CDF_VAR_NAME_LEN+1]
                                                        /* out */
rVAR_nINDEXENTRIES_      long *numEntries               /* out */
rVAR_nINDEXLEVELS_       long *numLevels                /* out */
rVAR_nINDEXRECORDS_      long *numRecords               /* out */
rVAR_NUMallocRECS_       long *numRecords               /* out */
rVAR_NUMBER_             char *varName                  /* in  */
                         long *varNum                   /* out */
rVAR_NUMELEMS_           long *numElements              /* out */
rVAR_NUMRECS_            long *numRecords               /* out */
rVAR_PADVALUE_           void *value                    /* out */
rVAR_RECVARY_            long *recVary                  /* out */
rVAR_SEQDATA_            void *value                    /* out */
rVAR_SPARSEARRAYS_       long *sArraysType              /* out */
                         long sArraysParms[CDF_MAX_PARMS]
                                                        /* out */
                         long *sArraysPct               /* out */
rVAR_SPARSERECORDS_      long *sRecordsType             /* out */
```

```
rVARs_DIMSIZES_          long dimSizes[CDF_MAX_DIMS]    /* out */
rVARs_MAXREC_            long *maxRec                   /* out */
rVARs_NUMDIMS_           long *numDims                  /* out */
rVARs_RECDATA_           long numVars                   /* in  */
                         long varNums[]                 /* in  */
                         void *buffer                   /* out */
STATUS_TEXT_             char text[CDF_STATUSTEXT_LEN+1]
                                                        /* out */
zENTRY_DATA_             void *value                    /* out */
zENTRY_DATATYPE_         long *dataType                 /* out */
zENTRY_NUMELEMS_         long *numElements              /* out */
zVAR_ALLOCATEDFROM_      long startRecord               /* in  */
                         long *nextRecord               /* out */
zVAR_ALLOCATEDTO_        long startRecord               /* in  */
                         long *lastRecord               /* out */
zVAR_BLOCKINGFACTOR_     long *blockingFactor           /* out */
zVAR_COMPRESSION_        long *cType                    /* out */
                         long cParms[CDF_MAX_PARMS]     /* out */
                         long *cPct                     /* out */
zVAR_DATA_               void *value                    /* out */
zVAR_DATATYPE_           long *dataType                 /* out */
zVAR_DIMSIZES_           long dimSizes[CDF_MAX_DIMS]    /* out */
zVAR_DIMVARYS_           long dimVarys[CDF_MAX_DIMS]    /* out */
zVAR_HYPERDATA_          void *buffer                   /* out */
zVAR_MAXallocREC_        long *maxRec                   /* out */
zVAR_MAXREC_             long *maxRec                   /* out */
zVAR_NAME_               char varName[CDF_VAR_NAME_LEN+1]
                                                        /* out */
zVAR_nINDEXENTRIES_      long *numEntries               /* out */
zVAR_nINDEXLEVELS_       long *numLevels                /* out */
zVAR_nINDEXRECORDS_      long *numRecords               /* out */
zVAR_NUMallocRECS_       long *numRecords               /* out */
zVAR_NUMBER_             char *varName                  /* in  */
                         long *varNum                   /* out */
zVAR_NUMDIMS_            long *numDims                  /* out */
zVAR_NUMELEMS_           long *numElements              /* out */
zVAR_NUMRECS_            long *numRecords               /* out */
zVAR_PADVALUE_           void *value                    /* out */
zVAR_RECVARY_            long *recVary                  /* out */
zVAR_SEQDATA_            void *value                    /* out */
zVAR_SPARSEARRAYS_       long *sArraysType              /* out */
                         long sArraysParms[CDF_MAX_PARMS]
                                                        /* out */
                         long *sArraysPct               /* out */
zVAR_SPARSERECORDS_      long *sRecordsType             /* out */
zVARs_MAXREC_            long *maxRec                   /* out */
zVARs_RECDATA_           long numVars                   /* in  */
                         long varNums[]                 /* in  */
                         void *buffer                   /* out */
```

```
NULL_
```

```
OPEN_
           CDF_                     char *CDFname                  /* in  */
                                    CDFid *id                      /* out */

PUT_
           ATTR_NAME_               char *attrName                 /* in  */
           ATTR_SCOPE_              long scope                     /* in  */
           CDF_COMPRESSION_         long cType                     /* in  */
                                    long cParms[]                  /* in  */
           CDF_ENCODING_            long encoding                  /* in  */
           CDF_FORMAT_              long format                    /* in  */
           CDF_MAJORITY_            long majority                  /* in  */
           gENTRY_DATA_             long dataType                  /* in  */
                                    long numElements               /* in  */
                                    void *value                    /* in  */
           gENTRY_DATASPEC_         long dataType                  /* in  */
                                    long numElements               /* in  */
           rENTRY_DATA_             long dataType                  /* in  */
                                    long numElements               /* in  */
                                    void *value                    /* in  */
           rENTRY_DATASPEC_         long dataType                  /* in  */
                                    long numElements               /* in  */
           rVAR_ALLOCATEBLOCK_      long firstRecord               /* in  */
                                    long lastRecord                /* in  */
           rVAR_ALLOCATERECS_       long numRecords                /* in  */
           rVAR_BLOCKINGFACTOR_     long blockingFactor            /* in  */
           rVAR_COMPRESSION_        long cType                     /* in  */
                                    long cParms[]                  /* in  */
           rVAR_DATA_               void *value                    /* in  */
           rVAR_DATASPEC_           long dataType                  /* in  */
                                    long numElements               /* in  */
           rVAR_DIMVARYS_           long dimVarys[]                /* in  */
           rVAR_HYPERDATA_          void *buffer                   /* in  */
           rVAR_INITIALRECS_        long nRecords                  /* in  */
           rVAR_NAME_               char *varName                  /* in  */
           rVAR_PADVALUE_           void *value                    /* in  */
           rVAR_RECVARY_            long recVary                   /* in  */
           rVAR_SEQDATA_            void *value                    /* in  */
           rVAR_SPARSEARRAYS_       long sArraysType               /* in  */
                                    long sArraysParms[]            /* in  */
           rVAR_SPARSERECORDS_      long sRecordsType              /* in  */
           rVARs_RECDATA_           long numVars                   /* in  */
                                    long varNums[]                 /* in  */
                                    void *buffer                   /* in  */
           zENTRY_DATA_             long dataType                  /* in  */
                                    long numElements               /* in  */
                                    void *value                    /* in  */
           zENTRY_DATASPEC_         long dataType                  /* in  */
                                    long numElements               /* in  */
           zVAR_ALLOCATEBLOCK_      long firstRecord               /* in  */
```

```
                                    long lastRecord                /* in  */
           zVAR_ALLOCATERECS_       long numRecords                /* in  */
           zVAR_BLOCKINGFACTOR_     long blockingFactor            /* in  */
           zVAR_COMPRESSION_        long cType                     /* in  */
                                    long cParms[]                  /* in  */
           zVAR_DATA_               void *value                    /* in  */
           zVAR_DATASPEC_           long dataType                  /* in  */
                                    long numElements               /* in  */
           zVAR_DIMVARYS_           long dimVarys[]                /* in  */
           zVAR_INITIALRECS_        long nRecords                  /* in  */
           zVAR_HYPERDATA_          void *buffer                   /* in  */
           zVAR_NAME_               char *varName                  /* in  */
           zVAR_PADVALUE_           void *value                    /* in  */
           zVAR_RECVARY_            long recVary                   /* in  */
           zVAR_SEQDATA_            void *value                    /* in  */
           zVAR_SPARSEARRAYS_       long sArraysType               /* in  */
                                    long sArraysParms[]            /* in  */
           zVAR_SPARSERECORDS_      long sRecordsType              /* in  */
           zVARs_RECDATA_           long numVars                   /* in  */
                                    long varNums[]                 /* in  */
                                    void *buffer                   /* in  */

SELECT_
           ATTR_                    long attrNum                   /* in  */
           ATTR_NAME_               char *attrName                 /* in  */
           CDF_                     CDFid id                       /* in  */
           CDF_CACHESIZE_           long numBuffers                /* in  */
           CDF_DECODING_            long decoding                  /* in  */
           CDF_NEGtoPOSfp0_MODE_    long mode                      /* in  */
           CDF_READONLY_MODE_       long mode                      /* in  */
           CDF_SCRATCHDIR_          char *dirPath                  /* in  */
           CDF_STATUS_              CDFstatus status               /* in  */
           CDF_zMODE_               long mode                      /* in  */
           COMPRESS_CACHESIZE_      long numBuffers                /* in  */
           gENTRY_                  long entryNum                  /* in  */
           rENTRY_                  long entryNum                  /* in  */
           rENTRY_NAME_             char *varName                  /* in  */
           rVAR_                    long varNum                    /* in  */
           rVAR_CACHESIZE_          long numBuffers                /* in  */
           rVAR_NAME_               char *varName                  /* in  */
           rVAR_RESERVEPERCENT_     long percent                   /* in  */
           rVAR_SEQPOS_             long recNum                    /* in  */
                                    long indices[]                 /* in  */
           rVARs_CACHESIZE_         long numBuffers                /* in  */
           rVARs_DIMCOUNTS_         long counts[]                  /* in  */
           rVARs_DIMINDICES_        long indices[]                 /* in  */
           rVARs_DIMINTERVALS_      long intervals[]               /* in  */
           rVARs_RECCOUNT_          long recCount                  /* in  */
           rVARs_RECINTERVAL_       long recInterval               /* in  */
           rVARs_RECNUMBER_         long recNum                    /* in  */
           STAGE_CACHESIZE_         long numBuffers                /* in  */
```

```
              zENTRY_                  long entryNum                  /* in  */
              zENTRY_NAME_             char *varName                  /* in  */
              zVAR_                    long varNum                    /* in  */
              zVAR_CACHESIZE_          long numBuffers                /* in  */
              zVAR_DIMCOUNTS_          long counts[]                  /* in  */
              zVAR_DIMINDICES_         long indices[]                 /* in  */
              zVAR_DIMINTERVALS_       long intervals[]               /* in  */
              zVAR_NAME_               char *varName                  /* in  */
              zVAR_RECCOUNT_           long recCount                  /* in  */
              zVAR_RECINTERVAL_        long recInterval               /* in  */
              zVAR_RECNUMBER_          long recNum                    /* in  */
              zVAR_RESERVEPERCENT_     long percent                   /* in  */
              zVAR_SEQPOS_             long recNum                    /* in  */
                                       long indices[]                 /* in  */
              zVARs_CACHESIZE_         long numBuffers                /* in  */
              zVARs_RECNUMBER_         long recNum                    /* in  */
```

B.3 EPOCH Utility Routines

```
double computeEPOCH (year, month, day, hour, minute, second, msec)
long year;                                                            /* in  */
long month;                                                           /* in  */
long day;                                                             /* in  */
long hour;                                                            /* in  */
long minute;                                                          /* in  */
long second;                                                          /* in  */
long msec;                                                            /* in  */

void EPOCHbreakdown (epoch, year, month, day, hour, minute, second, msec)
double epoch;                                                         /* in  */
long *year;                                                           /* out */
long *month;                                                          /* out */
long *day;                                                            /* out */
long *hour;                                                           /* out */
long *minute;                                                         /* out */
long *second;                                                         /* out */
long *msec;                                                           /* out */

void encodeEPOCH (epoch, epString)
double epoch;                                                         /* in  */
char epString[EPOCH_STRING_LEN+1];                                    /* out */

void encodeEPOCH1 (epoch, epString)
double epoch;                                                         /* in  */
char epString[EPOCH1_STRING_LEN+1];                                   /* out */

void encodeEPOCH2 (epoch, epString)
double epoch;                                                         /* in  */
char epString[EPOCH2_STRING_LEN+1];                                   /* out */
```

```
void encodeEPOCH3 (epoch, epString)
double epoch;                                                         /* in  */
char epString[EPOCH3_STRING_LEN+1];                                   /* out */

void encodeEPOCHx (epoch, format, epString)
double epoch;                                                         /* in  */
char format[EPOCHx_FORMAT_MAX+1];                                     /* in  */
char epString[EPOCHx_STRING_MAX+1];                                   /* out */

double parseEPOCH (epString)
char epString[EPOCH_STRING_LEN+1];                                    /* in  */

double parseEPOCH1 (epString)
char epString[EPOCH1_STRING_LEN+1];                                   /* in  */

double parseEPOCH2 (epString)
char epString[EPOCH2_STRING_LEN+1];                                   /* in  */

double parseEPOCH3 (epString)
char epString[EPOCH3_STRING_LEN+1];                                   /* in  */
```

Index