CDF
Fortran 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 program, subroutine, or function that calls the CDF library or references CDF parameters must include one or more CDF include files. On VMS systems a logical name, `CDF$INC`, that specifies the location of the CDF include files is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_INC`, that serves the same purpose is defined in the definitions file `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.

On VMS and UNIX systems the following line would be included at/near the top of each routine:

```
INCLUDE '<inc-path>cdf.inc'
```

where `<inc-path>` is the file name of the directory containing `cdf.inc`. On VMS systems `CDF$INC:` may be used for `<inc-path>`. On UNIX systems `<inc-path>` must be a relative or absolute file name. (An environment variable may not be used.) Another option would be to create a symbolic link to `cdf.inc` (using `ln -s`) making `cdf.inc` appear to be in the same directory as the source file to be compiled. In that case specifying `<inc-path>` would not be necessary. On UNIX systems you will need to know where on your system `cdf.inc` has been installed.

The `cdf.inc` include file declares the `FUNCTION`s available in the CDF library (`CDF_var_num`, `CDF_lib`, etc.). Some Fortran compilers will display warning messages about unused variables if these functions are not used in a routine (because they will be assumed to be variables not function declarations). Most of these Fortran compilers have a command line option (e.g., `-nounused`) which will suppress these warning messages. If a suitable command line option is not available (and the messages are too annoying to ignore), the function declarations could be removed from `cdf.inc` but be sure to declare each CDF function that a routine uses.[1]

Microsoft Fortran

On MS-DOS systems using Microsoft Fortran the following lines would be included at/near the top of each routine/source file:

[1] Removing the function declarations from `cdf.inc` should be avoided if possible.

```
INCLUDE 'cdfmsf.inc'
.
. (PROGRAM, SUBROUTINE, or FUNCTION statement)
.
INCLUDE 'cdf.inc'
```

The include file `cdfmsf.inc` contains an `INTERFACE` statement for each subroutine/function in the CDF library. Including this file is absolutely essential if you are using the Internal Interface (`CDF_lib`). `cdfmsf.inc` is located in the same directory as `cdf.inc`.

NOTE: There are limitations on where `cdfmsf.inc` can be included. It must generally be included before the `PROGRAM`, `SUBROUTINE`, or `FUNCTION` statement of a routine. If a source file contains multiple routines, `cdfmsf.inc` only needs to be included once (at the very top of the source file). `cdf.inc`, however, may need to be included inside each routine.

On Macintosh systems using Macintosh Programmer's Workshop (MPW) Fortran the following line would be included at/near the top of each routine:

```
INCLUDE 'cdf.inc'
```

The location of `cdf.inc` will be made known to MPW Fortran when the routine is compiled.

1.1 VMS/OpenVMS Systems

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

```
$ FORTRAN <source-name>
```

where `<source-name>` is the name of the source file being compiled. (The `.FOR` extension is not necessary.) The object module created will be named `<source-name>.OBJ`.

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

1.2 UNIX Systems

An example of the command to compile a source file on UNIX flavored systems would be as follows:[2]

```
% f77 -c <source-name>.f
```

[2]The name of the Fortran compiler may be different depending on the flavor of UNIX being used.

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

The `-c` option specifies that only an object module is to be produced. (The link step is described in Chapter 2.) The object module created will be named `<source-name>.o`.

1.3 MS-DOS Systems, Microsoft Fortran

NOTE: Even though your application is written in Fortran and compiled with a Fortran compiler, compatible C run-time system libraries will be necessary to successfully link your application. This is because the CDF library is written in C and calls C run-time system functions.

An example of the command to compile a source file on MS-DOS systems using Microsoft Fortran would be as follows:[3]

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

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

The `/c` option specifies that only an object module is to be produced. (The link step is described in Chapter 2.) 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 Fortran 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.

NOTE: The CDF library is written in C. The Fortran compiler used to compile your applications must be compatible with the C compiler used to build the CDF library. (Microsoft C and Microsoft Fortran have been shown in the examples in this document.) The linker used must also be configured to allow a Fortran application to call C routines (in the CDF library). Your Fortran applications, however, should not be concerned with calling functions written in C. (They can assume that they are calling Fortran.) The CDF library is written to handle the default Fortran calling conventions.

[3] This example assumes you have properly set the MS-DOS environment variables used by the Microsoft Fortran compiler and linker. It is also assumed that the environment variables are set such that the linker will be able to find both the Fortran and C run-time system libraries that are needed.

1.4 Macintosh Systems, MPW Fortran

Macintosh Programmer's Workshop (MPW) Fortran 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 Fortran would be as follows:

```
Fortran -i <inc-path> <source-name>.f
```

where `<source-name>.f` is the name of the source file being compiled and `<inc-path>` is an absolute or relative file name of the folder containing `cdf.inc`. You will need to know where on your system `cdf.inc` 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>.f.o` in the current directory. Note that this example also assumes that `<source-name>.f` is in the current directory.

Chapter 2

Linking

Your applications must be linked with the CDF library.[1] Both the Standard and Internal interfaces for Fortran applications are built into the CDF library. On VMS systems a logical name, `CDF$LIB`, that specifies the location of the CDF library is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_LIB`, that serves the same purpose is defined in the definitions file `definitions.<shell-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 used to link an application to the CDF library (`LIBCDF.OLB`) on a VAX/VMS or VAX/OpenVMS system would be as follows:

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

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

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 Windows applications. Consult the Microsoft documentation for details on using a DLL. Note that the DLL for Microsoft is created using Microsoft 7.00.

2.2 DEC Alpha/OpenVMS Systems

An example of the command used to link an application to the CDF library (`LIBCDF.OLB`) on a DEC Alpha/OpenVMS system would be as follows:

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

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

2.3 UNIX Systems

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

```
% f77 <object-file(s)>.o ${CDF_LIB}/libcdf.a
```

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

2.3.1 Combining the Compile and Link

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

```
% f77 <source-file(s)>.f ${CDF_LIB}/libcdf.a
```

where `<source-file(s)>.f` is the name of the source file(s) being compiled/linked. (The `.f` extension is required.) Some UNIX systems may also require that `-lc`, `-lm`, and/or `-ldl` be specified at the end of the command line. Note that in a "makefile" where `CDF_LIB` is imported, `$(CDF_LIB)` would be specified instead of `${CDF_LIB}`.

2.4 MS-DOS Systems, Microsoft Fortran

NOTE: Even though your application is written in Fortran and compiled with a Fortran compiler, compatible C run-time system libraries (as supplied with Microsoft C) will be necessary to successfully link your application. This is because the CDF library is written in C and calls C run-time system functions.

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

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

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 (`.exe` will be appended by default); and `<lib-path>` is the file name of the directory containing `LIBCDF.LIB`. You will need to know where on your system `LIBCDF.LIB` has been installed. `<lib-path>` may be either an absolute or relative file name.

A map file is created by default unless the special name `nul.map` is used (as shown). If a map file is desired, the map file parameter should be omitted (in which case the name of the map file will be the name part of the executable file with `.map` appended), or a map file should be explicitly specified.

The `/NOE` option specifies that the linker should not search extended dictionaries of library symbols. This is necessary to suppress errors that would be generated because of multiply defined symbols between the Microsoft Fortran and Microsoft C system libraries.

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`, and the Fortran run-time library shown, `LLIBFORE`, assume 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 (immediately after `LLIBCE`) to handle the function naming differences between the Microsoft C 6.00 and Microsoft C 7.00 run-time libraries. **NOTE:** Specify the libraries in the order shown or errors involving multiply defined symbols may result.

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 Fortran 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] This example assumes you have properly set the MS-DOS environment variables (e.g., `INCLUDE` and `LIB`) used by the Microsoft Fortran (and 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 MS-DOS 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`.

2.5 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>.f.o <object-file>.f.o ... <object-file>.f.o δ
     <lib-path>libcdf.o δ
     "{FLibraries}"FORTRANlib.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>.cf.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>.f.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 "{FLibraries}", "{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 shared 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`. (You must specify a VAX C run-time library [RTL] because the CDF library is written in C.) The name of the executable created will be the name part of the first object file listed with `.EXE` appended. A different executable name may be specified by using the `/EXECUTABLE` qualifier.

NOTE: On 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)

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lm -ldl
```

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

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lc -lm
```

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

```
% f77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.sl
```

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

3.6 IBM RS6000 (AIX)

```
% f77 -o <exe-file> <object-file(s)>.o -L${CDF_LIB} ${CDF_LIB}/libcdf.o -lc -lm
```

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

```
% f77 -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.

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}.

Chapter 4

Programming Interface

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

4.1 Argument Passing

The CDF library is written entirely in C. Most computer systems have Fortran and C compilers that allow a Fortran application to call a C function without being concerned that different programming languages are involved. The CDF library takes advantage of the mechanisms provided by these compilers so that your Fortran application can appear to be calling another Fortran subroutine/function (in actuality the CDF library written in C). Pass all arguments exactly as shown in the description of each CDF function. This includes character strings (i.e., `%REF(...)` is not required). Be aware, however, that trailing blanks on variable and attribute names will be considered as part of the name. If the trailing blanks are not desired, pass only the part of the character string containing the name (e.g., `VAR_NAME(1:8)`).

NOTE: Unfortunately, the Microsoft C and Microsoft Fortran compilers on the IBM PC and the C and Fortran compilers on the NeXT computer do not provide the needed mechanism to pass character strings from Fortran to C without explicitly `NUL` terminating the strings. Your Fortran application must place an ASCII `NUL` character after the last character of a CDF, variable, or attribute name. An example of this follows:

```
.
.
CHARACTER ATTR_NAME*9              ! Attribute name
.
.
ATTR_NAME(1:8) = 'VALIDMIN'        ! Actual attribute name
ATTR_NAME(9:9) = CHAR(0)           ! ASCII NUL character
.
.
```

CHAR(0) is an intrinsic Fortran function that returns the ASCII character for the numerical value passed in (0 is the numerical value for an ASCII NUL character). ATTR_NAME could then be passed to one of the CDF library functions.

When the CDF library passes out a character string on an IBM PC (using the Microsoft compilers) or on a NeXT computer, the number of characters written will be exactly as shown in the description of the function called. You must declare your Fortran variable to be exactly that size.

4.2 Item Referencing

For Fortran applications, all items are referenced starting at one (1). These include variable, attribute, and attribute entry numbers, record numbers, dimensions, and dimension indices. Note that both rVariables and zVariables are numbered starting at one (1).

4.3 Status Code Constants

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

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).

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 Fortran applications the compiler defined majority for arrays is column major. The first dimension of multidimensional arrays varies the fastest 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] Disabled for PC running 16-bit DOS/Windows 3.x.

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.

`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

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

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.
`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 `.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 system being used (including logical names on VMS systems and environment variables on UNIX systems).
`CDF_VAR_NAME_LEN`	Maximum length of a variable name.
`CDF_ATTR_NAME_LEN`	Maximum length of an attribute name.
`CDF_COPYRIGHT_LEN`	Maximum length of the CDF copyright text.
`CDF_STATUSTEXT_LEN`	Maximum length of the explanation text for a status code.

Chapter 5

Standard Interface

The following sections describe the Standard Interface routines callable from Fortran applications. Most routines return a status code of type INTEGER*4 (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.

5.1 CDF_create

```
SUBROUTINE CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                      id, status)

CHARACTER CDF_name*(*)        ! in  -- CDF file name.
INTEGER*4 num_dims            ! in  -- Number of dimensions, rVariables.
INTEGER*4 dim_sizes(*)        ! in  -- Dimension sizes, rVariables.
INTEGER*4 encoding            ! in  -- Data encoding.
INTEGER*4 majority            ! in  -- Variable majority.
INTEGER*4 id                  ! out -- CDF identifier.
INTEGER*4 status              ! out -- Completion status.
```

CDF_create creates a CDF as defined by the arguments. A CDF cannot be created if it already exists. (The existing CDF will not be overwritten.) If you want to overwrite an existing CDF, you must first open it with CDF_open, delete it with CDF_delete, and then recreate it with CDF_create. If the existing CDF is corrupted, the call to CDF_open will fail. (An error code will be returned.) In this case you must delete the CDF at the command line. Delete the 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 CDF_create are defined as follows:

CDF_name
: The file name of the CDF to create. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters. A CDF file name may contain disk and directory specifications that conform to the conventions of the operating

system being used (including logical names on VMS systems and environment variables on UNIX systems).

UNIX: File names are case-sensitive.

`num_dims` Number of dimensions the rVariables in the CDF are to have. This may be as few as zero (0) and at most `CDF_MAX_DIMS`.

`dim_sizes` The size of each dimension. Each element of `dim_sizes` specifies the corresponding dimension size. Each size must be greater than zero (0). If there are zero (0) dimensions, this argument is ignored (but 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.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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

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

5.1.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 num_dims          ! Number of dimensions, rVariables.
INTEGER*4 dim_sizes(3)      ! Dimension sizes, rVariables.
INTEGER*4 majority          ! Variable majority.

DATA num_dims/3/, dim_sizes/180,360,10/, majority/ROW_MAJOR/
.
.
CALL CDF_create ('test1', num_dims, dim_sizes, NETWORK_ENCODING,
1                 majority, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
```

```
.
.
```

ROW_MAJOR and NETWORK_ENCODING are defined in cdf.inc.

5.2 CDF_open

```
SUBROUTINE CDF_open (CDF_name, id, status)

CHARACTER CDF_name*(*)        ! in  -- CDF file name.
INTEGER*4 id                  ! out -- CDF identifier.
INTEGER*4 status              ! out -- Completion status.
```

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

The arguments to CDF_open are defined as follows:

CDF_name	The file name of the CDF to open. (Do not specify an extension.) This may be at most CDF_PATHNAME_LEN characters. A CDF file name may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on 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.
status	The completion status code. Chapter 7 explains how to interpret status codes.

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

5.2.1 Example(s)

The following example will open a CDF named NOAA1.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
```

```
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 status                        ! Returned status code.
CHARACTER CDF_name*(CDF_PATHNAME_LEN)  ! File name of CDF.

DATA CDF_name/'NOAA1'/
.
.
CALL CDF_open (CDF_name, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.3 CDF_doc

```
SUBROUTINE CDF_doc (id, version, release, copyright, status)

INTEGER*4 id,                          ! in  -- CDF identifier.
INTEGER*4 version,                     ! out -- Version number.
INTEGER*4 release,                     ! out -- Release number.
CHARACTER copyright*(CDF_DOCUMENT_LEN), ! out -- Copyright.
INTEGER*4 status                       ! out -- Returned status code.
```

`CDF_doc` is used to inquire general documentation about a CDF. The version/release of the CDF that which created the CDF is provided (e.g., CDF V2.3 is version 2, release 3) along with the CDF copyright notice.

The arguments to `CDF_doc` are defined as follows:

`id` — The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.

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

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

`copyright` — The copyright notice of the CDF library that created the CDF. This character string must be large enough to hold `CDF_COPYRIGHT_LEN` characters and will be blank padded if necessary. This string will contain a linefeed character after each line of the copyright notice. (It can be printed without modification but may contain many trailing blanks.)

`status` — The completion status code. Chapter 7 explains how to interpret status codes.

5.3.1 Example(s)

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

```
.
```

```
    .
    INCLUDE '<path>cdf.inc'
    .
    .
    INTEGER*4 id                          ! CDF identifier.
    INTEGER*4 status                      ! Returned status code.
    INTEGER*4 version                     ! CDF version number.
    INTEGER*4 release                     ! CDF release number.
    CHARACTER copyright*(CDF_COPYRIGHT_LEN)  ! Copyright notice.
    INTEGER*4 last_char                   ! Last character position
                                          ! actually used in the copyright.
    INTEGER*4 start_char                  ! Starting character position
                                          ! in a line of the copyright.
    CHARACTER lf*1                        ! Linefeed character.
    .
    .
    CALL CDF_doc (id, version, release, copyright, status)
    IF (status .LT. CDF_OK) THEN               ! INFO status codes ignored
      CALL UserStatusHandler (status)
    ELSE
      WRITE (6,101) version, release
101   FORMAT (' ','Version: ',I3,' Release:',I3)

      last_char = CDF_COPYRIGHT_LEN
      DO WHILE (copyright(last_char:last_char) .EQ. ' ')
        last_char = last_char - 1
      END DO

      lf = CHAR(10)

      start_char = 1
      DO i = 1, last_char
        IF (copyright(i:i) .EQ. lf) THEN
          WRITE (6,301) copyright(start_char:i-1)
301       FORMAT (' ',A)
          start_char = i + 1
        END IF
      END DO
    END IF
    .
    .
```

5.4 CDF_inquire

```
SUBROUTINE CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
1                       max_rec, num_vars, num_attrs, status)

INTEGER*4 id                       ! in  -- CDF identifier.
INTEGER*4 num_dims                 ! out -- Number of dimensions, rVariables.
```

```
INTEGER*4 dim_sizes(CDF_MAX_DIMS)  ! out -- Dimension sizes, rVariables.
INTEGER*4 encoding                 ! out -- Data encoding.
INTEGER*4 majority                 ! out -- Variable majority.
INTEGER*4 max_rec                  ! out -- Maximum record number in the CDF,
                                           rVariables.
INTEGER*4 num_vars                 ! out -- Number of rVariables in
                                           the CDF.
INTEGER*4 num_attrs                ! out -- Number of attributes in the CDF.
INTEGER*4 status                   ! out -- Completion status.
```

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

The arguments to `CDF_inquire` are defined as follows:

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`num_dims`	The number of dimensions in the CDF for the rVariables.
`dim_sizes`	The dimension sizes for the rVariables. Each element of `dim_sizes` receives the corresponding dimension size. If there are zero (0) dimensions, this argument is ignored (but must be present).
`encoding`	The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.
`majority`	The majority of the variable data. The majorities are defined in Section 4.8.
`max_rec`	The maximum record number written to an rVariable in the CDF. Note that the maximum record number written is also kept separately for each rVariable in the CDF. The value of `max_rec` is the largest of these. Some rVariables may have fewer records actually written. `CDF_lib` (Internal Interface) may be used to inquire the maximum record written for an individual rVariable (see Section 6).
`num_vars`	The number of rVariables in the CDF.
`num_attrs`	The number of attributes in the CDF.
`status`	The completion status code. Chapter 7 explains how to interpret status codes.

5.4.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
```

```
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 num_dims                  ! Number of dimensions, rVariables.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)   ! Dimension sizes, rVariables
                                    ! (allocate to allow the maximum
                                    ! number of dimensions).
INTEGER*4 encoding                  ! Data encoding.
INTEGER*4 majority                  ! Variable majority.
INTEGER*4 max_rec                   ! Maximum record number.
INTEGER*4 num_vars                  ! Number of rVariables in CDF.
INTEGER*4 num_attrs                 ! Number of attributes in CDF.
.
.
CALL CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
.                 max_rec, num_vars, num_attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.5 CDF_close

```
SUBROUTINE CDF_close (id, status)

INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 status         ! out -- Completion status.
```

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

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

The arguments to `CDF_close` are defined as follows:

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

`status` The completion status code. Chapter 7 explains how to interpret status codes.

5.5.1 Example(s)

The following example will close an open CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
.
.
CALL CDF_close (id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.6 CDF_delete

```
SUBROUTINE CDFdelete (id, status)

INTEGER*4 id              ! in  -- CDF identifier.
INTEGER*4 status          ! out -- Completion status.
```

`CDF_delete` deletes the specified CDF. The CDF files deleted include the 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 deleted it. If you have no privilege to delete the CDF file(s), they will not be deleted. If the CDF is corrupted and cannot be opened, the CDF file(s) must be deleted at the command line.

The arguments to `CDF_delete` are defined as follows:

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`status`	The completion status code. Chapter 7 explains how to interpret status codes.

5.6.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
```

```
.
.
CALL CDF_open ('test2', id, status)
IF (status .LT. CDF_OK) THEN        ! INFO status codes ignored.
  CALL UserStatusHandler (status)
ELSE
  CALL CDF_delete (id, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

5.7 CDF_error

```
SUBROUTINE CDF_error (status, message)

INTEGER*4 status                        ! in  -- Status code.
CHARACTER message*(CDF_STATUSTEXT_LEN)  ! out -- Explanation text for
                                                 the status code.
```

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

The arguments to `CDF_error` are defined as follows:

`status` The status code to check.

`message` The explanation of the status code. This character string must be large enough to hold `CDF_STATUSTEXT_LEN` characters and will be blank padded if necessary.

5.7.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                         ! CDF identifier.
INTEGER*4 status                     ! Returned status code.
CHARACTER text*(CDF_STATUSTEXT_LEN)  ! Explanation text.
INTEGER*4 last_char                  ! Last character position
                                     ! actually used in the copyright.
.
.
```

```
     CALL CDF_open ('giss_wetl', id, status)
     IF (status .LT. CDF_WARN) THEN         ! INFO and WARNING codes ignored.
       CALL CDF_error (status, text)
       last_char = CDF_STATUSTEXT_LEN
       DO WHILE (text(last_char:last_char) .EQ. ' ')
         last_char = last_char - 1
       END DO
       WRITE (6,101) text(1:last_char)
 101   FORMAT (' ','ERROR> ',A)
     END IF
     .
     .
```

5.8 CDF_attr_create

```
SUBROUTINE CDF_attr_create (id, attr_name, attr_scope, attr_num, status)
INTEGER*4 id                 ! in  -- CDF identifier.
CHARACTER attr_name*(*)      ! in  -- Attribute name.
INTEGER*4 attr_scope         ! in  -- Scope of attribute.
INTEGER*4 attr_num           ! out -- Attribute number.
INTEGER*4 status             ! out -- Completion status.
```

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

The arguments to `CDF_attr_create` are defined as follows:

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_name`	The name of the attribute to create. This may be at most `CDF_ATTR_NAME_LEN` characters. Attribute names are case-sensitive.
`attr_scope`	The scope of the new attribute. Specify one of the scopes described in Section 4.12.
`attr_num`	The number assigned to the new attribute. This number must be used in subsequent CDF function calls when referring to this attribute. An existing attribute's number may be determined with the `CDF_attr_num` function.
`status`	The completion status code. Chapter 7 explains how to interpret status codes.

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 '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
CHARACTER UNITS_attr_name*5         ! Name of "Units" attribute.
INTEGER*4 UNITS_attr_num            ! "Units" attribute number.
INTEGER*4 TITLE_attr_num            ! "TITLE" attribute number.
INTEGER*4 TITLE_attr_scope          ! "TITLE" attribute scope.

DATA UNITS_attr_name/'Units'/, TITLE_attr_scope/GLOBAL_SCOPE/
.
.
CALL CDF_attr_create (id, 'TITLE', TITLE_attr_scope, TITLE_attr_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_attr_create (id, UNITS_attr_name, VARIABLE_SCOPE, UNITS_attr_num,
1                     status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.9 CDF_attr_num

```
INTEGER*4 FUNCTION CDF_attr_num (id, attr_name)

INTEGER*4 id                ! in --  CDF id.
CHARACTER attr_name*(*)     ! in -- attribute name.
```

`CDF_attr_num` is used to determine the attribute number associated with a given attribute name. If the attribute is found, `CDF_attr_num` returns its number — which will be equal to or greater than one (1). If an error occurs (e.g., the attribute name does not exist in the CDF), an error code is returned. Error codes are less than zero (0).

The arguments to `CDF_attr_num` are defined as follows:

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_name`	The name of the attribute for which to search. This may be at most `CDF_ATTR_NAME_LEN` characters. Attribute names are case-sensitive.

`CDF_attr_num` may be used as an embedded function call when an attribute number is needed. `CDF_attr_num` is declared in `cdf.inc`. (Fortran functions must be declared so that the returned value is interpreted correctly.)

5.9.1 Example(s)

In the following example the attribute named `pressure` will be renamed to `PRESSURE` with `CDF_attr_num` being used as an embedded function call.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
.
.
CALL CDF_attr_rename (id, CDF_attr_num(id,'pressure'), 'PRESSURE', status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Note that if the attribute `pressure` did not exist in the CDF, the call to `CDF_attr_num` would have returned an error code. Passing that error code to `CDF_attr_rename` as an attribute number would have resulted in `CDF_attr_rename` also returning an error code. `CDF_attr_rename` is described in Section 5.10.

5.10 CDF_attr_rename

```
SUBROUTINE CDF_attr_rename (id, attr_num, attr_name, status)

INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 attr_num       ! in  -- Attribute number.
CHARACTER attr_name*(*)  ! in  -- New attribute name.
INTEGER*4 status         ! out -- Completion status.
```

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

The arguments to `CDF_attr_rename` are defined as follows:

`id`	The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create` or `CDF_open`.
`attr_num`	The number of the attribute to rename. This number may be determined with a call to `CDF_attr_num` (see Section 5.9).
`attr_name`	The new attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters. Attribute names are case-sensitive.
`status`	The completion status code. Chapter 7 explains how to interpret status codes.

5.10.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                       ! CDF identifier.
INTEGER*4 status                   ! Returned status code.
.
.
CALL CDF_attr_rename (id, CDF_attr_num(id,'LAT'), 'LATITUDE', status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.11 CDF_attr_inquire

```
SUBROUTINE CDF_attr_inquire (id, attr_num, attr_name, attr_scope, max_entry,
1                              status)
INTEGER*4 id                             ! in  -- CDF identifier.
INTEGER*4 attr_num                       ! in  -- Attribute number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN)  ! out -- Attribute name.
INTEGER*4 attr_scope                     ! out -- Attribute scope.
INTEGER*4 max_entry                      ! out -- Maximum gEntry or
                                         !        rEntry number.
INTEGER*4 status                         ! out -- Completion status.
```

CDF_attr_inquire is used to inquire about the specified attribute. To inquire about a specific attribute entry, use CDF_attr_entry_inquire (see Section 5.12).

The arguments to CDF_attr_inquire are defined as follows:

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

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

attr_name — The attribute's name. This character string must be large enough to hold CDF_ATTR_NAME_LEN characters and will be blank padded if necessary.

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

max_entry — 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 `CDF_lib` function (see Section 6). If no entries exist for the attribute, then a value of zero (0) will be passed back.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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 `CDF_inquire`. Note that attribute numbers start at one (1) and are consecutive.

```
   .
   .
   INCLUDE '<path>cdf.inc'
   .
   .
   INTEGER*4 id                           ! CDF identifier.
   INTEGER*4 status                       ! Returned status code.
   INTEGER*4 num_dims                     ! Number of dimensions.
   INTEGER*4 dim_sizes(CDF_MAX_DIMS)      ! Dimension sizes (allocate to
                                          ! allow the maximum number of
                                          ! dimensions).
   INTEGER*4 encoding                     ! Data encoding.
   INTEGER*4 majority                     ! Variable majority.
   INTEGER*4 max_rec                      ! Maximum record number in CDF.
   INTEGER*4 num_vars                     ! Number of variables in CDF.
   INTEGER*4 num_attrs                    ! Number of attributes in CDF.
   INTEGER*4 attr_n                       ! Attribute number.
   CHARACTER attr_name*(CDF_ATTR_NAME_LEN)  ! Attribute name.
   INTEGER*4 attr_scope                   ! Attribute scope.
   INTEGER*4 max_entry                    ! Maximum entry number.
   .
   .
   CALL CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
   1                 max_rec, num_vars, num_attrs, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

   DO attr_n = 1, num_attrs
     CALL CDF_attr_inquire (id, attr_n, attr_name, attr_scope, max_entry,
     1                      status)
     IF (status .LT. CDF_OK) THEN       ! INFO status codes ignored.
       CALL UserStatusHandler (status)
     ELSE
       WRITE (6,10) attr_name
10     FORMAT (' ',A)
     END IF
   END DO
   .
```

.

5.12 CDF_attr_entry_inquire

```
SUBROUTINE CDF_attr_entry_inquire (id, attr_num, entry_num, data_type,
1                                   num_elements, status)
INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 attr_num       ! in  -- Attribute number.
INTEGER*4 entry_num      ! in  -- Entry number.
INTEGER*4 data_type      ! out -- Data type.
INTEGER*4 num_elements   ! out -- Number of elements (of the data type).
INTEGER*4 status         ! out -- Completion status.
```

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

The arguments to `CDF_attr_entry_inquire` are defined as follows:

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

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

`entry_num` 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 entry).

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

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

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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 one (1) and the maximum entry number must exist. For this reason `NO_SUCH_ENTRY` is an expected error code. Note also that if the attribute is variable in scope, then the rEntry numbers are actually rVariable numbers.

.

```
   .
   INCLUDE '<path>cdf.inc'
   .
   .
   INTEGER*4 id                            ! CDF identifier.
   INTEGER*4 status                        ! Returned status code.
   INTEGER*4 attr_n                        ! Attribute number.
   INTEGER*4 entryN                        ! Entry number.
   CHARACTER attr_name*(CDF_ATTR_NAME_LEN) ! Attribute name.
   INTEGER*4 attr_scope                    ! Attribute scope.
   INTEGER*4 max_entry                     ! Maximum entry number used.
   INTEGER*4 data_type                     ! Data type.
   INTEGER*4 num_elems                     ! Number of elements (of the
                                           ! data type).
   .
   .
   attr_n = CDF_attr_num (id, 'TMP')
   IF (attr_n .LT. 1) CALL UserStatusHandler (attr_n)  ! If less than one (1),
                                                       ! then it must be a
                                                       ! warning/error code.

   CALL CDF_attr_inquire (id, attr_n, attr_name, attr_scope, max_entry, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

   DO entryN = 1, max_entry
     CALL CDF_attr_entry_inquire (id, attr_n, entryN, data_type, num_elems,
     1                            status)
     IF (status .LT. CDF_OK) THEN
       IF (status .NE. NO_SUCH_ENTRY) CALL UserStatusHandler (status)
     ELSE
C      (process entries)
       .
       .
     END IF
   END DO
```

5.13 CDF_attr_put

```
SUBROUTINE CDF_attr_put (id, attr_num, entry_num, data_type, num_elements,
1                        value, status)

INTEGER*4 id,              ! in --  CDF identifier.
INTEGER*4 attr_num         ! in --  Attribute number.
INTEGER*4 entry_num        ! in --  Entry number.
INTEGER*4 data_type        ! in --  Data type of the entry.
INTEGER*4 num_elements     ! in --  Number of elements (of the data type).
<type>    value            ! in --  Value (<type> depends on the data type
                           !        of the entry).
INTEGER*4 status           ! out -- Completion status.
```

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

The arguments to CDF_attr_put are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
attr_num	The attribute number. This number may be determined with a call to CDF_attr_num (see Section 5.9).
entry_num	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).
data_type	The data type of the entry. Specify one of the data types defined in Section 4.5.
num_elements	The number of elements of the data type. For character data types (CDF_CHAR and CDF_UCHAR) this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.
value	The value(s) to write. The value is written to the CDF from value. **WARNING:** If the entry has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the entry does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.
status	The completion status code. Chapter 7 explains how to interpret status codes.

5.13.1 Example(s)

The following example writes two attribute entries. The first is to gEntry number one (1) of the gAttribute TITLE. The second is to vAttribute VALIDs for the rEntry that corresponds to the rVariable TMP.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
PARAMETER TITLE_LEN = 10          ! Length of CDF title.
.
.
INTEGER*4 id                      ! CDF identifier.
INTEGER*4 status                  ! Returned status code.
INTEGER*4 entry_num               ! Entry number.
INTEGER*4 num_elements            ! Number of elements (of data type).
CHARACTER title*(TITLE_LEN)       ! Value of TITLE attribute, rEntry
                                  ! number 1.
```

```
INTEGER*2 TMPvalids(2)            ! Value(s) of VALIDs attribute,
                                  ! rEntry for rVariable TMP

DATA title/'CDF title.'/, TMPvalids/15,30/
.
.
entry_num = 1
CALL CDF_attr_put (id, CDF_attr_num(id,'TITLE'), entry_num, CDF_CHAR,
1                  TITLE_LEN, title, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
num_elements = 2
CALL CDF_attr_put (id, CDF_attr_num(id,'VALIDs'), CDF_var_num(id,'TMP'),
1                  CDF_INT2, num_elements, TMPvalids, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.14 CDF_attr_get

```
SUBROUTINE CDF_attr_get (id, attr_num, entry_num, value, status)

INTEGER*4 id             ! in  -- CDF identifier.
INTEGER*4 attr_num       ! in  -- Attribute number.
INTEGER*4 entry_num      ! in  -- Entry number.
<type>    value          ! out -- Value (<type> is dependent on
                                  the data type of the entry).
INTEGER*4 status         ! out -- Completion status.
```

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

The arguments to CDF_attr_get are defined as follows:

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

attr_num
: The attribute number. This number may be determined with a call to CDF_attr_num (see Section 5.9).

entry_num
: 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 subroutine CDF_attr_entry_inquire would be used to determine the entry data

type and number of elements (of that data type). The value is read from the CDF and placed into `value`.

WARNING: If the entry has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the entry does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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`).

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
INTEGER*4 attr_n                      ! Attribute number.
INTEGER*4 entryN                      ! Entry number.
INTEGER*4 data_type                   ! Data type.
INTEGER*4 num_elems                   ! Number of elements (of data type).
CHARACTER buffer*100                  ! Buffer to receive value (in
                                      ! this case it is assumed that 100
                                      ! characters is enough).
.
.
attr_n = CDF_attr_Num (id, 'UNITS')
IF (attr_n .LT. 0) CALL UserStatusHandler (attr_n)  ! If less than one (1),
                                                    ! then it must be a
                                                    ! warning/error code.

entryN = CDF_var_num (id, 'PRES_LVL')              ! The rEntry number is
                                                   ! the rVariable number.
IF (entryN .LT. 0) CALL UserStatusHandler (entryN) ! If less than one (1),
                                                   ! then it must be a
                                                   ! warning/error code.

CALL CDF_attr_entry_inquire (id, attr_n, entryN, data_type, num_elems,
1                            status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

IF (data_type .EQ. CDF_CHAR) THEN
  CALL CDF_attr_get (id, attr_n, entryN, buffer, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
  WRITE (6,10) buffer(1:num_elems)
```

```
10   FORMAT (' ',A)
   END IF
   .
   .
```

5.15 CDF_var_create

```
SUBROUTINE CDF_var_create (id, var_name, data_type, num_elements,
1                          rec_variance, dim_variances, var_num, status)

INTEGER*4 id                  ! in --  CDF identifier.
CHARACTER var_name*(*)        ! in --  rVariable name.
INTEGER*4 data_type           ! in --  Data type.
INTEGER*4 num_elements        ! in --  Number of elements (of the data type).
INTEGER*4 rec_variance        ! in --  Record variance.
INTEGER*4 dim_variances(*)    ! in --  Dimension variances.
INTEGER*4 var_num             ! out -- rVariable number.
INTEGER*4 status              ! out -- Completion status.
```

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

The arguments to CDF_var_create are defined as follows:

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

var_name — The name of the rVariable to create. This may be at most CDF_VAR_NAME_LEN characters. Variable names are case-sensitive.

data_type — The data type of the new rVariable. Specify one of the data types defined in Section 4.5.

num_elements — 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.

rec_variance — The variable's record variance. Specify one of the variances defined in Section 4.9.

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

var_num — The number assigned to the new rVariable. This number must be used in subsequent CDF function calls when referring to this rVariable. An existing rVariable's number may be determined with the CDF_var_num function.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

5.15.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 EPOCH_rec_vary            ! EPOCH record variance.
INTEGER*4 LAT_rec_vary              ! LAT record variance.
INTEGER*4 LON_rec_vary              ! LON record variance.
INTEGER*4 TMP_rec_vary              ! TMP record variance.
INTEGER*4 EPOCH_dim_varys(2)        ! EPOCH dimension variances.
INTEGER*4 LAT_dim_varys(2)          ! LAT dimension variances.
INTEGER*4 LON_dim_varys(2)          ! LON dimension variances.
INTEGER*4 TMP_dim_varys(2)          ! TMP dimension variances.
INTEGER*4 EPOCH_var_num             ! EPOCH variable number.
INTEGER*4 LAT_var_num               ! LAT rVariable number.
INTEGER*4 LON_var_num               ! LON rVariable number.
INTEGER*4 TMP_var_num               ! TMP rVariable number.

DATA EPOCH_rec_vary/VARY/, LAT_rec_vary/NOVARY/,
1    LON_rec_vary/NOVARY/, TMP_rec_vary/VARY/
DATA EPOCH_dim_varys/NOVARY,NOVARY/, LAT_dim_varys/NOVARY,VARY/,
1    LON_dim_varys/VARY,NOVARY/, TMP_dim_varys/VARY,VARY/
.
.
CALL CDF_var_create (id, 'EPOCH', CDF_EPOCH, 1,
1                    EPOCH_rec_vary, EPOCH_dim_varys, EPOCH_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'LATITUDE', CDF_INT2, 1,
1                    LAT_rec_vary, LAT_dim_varys, LAT_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'LONGITUDE', CDF_INT2, 1,
1                    LON_rec_vary, LON_dim_varys, LON_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_var_create (id, 'TEMPERATURE', CDF_REAL4, 1,
1                    TMP_rec_vary, TMP_dim_varys, TMP_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
```

.

5.16 CDF_var_num

```
INTEGER*4 FUNCTION CDF_var_num (id, var_name)

INTEGER*4 id                ! in -- CDF identifier.
CHARACTER var_name*(*)      ! in -- rVariable name.
```

CDF_var_num is used to determine the number associated with a given rVariable name. If the rVariable is found, CDF_var_num returns its number — which will be equal to or greater than one (1). If an error occurs (e.g., the rVariable does not exist in the CDF), an error code is returned. Error codes are less than zero (0).

The arguments to CDF_var_num are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
var_name	The name of the rVariable for which to search. This may be at most CDF_VAR_NAME_LEN characters. Variable names are case-sensitive.

CDF_var_num may be used as an embedded function call when an rVariable number is needed. CDF_var_num is declared in cdf.inc. (Fortran functions must be declared so that the returned value is interpreted correctly.)

5.16.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                          ! CDF identifier.
INTEGER*4 status                      ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN) ! rVariable name.
INTEGER*4 data_type                   ! Data type of the rVariable.
INTEGER*4 num_elements                ! Number of elements (of the
                                      ! data type).
INTEGER*4 rec_variances               ! Record variance.
INTEGER*4 dim_variances(CDF_MAX_DIMS) ! Dimension variances.
.
.
CALL CDF_var_inquire (id, CDF_var_num(id,'LATITUDE'), var_name, data_type,
1                     num_elements, rec_variance, dim_variances, status)
```

```
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

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

5.17 CDF_var_rename

```
SUBROUTINE CDF_var_rename (id, var_num, var_name, status)

INTEGER*4 id               ! in  -- CDF identifier.
INTEGER*4 var_num          ! in  -- rVariable number.
CHARACTER var_name*(*)     ! in  -- New name.
INTEGER*4 status           ! out -- Completion status.
```

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

The arguments to CDF_var_rename are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
var_num	The number of the rVariable to rename. This number may be determined with a call to CDF_var_num (see Section 5.16).
var_name	The new rVariable name. This may be at most CDF_VAR_NAME_LEN characters. Variables names are case-sensitive.
status	The completion status code. Chapter 7 explains how to interpret status codes.

5.17.1 Example(s)

In the following example the rVariable named TEMPERATURE is renamed to TMP (if it exists). Note that if CDF_var_num returns a value less than one (1), then that value is not an rVariable number but rather a warning/error code.

```
.
.
INCLUDE '<path>cdf.inc'
.
```

```
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*4 var_num                   ! rVariable number.
.
.
var_num = CDF_var_num (id, 'TEMPERATURE')
IF (var_num .LT. 1) THEN
  IF (var_num .NE. NO_SUCH_VAR) CALL UserStatusHandler (var_num)
ELSE
  CALL CDF_var_rename (id, var_num, 'TMP', status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

5.18 CDF_var_inquire

```
SUBROUTINE CDF_var_inquire (id, var_num, var_name, data_type, num_elements,
1                           rec_variance, dim_variances, status)

INTEGER*4 id                        ! in  -- CDF identifier.
INTEGER*4 var_num                   ! in  -- rVariable number.
CHARACTER var_name*(CDF_VAR_NAME_LEN) ! out -- rVariable name.
INTEGER*4 data_type                 ! out -- Data type.
INTEGER*4 num_elements              ! out -- Number of elements (of the
                                    !        data type).
INTEGER*4 rec_variance              ! out -- Record variance.
INTEGER*4 dim_variances(CDF_MAX_DIMS) ! out -- Dimension variances.
INTEGER*4 status                    ! out -- Completion status.
```

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

The arguments to `CDF_var_inquire` are defined as follows:

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

`var_num` The number of the rVariable to inquire. This number may be determined with a call to `CDF_var_num` (see Section 5.16).

`var_name` The rVariable's name. This character string must be large enough to hold `CDF_VAR_NAME_LEN` characters and will be blank padded if necessary.

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

`num_elements` 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 will always be one (1) — multiple elements at each value are not allowed for non-character data types.

rec_variance — The record variance. The record variances are defined in Section 4.9.

dim_variances — The dimension variances. Each element of dim_variances receives the corresponding dimension variance. The dimension variances are defined in Section 4.9. For 0-dimensional rVariable this argument is ignored (but must be present).

status — The completion status code. Chapter 7 explains how to interpret status codes.

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 CDF_var_inquire will be the same as that passed in to CDF_var_num.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 status                        ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN)   ! rVariable name.
INTEGER*4 data_type                     ! Data type.
INTEGER*4 num_elems                     ! Number of elements (of data type).
INTEGER*4 rec_vary                      ! Record variance.
INTEGER*4 dim_varys(CDF_MAX_DIMS)       ! Dimension variances (allocate
                                        ! to allow the maximum number of
                                        ! dimensions).
.
.
CALL CDF_var_inquire (id, CDF_var_num(id,'HEAT_FLUX'), var_name, data_type,
1                     num_elems, rec_vary, dim_varys, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

5.19 CDF_var_put

```
SUBROUTINE CDF_var_put (id, var_num, rec_num, indices, value, status)

INTEGER*4 id              ! in -- CDF identifier.
INTEGER*4 var_num         ! in -- rVariable number.
INTEGER*4 rec_num         ! in -- Record number.
INTEGER*4 indices(*)      ! in -- Dimension indices.
```

```
<type>    value         ! in -- Value. (<type> depends on
                        !       the data type of the rVariable).
INTEGER*4 status        ! out -- Completion status.
```

CDF_var_put is used to write a single value to an rVariable. CDF_var_hyper_put may be used to write more than one rVariable value with a single call.

The arguments to CDF_var_put are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
var_num	The number of the rVariable to which to write. This number may be determined with a call to CDF_var_num (see Section 5.16).
rec_num	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 value.
	WARNING: If the rVariable has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the rVariable does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.
status	The completion status code. Chapter 7 explains how to interpret status codes.

5.19.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                    ! CDF identifier.
INTEGER*4 status                ! Returned status code.
INTEGER*2 lat                   ! Latitude value.
INTEGER*4 var_n                 ! rVariable number.
INTEGER*4 rec_num               ! Record number.
INTEGER*4 indices(2)            ! Dimension indices.

DATA rec_num/1/, indices/1,1/
.
```

```
.
var_n = CDF_var_num (id, 'LATITUDE')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then not an rVariable
                                                  ! number but rather a
                                                  ! warning/error code.
DO lat = -90, 90
   indices(2) = 91 + lat
   CALL CDF_var_put (id, var_n, rec_num, indices, lat, status)
   IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END DO
.
.
```

Since the record variance is `NOVARY`, the record number (`rec_num`) is set to one (1). Also note that because the dimension variances are `[NOVARY,VARY]`, only the second dimension is varied as values are written. (The values are "virtually" the same at each index of the first dimension.)

5.20 CDF_var_get

```
SUBROUTINE CDF_var_get (id, var_num, rec_num, indices, value, status)

INTEGER*4 id            ! in  -- CDF identifier.
INTEGER*4 var_num       ! in  -- rVariable number.
INTEGER*4 rec_num       ! in  -- Record number.
INTEGER*4 indices(*)    ! in  -- Dimension indices.
<type>    value         ! out -- Value (<type> depends on
                        !        the data type of the rVariable).
INTEGER*4 status        ! out -- Completion status.
```

`CDF_var_get` is used to read a single value from an rVariable. `CDF_var_hyper_get` may be used to read more than one rVariable value with a single call.

The arguments to `CDF_var_get` are defined as follows:

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

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

`rec_num` 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. For 0-dimensional rVariables this argument is ignored (but must be present).

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

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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 the variable the record variance is `VARY`; the dimension variances are `[VARY,VARY,VARY]`; and the data type is `CDF_REAL4`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
REAL*4    tmp(180,91,10)            ! Temperature values.
INTEGER*4 indices(3)                ! Dimension indices.
INTEGER*4 var_n                     ! rVariable number.
INTEGER*4 rec_num                   ! Record number.
INTEGER*4 d1, d2, d3                ! Dimension index values.
.
.
var_n = CDF_var_num (id, 'Temperature')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then it is actually a
                                                  ! warning/error code.

rec_num = 13

DO d1 = 1, 180
  indices(1) = d1
  DO d2 = 1, 91
    indices(2) = d2
    DO d3 = 1, 10
      indices(3) = d3
      CALL CDF_var_get (id, var_n, rec_num, indices, tmp(d1,d2,d3), status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    END DO
  END DO
END DO
.
.
```

5.21 CDF_var_hyper_put

```
SUBROUTINE CDF_var_hyper_put (id, var_num, rec_start, rec_count, rec_interval,
1                              indices, counts, intervals, buffer, status)

INTEGER*4 id               ! in -- CDF identifier.
INTEGER*4 var_num          ! in -- rVariable number.
INTEGER*4 rec_start        ! in -- Starting record number.
INTEGER*4 rec_count        ! in -- Number of records.
INTEGER*4 rec_interval     ! in -- Interval between records.
INTEGER*4 indices(*)       ! in -- Dimension indices of starting value.
INTEGER*4 counts(*)        ! in -- Number of values along each dimension.
INTEGER*4 intervals(*)     ! in -- Interval between values along each
                           !       dimension.
<type>    buffer           ! in -- Buffer of values (<type> depends
                           !       on the data type of the rVariable.)
INTEGER*4 status           ! out -- Completion status.
```

`CDF_var_hyper_put` is used to write a buffer of one or more values to an rVariable. It is important to know the variable majority of the CDF before using `CDF_var_hyper_put` because the values in the buffer to be written must be in the same majority. `CDF_inquire` is used to determine the variable majority of a CDF. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to `CDF_var_hyper_put` are defined as follows:

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

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

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

`rec_count` The number of records to write.

`rec_interval` The interval between records for subsampling[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. For 0-dimensional rVariables this argument is ignored (but must be present).

`counts` The number of values along each dimension to write. Each element of `counts` specifies the corresponding dimension count. For 0-dimensional rVariables 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). Each element of `intervals` specifies the corresponding dimension interval. For 0-dimensional rVariables this argument is ignored (but must be present).

[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 in `buffer` are written to the CDF.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `buffer` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `buffer` must NOT be a `CHARACTER` Fortran variable.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

5.21.1 Example(s)

The following example writes values to the rVariable `LATITUDE` in a CDF whose rVariables are 2-dimensional and have dimension sizes `[360,181]`. For `LATITUDE` the record variance is `NOVARY`; the dimension variances 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 `CDF_var_hyper_put` rather than numerous calls to `CDF_var_put`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
INTEGER*2 lat                       ! Latitude value.
INTEGER*2 lats(181)                 ! Buffer of latitude values.
INTEGER*4 var_n                     ! rVariable number.
INTEGER*4 rec_start                 ! Record number.
INTEGER*4 rec_count                 ! Record counts.
INTEGER*4 rec_interval              ! Record interval.
INTEGER*4 indices(2)                ! Dimension indices.
INTEGER*4 counts(2)                 ! Dimension counts.
INTEGER*4 intervals(2)              ! Dimension intervals.

DATA rec_start/1/, rec_count/1/, rec_interval/1/,
1    indices/1,1/, counts/1,181/, intervals/1,1/
.
.
var_n = CDF_var_num (id, 'LATITUDE')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then not an rVariable
                                                  ! number but rather a
                                                  ! warning/error code

DO lat = -90, 90
  lats(91+lat) = lat
END DO

CALL CDF_var_hyper_put (id, var_n, rec_start, rec_count, rec_interval,
1                       indices, counts, intervals, lats, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
```

.
.

5.22 CDF_var_hyper_get

```
SUBROUTINE CDF_var_hyper_get (id, var_num, rec_start, rec_count, rec_interval,
1                             indices, counts, intervals, buffer, status)

INTEGER*4 id                ! in -- CDF identifier.
INTEGER*4 var_num           ! in -- rVariable number.
INTEGER*4 rec_start         ! in -- Starting record number.
INTEGER*4 rec_count         ! in -- Number of records.
INTEGER*4 rec_interval      ! in -- Subsampling interval between records.
INTEGER*4 indices(*)        ! in -- Dimension indices of starting value.
INTEGER*4 counts(*)         ! in -- Number of values along each dimension.
INTEGER*4 intervals(*)      ! in -- Subsampling intervals along each dimension.
<type>    buffer            ! out -- Buffer of values (<type> depends
                            !        on the data type of the rVariable).
INTEGER*4 status            ! out -- Completion status.
```

CDF_var_hyper_get is used to read a buffer of one or more values from an rVariable. It is important to know the variable majority of the CDF before using CDF_var_hyper_get because the values placed into the buffer will be in that majority. CDF_var_inquire is used to determine the variable majority of a CDF. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to CDF_var_hyper_get are defined as follows:

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

var_num
: The number of the rVariable from which to read. This number may be determined with a call to CDF_var_num (see Section 5.16).

rec_start
: The record number at which to start reading.

rec_count
: The number of records to read.

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

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

counts
: The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. For 0-dimensional rVariables 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. For 0-dimensional rVariables 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 enough to hold the values. The values are read from the CDF and placed into `buffer`. `CDF_var_inquire` would be used to determine the variable's data type and number of elements (of that data type) at each value.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `buffer` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `buffer` must NOT be a `CHARACTER` Fortran variable.

`status` The completion status code. Chapter 7 explains how to interpret status codes.

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 the CDF's variable majority is `COLUMN_MAJOR`. For the variable the record variance is `VARY`; the dimension variances are `[VARY,VARY,VARY]`; and the data type is `CDF_REAL4`. This example is similar to the example in Section 5.20 except that it uses a single call to `CDF_var_hyper_get` rather than numerous calls to `CDF_var_get`.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                        ! CDF identifier.
INTEGER*4 status                    ! Returned status code.
REAL*4    tmp(180,91,10)            ! Temperature values.
INTEGER*4 var_n                     ! rVariable number.
INTEGER*4 rec_start                 ! Record number.
INTEGER*4 rec_count                 ! Record counts.
INTEGER*4 rec_interval              ! Record interval.
INTEGER*4 indices(3)                ! Dimension indices.
INTEGER*4 counts(3)                 ! Dimension counts.
INTEGER*4 intervals(3)              ! Dimension intervals.

DATA rec_start/13/, rec_count/1/, rec_interval/1/,
1    indices/1,1,1/, counts/180,91,10/, intervals/1,1,1/
.
.
var_n = CDF_var_num (id, 'Temperature')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n)  ! If less than one (1),
                                                  ! then it is actually a
                                                  ! warning/error code.

CALL CDF_var_hyper_get (id, var_n, rec_start, rec_count, rec_interval,
```

```
1                        indices, counts, intervals, tmp, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Note that if the CDF's variable majority had been ROW_MAJOR, the tmp array would have been declared REAL*4 tmp(10,91,180) for proper indexing.

5.23 CDF_var_close

```
SUBROUTINE CDF_var_close (id, var_num, status)

INTEGER*4 id                ! in -- CDF identifier.
INTEGER*4 var_num           ! in -- rVariable number.
INTEGER*4 status            ! out -- Completion status.
```

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

- id — The identifier of the CDF. This identifier must have been initialized by a call to CDF_create or CDF_open.
- var_num — The number of the rVariable to close. This number may be determined with a call to CDF_var_num (see Section 5.16).
- status — The completion status code. Chapter 7 explains how to interpret status codes.

5.23.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
```

```
.
INTEGER*4 id                           ! CDF identifier.
INTEGER*4 status                       ! Returned status code.
.
.
CALL CDF_var_close (id, CDF_var_num(id,'Flux'), status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Chapter 6

Internal Interface — CDF_lib

The Internal Interface consists of only one routine, `CDF_lib`.[1] `CDF_lib` can be used to perform all possible operations on a CDF. In fact, all of the Standard Interface functions are implemented using the Internal Interface. `CDF_lib` must be used to perform operations not possible with the Standard Interface functions. These operations would involve CDF features added after the Standard Interface functions had been defined (e.g., specifying a single-file format for a CDF, accessing zVariables, or specifying a pad value for an rVariable or zVariable). Note that `CDF_lib` can also be used to perform certain operations more efficiently than with the Standard Interface functions.

`CDF_lib` takes a variable number of arguments that specify one or more operations to be performed (e.g., opening a CDF, creating an attribute, or writing a variable value). The operations are performed according to the order of the arguments. Each operation 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.

6.1 Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                      ! CDF identifier.
INTEGER*4 status                  ! Returned status code.
CHARACTER CDF_name*5              ! Name of the CDF.
INTEGER*4 num_dims                ! Number of dimensions.
```

[1] See Section 6.5.1 for an ugly exception to this.

```
INTEGER*4 dim_sizes(2)              ! Dimension sizes.
INTEGER*4 encoding                  ! Data encoding.
INTEGER*4 majority                  ! Variable data majority.
INTEGER*4 format                    ! Format of CDF.

DATA CDF_name/'test1'/, num_dims/2/, dim_sizes/100,200/,
1    encoding/HOST_ENCODING/, majority/ROW_MAJOR/, format/SINGLE_FILE/
.
.
CALL CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

status = CDF_lib (PUT_, CDF_FORMAT_, format,
2                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

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

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

`PUT_` The first 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.inc` (as are all CDF constants). It was not necessary to select a current CDF since the call to `CDF_create` implicitly selected the CDF created as the current CDF.[2] This is the case since all of the Standard Interface functions actually call the Internal Interface to perform their operations.

`CDF_FORMAT_` The item to be put. In this case it is the CDF's format. One required argument for this operation follows.

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

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

`status` The completion status code. Note that `CDF_lib` also returns the completion status code.[3] Chapter 7 explains how to interpret status codes.

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

[3] Section 6.5 explains why it does both.

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

```
.
.
status = CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id,
1                 PUT_, CDF_ENCODING_, encoding,
2                       CDF_MAJORITY_, majority,
3                       CDF_FORMAT_, format,
4                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

The purpose of each argument is as follows.

`CREATE_`	The first 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 argument that must follow. When a CDF is created (with `CDF_lib`), the format, encoding, and majority default to values specified when your CDF distribution was built and installed. Consult your system manager for these defaults.
`CDF_name`	The file name of the CDF.
`num_dims`	The number of dimensions in the CDF.
`dim_sizes`	The dimension sizes.
`id`	The identifier to be used when referencing the created CDF in subsequent operations.
`PUT_`	This argument could have been one of two things. Another item to create or a new function to perform. In this case it is another function to perform — something will be put into the CDF.
`CDF_ENCODING_`	The item to be put — in this case the CDF's encoding. Note that the CDF did not have to be selected. It was implicitly selected as the current CDF when it was created. One required argument follows.
`encoding`	The encoding to be put to the CDF.
`CDF_MAJORITY_`	This argument could have been one of two things. Another item to put or a new function to perform. In this case it is another item to put — the CDF's majority. One required argument follows.
`majority`	The majority to be put to the CDF.
`CDF_FORMAT_`	Once again this argument could have been either another item to put or a new function to perform. It is another item to put — the CDF's format. One required argument follows.
`format`	The format to be put to the CDF.

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

`status` The completion status code. Note that `CDF_lib` also returns the completion status code. Chapter 7 explains how to interpret status codes.

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

6.2 Current Objects/States (Items)

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

CDF (object)

A CDF operation is always performed on the current CDF. The current CDF is implicitly selected whenever a CDF is opened or created. The current CDF may be explicitly selected using the `<SELECT_,CDF_>`[4] operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.[5]

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

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

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

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

rEntry number (state)

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

zEntry number (state)

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

record number, rVariables (state)

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

record count, rVariables (state)

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

record interval, rVariables (state)

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

dimension indices, rVariables (state)

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

dimension counts, rVariables (state)

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

dimension intervals, rVariables (state)

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

sequential value, rVariable (state)

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

record number, zVariable (state)

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

record count, zVariable (state)

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

record interval, zVariable (state)

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

dimension indices, zVariable (state)

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

dimension counts, zVariable (state)

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

dimension intervals, zVariable (state)

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

sequential value, zVariable (state)

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

status code (state)

When inquiring the explanation of a CDF status code, the text returned is always for the current status code. One current status code is maintained for the entire CDF library (regardless of the number of open CDFs). The current status code may be selected using the `<SELECT_,CDF_STATUS_>` operation. There is no default current status code. Note that the current status code is NOT the status code from the last operation performed.[6]

6.3 Returned Status

`CDF_lib` stores a status code of type `INTEGER*4` in the last argument given.[7] Since more than one operation may be performed with a single call to `CDF_lib`, the following rules apply.

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

[7] `CDF_lib` has been changed from a subroutine to a function and now also returns the status code.

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

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

6.4 Indentation/Style

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

```
status = CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id,
1                 PUT_, CDF_FORMAT_, format,
2                       CDF_MAJORITY_, majority,
3                 CREATE_, ATTR_, attr_name, scope, attr_num,
4                          rVAR_, var_name, data_type, num_elements,
5                                 rec_vary, dim_varys, var_num,
6                 NULL_, status)
```

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 `CDF_lib` without the proper indentation.

```
status = CDF_lib (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id, PUT_,
1                 CDF_FORMAT_, format, CDF_MAJORITY_, majority, CREATE_,
2                 ATTR_, attr_name, scope, attr_num, rVAR_, var_name,
3                 data_type, num_elements, rec_vary, dim_varys, var_num,
4                 NULL_, status)
```

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

6.5 Syntax

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

The syntax for CDF_lib is as follows:

```
 status = CDF_lib (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_, status)
```

where `fnc`x is a function to perform, `item`x is the item on which to perform the function, and `arg`x is a required argument for the operation. The `NULL_` function must be used to end the call to `CDF_lib`. The completion status is stored in `status` as well as being returned.

Previously, `CDF_lib` was a subroutine. It was changed to a function which returns the completion status code (and still stores it in the last argument) to ease the debugging of calls to `CDF_lib`.[8] If in a call to `CDF_lib` an unknown function or item is specified, or if an operation's argument is missing, the `status` argument would never be reached (and `BAD_FNC_OR_ITEM` would not be stored). By returning the completion status code this situation should not occur. Note that the same Fortran variable can be used to receive the status code and as the last argument in the call to `CDF_lib`.

6.5.1 Macintosh, MPW Fortran

The MPW Fortran compiler does not allow variable length argument lists such as those used by `CDF_lib`.[9] For that reason, a number of additional Internal Interface functions are available named `CDF_lib_4`, `CDF_lib_5`, etc. Each of these functions expects the number of arguments indicated by their names. The maximum number of arguments is at least 25 (corresponding to `CDF_lib_25`) but can be increased if necessary by contacting CDFsupport. Using these functions, the second example shown in this section would be as follows:

```
 .
 .
 status = CDF_lib_15 (CREATE_, CDF_, CDF_name, num_dims, dim_sizes, id,
```

[8] Current applications do not have to be changed because the completion status code is still stored in the last argument.

[9] If you know of a way to make MPW Fortran accept variable length argument lists, by all means let us know. We don't like having to do this any more than you do.

```
1                    PUT_, CDF_ENCODING_, encoding,
2                          CDF_MAJORITY_, majority,
3                          CDF_FORMAT_, format,
4                    NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Note that `CDF_lib` may still be used but with the same number of arguments for each occurrence.

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 argumant list that is passsed to 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 in the following sections. 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 60.

`<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 in only applicable to multi-file CDFs.

There are no required arguments.

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

`<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: `INTEGER*4 attr_num`

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: `CHARACTER attr_name*(*)`

The attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters.

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

`<CONFIRM_,CDF_>`

Confirms the current CDF. Required arguments are as follows:

out: `INTEGER*4 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: `INTEGER*4 num_buffers`

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: `INTEGER*4 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: `CHARACTER CDF_name*(CDF_FILENAME_LEN)`

File name of the CDF. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `CDF_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 status`

The status code.

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

<CONFIRM_,CDF_zMODE_>

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

out: `INTEGER*4 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: `INTEGER*4 num_buffers`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 var_num`

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: `INTEGER*4 num_buffers`

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: `CHARACTER var_name*(*)`

The rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

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

<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: `INTEGER*4 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: `INTEGER*4 rec_num`

Record number.

out: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 rec_count`

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: `INTEGER*4 rec_interval`

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: `INTEGER*4 rec_num`

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: `INTEGER*4 num_buffers`

Record number.

The number of cache buffers being used.

`<CONFIRM_,zENTRY_>`

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

out: `INTEGER*4 entry_num`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 var_num`

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: `INTEGER*4 num_buffers`

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `CHARACTER var_name*(*)`

The zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

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

`<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: `INTEGER*4 rec_count`

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: `INTEGER*4 rec_interval`

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: `INTEGER*4 rec_num`

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: `INTEGER*4 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: `INTEGER*4 rec_num`

Record number.

out: `INTEGER*4 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: `CHARACTER attr_name*(*)`

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

UNIX: For the proper operation of `CDF_lib`, `attr_name` MUST be a Fortran `CHARACTER` variable or constant.

in: `INTEGER*4 scope`

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

out: `INTEGER*4 attr_num`

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

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

<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: `CHARACTER CDF_name*(*)`

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

UNIX: File names are case-sensitive.

UNIX: For the proper operation of `CDF_lib`, `CDF_name` MUST be a Fortran `CHARACTER` variable or constant.

in: `INTEGER*4 num_dims`

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

in: `INTEGER*4 dim_sizes(*)`

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

out: `INTEGER*4 id`

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

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

<PUT_,CDF_ENCODING_>, and <PUT_,CDF_MAJORITY_> operations if necessary.

The <CLOSE_,CDF_> operation MUST be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk.

There are no required preselected objects/states.

<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: `CHARACTER var_name*(*)`

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

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

in: `INTEGER*4 data_type`

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

in: `INTEGER*4 num_elements`

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. 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: `INTEGER*4 rec_vary`

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

in: `INTEGER*4 dim_varys(*)`

Dimension variances. Each element of `dim_varys` 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: `INTEGER*4 var_num`

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

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

<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: `CHARACTER var_name*(*)`

Name of the zVariable to be created. This can be at most `CDF_VAR_NAME_LEN` characters. Variable names are case-sensitive.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

in: INTEGER*4 data_type

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

in: INTEGER*4 num_elements

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. 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: INTEGER*4 num_dims

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

in: INTEGER*4 dim_sizes(*)

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

in: INTEGER*4 rec_vary

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

in: INTEGER*4 dim_varys(*)

Dimension variances. Each element of dim_varys 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: INTEGER*4 var_num

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

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

<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: `INTEGER*4 first_record`

The record number of the first record to be deleted.

in: `INTEGER*4 last_record`

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 zEntries which numerically follow the zEntries 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: `INTEGER*4 first_record`

The record number of the first record to be deleted.

in: `INTEGER*4 last_record`

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: `INTEGER*4 max_entry`

The maximum gEntry number for the attribute. If no gEntries exist, then a value of zero (0) 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: `INTEGER*4 max_entry`

The maximum rEntry number for the attribute. If no rEntries exist, then a value of zero (0) 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: `INTEGER*4 max_entry`

The maximum zEntry number for the attribute. If no zEntries exist, then a value of zero (0) 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: `CHARACTER attr_name*(CDF_ATTR_NAME_LEN)`

Attribute name. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `attr_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `CHARACTER attr_name*(*)`

Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `attr_name` MUST be a Fortran `CHARACTER` variable or constant.

out: `INTEGER*4 attr_num`

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: `INTEGER*4 num_entries`

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: `INTEGER*4 num_entries`

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: `INTEGER*4 num_entries`

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: `INTEGER*4 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: `INTEGER*4 c_type`

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

out: `INTEGER*4 c_parms(CDF_MAX_PARMS)`

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

out: `INTEGER*4 c_pct`

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: `CHARACTER copy_right*(CDF_COPYRIGHT_LEN)`

CDF copyright text. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `copy_right` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `CHARACTER CDF_name*(*)`

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

UNIX: File names are case-sensitive.

UNIX: For the proper operation of `CDF_lib`, `CDF_name` MUST be a Fortran `CHARACTER` variable or constant.

out: `INTEGER*4 c_type`

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

out: `INTEGER*4 c_parms(CDF_MAX_PARMS)`

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

out: `INTEGER*4 c_size`

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

out: `INTEGER*4 u_size`

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: `INTEGER*4 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: `INTEGER*4 num_attrs`

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: `INTEGER*4 num_attrs`

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: `INTEGER*4 num_vars`

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: `INTEGER*4 num_attrs`

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: `INTEGER*4 num_vars`

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 data_type`

Data type.

out: `INTEGER*4 num_bytes`

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: `<type> value`

The value. `<type>` depends on the data type of the gEntry. The value is read from the CDF and placed into `value`. This buffer must be large enough to hold the value.

WARNING: If the gEntry has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the gEntry does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 data_type`

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: `INTEGER*4 num_elements`

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

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 for the CDF library being used. Required arguments are as follows:

out: `CHARACTER copy_right*(CDF_COPYRIGHT_LEN)`

CDF library copyright text. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `copy_right` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 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: INTEGER*4 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: CHARACTER*1 subincrement

Subincremental character.

UNIX: For the proper operation of CDF_lib, subincrement MUST be a Fortran CHARACTER variable or constant.

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: INTEGER*4 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: <type> value

The value. <type> depends on the data type of the rEntry. The value is read from the CDF and placed into value. This buffer must be large enough to hold the value.

WARNING: If the rEntry has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the rEntry does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.

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: `INTEGER*4 data_type`

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: `INTEGER*4 num_elements`

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

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: `INTEGER*4 start_record`

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: `INTEGER*4 next_record`

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: `INTEGER*4 start_record`

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

out: `INTEGER*4 next_record`

The number of the last allocated record.

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

`<GET_,rVAR_BLOCKINGFACTOR_>`[10]

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: INTEGER*4 blocking_factor

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: INTEGER*4 c_type

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

out: INTEGER*4 c_parms(CDF_MAX_PARMS)

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

out: INTEGER*4 c_pct

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

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

<GET_,rVAR_DATA_>

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

out: <type> value

The value. <type> depends on the data type of the rVariable. This buffer must be large enough to hold the value. The value is read from the CDF and placed into value.

WARNING: If the rVariable has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the rVariable does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.

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: INTEGER*4 data_type

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

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

[10]The item rVAR_BLOCKINGFACTOR_ was previously named rVAR_EXTENDRECS_.

<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: `INTEGER*4 dim_varys(CDF_MAX_DIMS)`

Dimension variances. Each element of `dim_varys` 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: `<type> buffer`

Value(s). `<type>` depends on the data type of the rVariable. The values are read from the CDF and placed into `buffer`. This buffer must be large enough to hold the values.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `buffer` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `buffer` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 max_rec`

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 one (1). A value of zero (0) indicates that no records have been written. Required arguments are as follows:

out: `INTEGER*4 max_rec`

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: `CHARACTER var_name*(CDF_VAR_NAME_LEN)`

Name of the rVariable. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 num_entries`

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: `INTEGER*4 num_levels`

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: `INTEGER*4 num_records`

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: `INTEGER*4 num_records`

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: `CHARACTER var_name*(*)`

The rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

out: `INTEGER*4 var_num`

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: `INTEGER*4 num_elements`

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: `INTEGER*4 num_records`

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: `<type> value`

Pad value. `<type>` depends on the data type of the rVariable. The pad value is read from the CDF and placed into `value`. This buffer must be large enough to hold the pad value.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 rec_vary`

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: `<type> value`

Value. `<type>` depends on the data type of the rVariable. The value is read from the CDF and placed into `value`. This buffer must be large enough to hold the value.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 s_arrays_type`

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

out: `INTEGER*4 s_arrays_parms(CDF_MAX_PARMS)`

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

out: `INTEGER*4 s_arrays_pct`

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: `INTEGER*4 s_records_type`

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: `INTEGER*4 dim_sizes(*)`

Dimension sizes. Each element of `dim_sizes` 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. A value of zero (0) indicates that no records have been written to the rVariables in the CDF. The maximum record number for an individual rVariable may be inquired using the `<GET_,rVAR_MAXREC_>` operation. Required arguments are as follows:

out: `INTEGER*4 max_rec`

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: `INTEGER*4 num_dims`

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: `INTEGER*4 num_vars`

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

in: `INTEGER*4 var_nums(*)`

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

in: `<type> buffer`

The buffer into which the full-physical rVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to place the full-physical records being read.) The order of the full-physical rVariable records in this buffer will correspond to the rVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical rVariable records. Be careful if using Fortran STRUCTUREs to receive multiple full-physical rVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See 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.

`<GET_,STATUS_TEXT_>`

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

out: `CHARACTER text*(CDF_STATUSTEXT_LEN)`

Text explaining the status code. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `text` MUST be a Fortran `CHARACTER` variable or constant.

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: `<type> value`

The value. `<type>` depends on the data type of the zEntry. The value is read from the CDF and placed into `value`. This buffer must be large enough to hold the value.

WARNING: If the zEntry has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zEntry does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 data_type`

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: `INTEGER*4 num_elements`

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

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: `INTEGER*4 start_record`

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: `INTEGER*4 next_record`

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: `INTEGER*4 start_record`

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

out: `INTEGER*4 next_record`

The number of the last allocated record.

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

<GET_,zVAR_BLOCKINGFACTOR_>[11]

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: `INTEGER*4 blocking_factor`

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.

[11] The item `zVAR_BLOCKINGFACTOR_` was previously named `zVAR_EXTENDRECS_`.

<GET_,zVAR_COMPRESSION_>

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

out: INTEGER*4 c_type

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

out: INTEGER*4 c_parms(CDF_MAX_PARMS)

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

out: INTEGER*4 c_pct

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: <type> value

The value. <type> depends on the data type of the zVariable. This buffer must be large enough to hold the value. The value is read from the CDF and placed into value.

WARNING: If the zVariable has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the zVariable does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.

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: INTEGER*4 data_type

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: INTEGER*4 dim_sizes(CDF_MAX_DIMS)

Dimension sizes. Each element of dim_sizes 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: `INTEGER*4 dim_varys(CDF_MAX_DIMS)`

Dimension variances. Each element of `dim_varys` 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: `<type> buffer`

The value(s). `<type>` depends on the data type of the zVariable. The values are read from the CDF and placed into `buffer`. This buffer must be large enough to hold the values.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 max_rec`

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 one (1). A value of zero (0) indicates that no records have been written. Required arguments are as follows:

out: `INTEGER*4 max_rec`

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: `CHARACTER var_name*(CDF_VAR_NAME_LEN)`

Name of the zVariable. This character string will be blank padded if necessary.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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 has significance only 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: `INTEGER*4 num_entries`

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: `INTEGER*4 num_levels`

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 has significance only 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: `INTEGER*4 num_records`

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: `INTEGER*4 num_records`

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: `CHARACTER var_name*(*)`

The zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

out: `INTEGER*4 var_num`

The rVariable 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: `INTEGER*4 num_dims`

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: `INTEGER*4 num_elements`

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: `INTEGER*4 num_records`

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: `<type> value`

The pad value. `<type>` depends on the data type of the zVariable. The pad value is read from the CDF and placed into `value`. This buffer must be large enough to hold the pad value.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 rec_vary`

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: `<type> value`

The value. `<type>` depends on the data type of the zVariable. The value is read from the CDF and placed into `value`. This buffer must be large enough to hold the value.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 s_arrays_type`

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

out: `INTEGER*4 s_arrays_parms(CDF_MAX_PARMS)`

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

out: `INTEGER*4 s_arrays_pct`

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: `INTEGER*4 s_records_type`

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. A value of zero (0) indicates that no records have been written to the zVariables in the CDF. The maximum record number for an individual zVariable may be inquired using the <GET_,zVAR_MAXREC_> operation. Required arguments are as follows:

out: `INTEGER*4 max_rec`

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: `INTEGER*4 num_vars`

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

in: `INTEGER*4 var_nums(*)`

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

in: `<type> buffer`

The buffer into which the full-physical zVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to place the full-physical records being read.) The order of the full-physical zVariable records in this buffer will correspond to the zVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical zVariable records. Be careful if using Fortran STRUCTUREs to receive multiple full-physical zVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See 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 the `<SELECT_,zVAR_RECNUMBER_>` operation).

`<NULL_>`

Ends the argument list that is passed to an internal interface call. No more other arguments are allowed to follow except the returned status code as the very last argument.

`<OPEN_,CDF_>`

Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: `CHARACTER CDF_name*(*)`

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

UNIX: File names are case-sensitive.

UNIX: For the proper operation of `CDF_lib`, `CDF_name` MUST be a Fortran `CHARACTER` variable or constant.

out: `INTEGER*4 id`

CDF identifier to be used in subsequent operations of 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: `CHARACTER attr_name*(*)`

New attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `attr_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 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: `INTEGER*4 c_type`

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

in: `INTEGER*4 c_parms(*)`

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 data_type`

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

in: `INTEGER*4 num_elements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (CDF_CHAR and CDF_UCHAR) this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

in: `<type> value`

The value(s). `<type>` depends on the data type of the gEntry. The value is written to the CDF from `value`.

WARNING: If the gEntry has one of the character data types (CDF_CHAR or CDF_UCHAR), then `value` must be a CHARACTER Fortran variable. If the gEntry does not have one of the character data types, then `value` must NOT be a CHARACTER Fortran variable.

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: `INTEGER*4 data_type`

New data type of the gEntry. Specify one of the data types described in Section 4.5.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

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

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: `INTEGER*4 data_type`

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

in: `INTEGER*4 num_elements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (CDF_CHAR and CDF_UCHAR) this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

in: `<type> value`

The value(s). `<type>` depends on the data type of the rEntry. The value is written to the CDF from `value`.

WARNING: If the rEntry has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the rEntry does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 data_type`

New data type of the rEntry. Specify one of the data types described in Section 4.5.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

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

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: `INTEGER*4 first_record`

The first record number to allocate.

in: `INTEGER*4 last_record`

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 `1` (one). 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: `INTEGER*4 num_records`

Number of records to allocate.

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

`<PUT_,rVAR_BLOCKINGFACTOR_>`[12]

Specifies the blocking factor for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes blocking factors. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs.

Required arguments are as follows:

in: `INTEGER*4 blocking_factor`

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: `INTEGER*4 c_type`

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

in: `INTEGER*4 c_parms(*)`

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: `<type> value`

Value. `<type>` depends on the data type of the rVariable. The value is written to the CDF from `value`.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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

`<PUT_,rVAR_DATASPEC_>`

Respecifies the data specification (data type and number of elements) of the current rVariable (in the current CDF). An rVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `INTEGER*4 data_type`

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

New data type. Specify one of the data types described in Section 4.5.

in: `INTEGER*4 num_elements`

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. (A string exists at each value.) For non-character data types this must be one (1) — multiple elements 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: `INTEGER*4 dim_varys(*)`

New dimension variances. Each element of `dim_varys` 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: `<type> buffer`

The values. `<type>` depends on the data type of the rVariable. The values in `buffer` are written to the CDF.

WARNING: If the rVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `buffer` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `buffer` must NOT be a `CHARACTER` Fortran variable.

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 `1` (one). 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: `INTEGER*4 num_records`

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: CHARACTER var_name*(*)

New name of the rVariable. This may consist of at most CDF_VAR_NAME_LEN characters.

UNIX: For the proper operation of CDF_lib, var_name MUST be a Fortran CHARACTER variable or constant.

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: <type> value

The pad value. <type> depends on the data type of the rVariable. The pad value is written to the CDF from value.

WARNING: If the rVariable has one of the character data types (CDF_CHAR or CDF_UCHAR), then value must be a CHARACTER Fortran variable. If the rVariable does not have one of the character data types, then value must NOT be a CHARACTER Fortran variable.

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: INTEGER*4 rec_vary

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: <type> value

Value. <type> depends on the data type of the rVariable. The value is written to the CDF from value.

WARNING: If the rVariable has one of the character data types (CDF_CHAR or CDF_UCHAR), then `value` must be a `CHARACTER` Fortran variable. If the rVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 s_arrays_type`

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

in: `INTEGER*4 s_arrays_parms(*)`

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: `INTEGER*4 s_records_type`

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: `INTEGER*4 num_vars`

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

in: `INTEGER*4 var_nums(*)`

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

in: `<type> buffer`

The buffer of full-physical rVariable records to be written. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to get the full-physical records being written.) The order of the full-physical rVariable records in this buffer must agree with the rVariable numbers listed in `varNums` and this buffer must be contiguous — there can be no spacing between full-physical rVariable records. Be careful if using Fortran STRUCTUREs to store multiple full-physical rVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See 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: `INTEGER*4 data_type`

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

in: `INTEGER*4 num_elements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string. For all other data types this is the number of elements in an array of that data type.

in: `<type> value`

The value(s). `<type>` depends on the data type of the zEntry. The value is written to the CDF from `value`.

WARNING: If the zEntry has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zEntry does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 data_type`

New data type of the zEntry. Specify one of the data types described in Section 4.5.

in: `INTEGER*4 num_elements`

Number of elements of the data type.

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

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: `INTEGER*4 first_record`

The first record number to allocate.

in: `INTEGER*4 last_record`

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 `1` (one). 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: `INTEGER*4 num_records`

Number of records to allocate.

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

`<PUT_,zVAR_BLOCKINGFACTOR_>`[13]

Specifies the blocking factor for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes blocking factors. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs.

Required arguments are as follows:

in: `INTEGER*4 blocking_factor`

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: `INTEGER*4 c_type`

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

in: `INTEGER*4 c_parms(*)`

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.

[13] The item `zVAR_BLOCKINGFACTOR_` was previously named `zVAR_EXTENDRECS_`.

<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: `<type> value`

The value. `<type>` depends on the data type of the zVariable. The value is written to the CDF from `value`.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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

<PUT_,zVAR_DATASPEC_>

Respecifies the data specification (data type and number of elements) of the current zVariable (in the current CDF). A zVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `INTEGER*4 data_type`

New data type. Specify one of the data types described in Section 4.5.

in: `INTEGER*4 num_elements`

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. (A string exists at each value.) For non-character data types this must be one (1) — multiple elements 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: `INTEGER*4 dim_varys(*)`

New dimension variances. Each element of `dim_varys` 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 `1` (one). 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: `INTEGER*4 num_records`

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: `<type> buffer`

The values. `<type>` depends on the data type of the zVariable. The values in `buffer` are written to the CDF.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `buffer` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `buffer` must NOT be a `CHARACTER` Fortran variable.

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: `CHARACTER var_name*(*)`

New name of the zVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `<type> value`

The pad value. `<type>` depends on the data type of the zVariable. The pad value is written to the CDF from `value`.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 rec_vary`

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: `<type> value`

The value. `<type>` depends on the data type of the zVariable. The value is written to the CDF from `value`.

WARNING: If the zVariable has one of the character data types (`CDF_CHAR` or `CDF_UCHAR`), then `value` must be a `CHARACTER` Fortran variable. If the zVariable does not have one of the character data types, then `value` must NOT be a `CHARACTER` Fortran variable.

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: `INTEGER*4 s_arrays_type`

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

in: `INTEGER*4 s_arrays_parms(*)`

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: `INTEGER*4 s_records_type`

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: `INTEGER*4 num_vars`

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

in: `INTEGER*4 var_nums(*)`

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

in: `<type> buffer`

The buffer of full-physical zVariable records to be written. `<type>` must be a Fortran variable that will be passed by reference and cannot be of type `CHARACTER`. (The CDF library is expecting an address at which to get the full-physical records being written.) The order of the full-physical zVariable records in this buffer must agree with the zVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical zVariable records. Be careful if using Fortran STRUCTUREs to store multiple full-physical zVariable records. Fortran compilers on some operating systems will pad between the elements of a STRUCTURE in order to prevent memory alignment errors (i.e., the elements of a STRUCTURE may not be contiguous). See 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 the `<SELECT_,zVAR_RECNUMBER_>` operation).

`<SELECT_,ATTR_>`

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

in: `INTEGER*4 attr_num`

Attribute number.

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

`<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: CHARACTER attr_name*(*)

Attribute name. This may be at most CDF_ATTR_NAME_LEN characters.

UNIX: For the proper operation of CDF_lib, attr_name MUST be a Fortran CHARACTER variable or constant.

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

<SELECT_,CDF_>

Explicitly selects the current CDF. Required arguments are as follows:

in: INTEGER*4 id

Identifier of the CDF. This identifier must have been initialized by a successful <CREATE_,CDF_> or <OPEN_,CDF_> operation.

There are no required preselected objects/states.

<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: INTEGER*4 num_buffers

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: INTEGER*4 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: INTEGER*4 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: INTEGER*4 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: `CHARACTER scratch_dir*(*)`

The directory to be used for scratch files. The length of this directory specification is limited only by the operating system being used.

UNIX: For the proper operation of `CDF_lib`, `scratch_dir` MUST be a Fortran `CHARACTER` variable or constant.

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

<SELECT_,CDF_STATUS_>

Selects the current status code. Required arguments are as follows:

in: `INTEGER*4 status`

CDF status code.

There are no required preselected objects/states.

<SELECT_,CDF_zMODE_>

Selects a zMode (for the current CDF). Required arguments are as follows:

in: `INTEGER*4 mode`

The zMode. Specify one of the zModes described in Section 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: `INTEGER*4 num_buffers`

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: `INTEGER*4 entry_num`

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: `INTEGER*4 entry_num`

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: `CHARACTER var_name*(*)`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 var_num`

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: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

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

`<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: `CHARACTER var_name*(*)`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 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 the current sequential value is maintained for each rVariable individually. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

in: `INTEGER*4 indices(*)`

Dimension indices. 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: `INTEGER*4 num_buffers`

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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 rec_count`

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: `INTEGER*4 rec_interval`

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: `INTEGER*4 rec_num`

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: `INTEGER*4 num_buffers`

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: `INTEGER*4 entry_num`

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: `CHARACTER var_name*(*)`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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: `INTEGER*4 var_num`

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: `INTEGER*4 num_buffers`

The number of cache buffers to be used.

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

<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: `CHARACTER var_name*(*)`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters.

UNIX: For the proper operation of `CDF_lib`, `var_name` MUST be a Fortran `CHARACTER` variable or constant.

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

<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: `INTEGER*4 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: `INTEGER*4 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: `INTEGER*4 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_RECCOUNT_>

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

in: `INTEGER*4 rec_count`

Record count.

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

<SELECT_,zVAR_RECINTERVAL_>

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

in: `INTEGER*4 rec_interval`

Record interval.

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

<SELECT_,zVAR_RECNUMBER_>

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

in: `INTEGER*4 rec_num`

Record number.

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

<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: `INTEGER*4 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 the current sequential value is maintained for each zVariable individually. Required arguments are as follows:

in: `INTEGER*4 rec_num`

Record number.

in: `INTEGER*4 indices(*)`

Dimension indices. 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: `INTEGER*4 num_buffers`

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: `INTEGER*4 rec_num`

Record number.

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

6.7 More Examples

Several more examples of the use of `CDF_lib` follow. In each example it is assumed that the current CDF has already been selected (either implicitly by creating/opening the CDF or explicitly with the `<SELECT_,CDF_>` operation).

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 '<path>cdf.h'
.
.
INTEGER*4 status                     ! Status returned from CDF library.
INTEGER*4 dim_varys(2)               ! Dimension variances.
INTEGER*4 var_num                    ! rVariable number.
REAL*4    pad_value                  ! Pad value.

DATA pad_value/-999.9/
.
.
dim_varys(1) = VARY
dim_varys(2) = VARY
status = CDF_lib (CREATE_, rVAR_, 'HUMIDITY', CDF_REAL4, 1, VARY,
1                                 dim_varys, var_num,
```

```
2                 PUT_, rVAR_PADVALUE_, pad_value,
3                       rVAR_INITIALRECS_, 500,
4                       rVAR_BLOCKINGFACTOR_, 50,
5                 NULL_, status)
IF (status .NE. CDF_OK) CALL 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 '<path>CDF.INC'
.
.
INTEGER*4 status                    ! Status returned from CDF library.
INTEGER*4 dim_varys(1)              ! Dimension variances.
INTEGER*4 var_num                   ! zVariable number.
INTEGER*4 num_dims                  ! Number of dimension.
INTEGER*4 dim_sizes(1)              ! Dimension sizes.
INTEGER*4 num_elems                 ! Number of elements (of the data
                                    ! type).
CHARACTER*10 pad_value              ! Pad value.

DATA pad_value/'**********'/,
0    num_dims/1/,
1    dim_sizes/20/,
2    num_elems/10/
.
.
dim_varys(1) = VARY
status = CDF_lib (CREATE_, zVAR_, 'Station', CDF_CHAR, num_elems, num_dims,
1                          dim_sizes, NOVARY, dim_varys, var_num,
2                 PUT_, zVAR_PADVALUE_, pad_value,
3                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

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 column major, and the data type of the rVariable is CDF_INT2. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<path>CDF.INC'
.
.
INTEGER*4 status                ! Status returned from CDF library.
INTEGER*2 values(50,100)        ! Buffer to receive values.
INTEGER*4 rec_count             ! Record count, one record per hyper get.
INTEGER*4 rec_interval          ! Record interval, set to one to indicate
                                ! contiguous records (really meaningless
                                ! since record count is one).
INTEGER*4 indices(2)            ! Dimension indices, start each read
                                ! at 1,1 of the array.
INTEGER*4 counts(2)             ! Dimension counts, half of the values along
                                ! each dimension will be read.
INTEGER*4 intervals(2)          ! Dimension intervals, every other value
                                ! along each dimension will be read.
INTEGER*4 rec_num               ! Record number.
INTEGER*4 max_rec               ! Maximum rVariable record in the
                                ! CDF - this was determined with a call
                                ! to CDF_inquire.

DATA rec_count/1/, rec_interval/1/, indices/1,1/, counts/50,100/,
1    intervals/2,2/
.
.
status = CDF_lib (SELECT_, rVAR_NAME_, 'BRIGHTNESS',
1                          rVARs_RECCOUNT_, rec_count,
2                          rVARs_RECINTERVAL_, rec_interval,
3                          rVARs_DIMINDICES_, indices,
4                          rVARs_DIMCOUNTS_, counts,
5                          rVARs_DIMINTERVALS_, intervals,
6                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

DO rec_num = 1, max_rec
  status = CDF_lib (SELECT_, rVARs_RECNUMBER_, rec_num,
  1                 GET_, rVAR_HYPERDATA_, values,
  2                 NULL_, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
  .
  .
  ! process values
  .
  .
END DO
.
.
```

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 '<path>CDF.INC'
.
.
INTEGER*4 status               ! Status returned from CDF library.
.
.
status = CDF_lib (SELECT_, ATTR_NAME_, 'Tmp',
1                 PUT_, ATTR_NAME, 'TMP',
2                 NULL_, status)
IF (status .NE. CDF_OK) CALL 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 '<path>CDF.INC'
.
.
INTEGER*4 status               ! Status returned from CDF library.
INTEGER*4 var_num              ! zVariable number.
INTEGER*4 rec_num              ! Record number, start at first record.
INTEGER*4 indices(2)           ! Dimension indices.
REAL*4    value                ! Value read.
REAL*8    sum                  ! Sum of all values.
INTEGER*4 count                ! Number of values.
REAL*4    ave                  ! Average value.

DATA indices/1,1/, sum/0.0/, count/0/, rec_num/1/
.
.
status = CDF_lib (GET_, zVAR_NUMBER_, 'FLUX', var_num,
1                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
```

```
status = CDF_lib (SELECT_, zVAR_, var_num,
1                          zVAR_SEQPOS_, rec_num, indices,
2                 GET_, zVAR_SEQDATA_, value,
3                 NULL_, status)

DO WHILE (status .GE. CDF_OK)
  sum = sum + value
  count = count + 1
  status = CDF_lib (GET_, zVAR_SEQDATA_, value,
  1                 NULL_, status)
END DO

IF (status .NE. END_OF_VAR) CALL 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 '<path>CDF.INC'
.
.
INTEGER*4 status                  ! Status returned from CDF library.
REAL*4    scale(2)                ! Scale, minimum/maximum.

DATA scale/-90.0,90.0/
.
.
status = CDF_lib (SELECT_, rENTRY_NAME_, 'LATITUDE',
1                          ATTR_NAME_, 'FIELDNAM',
2                 PUT_, rENTRY_DATA_, CDF_CHAR, 20, 'Latitude            ',
3                 SELECT_, ATTR_NAME_, 'SCALE',
4                 PUT_, rENTRY_DATA_, CDF_REAL4, 2, scale,
5                 SELECT_, ATTR_NAME_, 'UNITS',
6                 PUT_, rENTRY_DATA_, CDF_CHAR, 20, 'Degrees north       ',
7                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

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 '<path>CDF.INC'
.
.
INTEGER*4 status                  ! Status returned from CDF library.
INTEGER*2 time                    ! 'Time' value.
BYTE      vector_a(3)             ! 'vectorA' values.
REAL*8    vector_b(5)             ! 'vectorB' values.
INTEGER*4 rec_number              ! Record number.
BYTE      buffer(45)              ! Buffer of full-physical records.
INTEGER*4 var_numbers(3)          ! Variable numbers.

EQUIVALENCE (vector_b, buffer(1))
EQUIVALENCE (time, buffer(41))
EQUIVALENCE (vector_a, buffer(43))
.
.
status = CDF_lib (GET_, zVAR_NUMBER_, 'vectorB', var_numbers(1),
1                       zVAR_NUMBER_, 'time', var_numbers(2),
2                       zVAR_NUMBER_, 'vectorA', var_numbers(3),
3                 NULL_, status);
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
DO rec_number = 1, 100
  .
  /* read values from input file */
  .
  status = CDF_lib (SELECT_, zVARs_RECNUMBER_, rec_number,
  1                 PUT_, zVARs_RECDATA_, 3L, var_numbers, buffer,
  2                 NULL_, status);
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END DO
.
.
```

Chapter 7

Interpreting CDF Status Codes

Most CDF functions return a status code of type `INTEGER*4`. The symbolic names for these codes are defined in `cdf.inc` and should be used in your applications rather than using the true numeric values. Appendix 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.

```
INTEGER*4 status
.
.
CALL CDF_function (..., status)   ! any CDF function returning status
IF (status .NE. CDF_OK) THEN
  CALL UserStatusHandler (status, ...)
  .
  .
END IF
```

In your own status handler you can take whatever action is appropriate to the application. An example status handler follows. Note that no action is taken in the status handler if the status is `CDF_OK`.

```
INCLUDE '<path>cdf.inc'
```

```
   SUBROUTINE UserStatusHandler (status)
   INTEGER*4 status

   CHARACTER message*(CDF_STATUSTEXT_LEN)

   IF (status .LT. CDF_WARN) THEN
     WRITE (6,10)
10   FORMAT (' ','An error has occurred, halting...')
     CALL CDF_error (status, message)
     WRITE (6,11) message
11   FORMAT (' ',A)
     STOP
   ELSE
     IF (status .LT. CDF_OK) THEN
       WRITE (6,12)
12     FORMAT (' ','Warning, function may not have completed as expected...')
       CALL CDF_error (status, message)
       WRITE (6,13) message
13     FORMAT (' ',A)
     ELSE
       IF (status .GT. CDF_OK) THEN
         WRITE (6,14)
14       FORMAT (' ','Function completed successfully, but be advised that...')
         CALL CDF_error (status, message)
         WRITE (6,15) message
15       FORMAT (' ',A)
       END IF
     END IF
   END IF

   RETURN
   END
```

Explanations for all CDF status codes are available to your applications through the function `CDF_error`. `CDF_error` writes to a text string an explanation of a given status code.

Chapter 8

EPOCH Utility Routines

Several subroutines exist that compute, decompose, parse, and encode CDF_EPOCH values. These subroutines may be called by applications using the CDF_EPOCH data type and are included in the CDF library. The Concepts chapter in the CDF User's Guide describes EPOCH values.

8.1 compute_EPOCH

compute_EPOCH calculates a CDF_EPOCH value given the individual components. If an illegal component is detected, the value returned will be -1.0.

```
 SUBROUTINE compute_EPOCH (year, month, day, hour, minute, second, msec,
.                          epoch)
 INTEGER*4 year              ! In  -- Year (AD, e.g., 1994).
 INTEGER*4 month             ! In  -- Month (1-12).
 INTEGER*4 day               ! In  -- Day (1-31).
 INTEGER*4 hour              ! In  -- Hour (0-23).
 INTEGER*4 minute            ! In  -- Minute (0-59).
 INTEGER*4 second            ! In  -- Second (0-59).
 INTEGER*4 msec              ! In  -- Millisecond (0-999).
 REAL*8    epoch             ! Out -- CDF_EPOCH value.
```

NOTE: There are two variations on how compute_EPOCH 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 EPOCH_breakdown

EPOCH_breakdown decomposes a CDF_EPOCH value into the individual components.

```
SUBROUTINE EPOCH_breakdown (epoch, year, month, day, hour, minute, second,
.                           msec)
REAL*8    epoch           ! In  -- The CDF_EPOCH value.
INTEGER*4 year            ! Out -- Year (AD, e.g., 1994).
INTEGER*4 month           ! Out -- Month (1-12).
INTEGER*4 day             ! Out -- Day (1-31).
INTEGER*4 hour            ! Out -- Hour (0-23).
INTEGER*4 minute          ! Out -- Minute (0-59).
INTEGER*4 second          ! Out -- Second (0-59).
INTEGER*4 msec            ! Out -- Millisecond (0-999).
```

8.3 encode_EPOCH

encode_EPOCH 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).

```
SUBROUTINE encode_EPOCH (epoch, ep_string)
REAL*8    epoch                         ! In  -- The CDF_EPOCH value.
CHARACTER ep_string*(EPOCH_STRING_LEN)  ! Out -- The standard date/time
                                        !        character string.
```

EPOCH_STRING_LEN is defined in cdf.inc.

8.4 encode_EPOCH1

encode_EPOCH1 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmdd.ttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and ttttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
SUBROUTINE encode_EPOCH1 (epoch, ep_string)
REAL*8    epoch                          ! In  -- The CDF_EPOCH value.
CHARACTER ep_string*(EPOCH1_STRING_LEN)  ! Out -- The alternate date/time
                                         !        character string.
```

EPOCH1_STRING_LEN is defined in cdf.inc.

8.5 encode_EPOCH2

encode_EPOCH2 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).

```
SUBROUTINE encode_EPOCH2 (epoch, ep_string)
REAL*8    epoch                           ! In  -- The CDF_EPOCH value.
CHARACTER ep_string*(EPOCH2_STRING_LEN)   ! Out -- The alternate date/time
                                          !        character string.
```

EPOCH2_STRING_LEN is defined in cdf.inc.

8.6 encode_EPOCH3

encode_EPOCH3 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).

```
SUBROUTINE encode_EPOCH3 (epoch, ep_string)
REAL*8    epoch                           ! In  -- The CDF_EPOCH value.
CHARACTER ep_string*(EPOCH3_STRING_LEN)   ! Out -- The alternate date/time
                                          !        character string.
```

EPOCH3_STRING_LEN is defined in cdf.inc.

8.7 encode_EPOCHx

encode_EPOCHx encodes a CDF_EPOCH value into a custom date/time character string. The format of the encoded string is specified by a format string.

```
SUBROUTINE encode_EPOCHx (epoch, format, encoded)
REAL*8    epoch                           ! In  -- The CDF_EPOCH value.
CHARACTER format*(EPOCHx_FORMAT_MAX)      ! In  -- The format string.
CHARACTER 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.inc.

8.8 parse_EPOCH

parse_EPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is that produces by the encode_EPOCH function described in Section 8.3. If an illegal field is detected in the string, the value returned will be -1.0.

```
SUBROUTINE parse_EPOCH (ep_string, epoch)
CHARACTER ep_string*(EPOCH_STRING_LEN)    ! In  -- The standard date/time
                                          !        character string.
REAL*8    epoch                           ! Out -- CDF_EPOCH value.
```

EPOCH_STRING_LEN is defined in cdf.inc.

8.9 parse_EPOCH1

parse_EPOCH1 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produces by the encode_EPOCH1 function described in Section 8.4. If an illegal field is detected in the string, the value returned will be -1.0.

```
SUBROUTINE parse_EPOCH1 (ep_string, epoch)
CHARACTER ep_string*(EPOCH1_STRING_LEN)   ! In  -- The alternate date/time
                                          !        character string.
REAL*8    epoch                           ! Out -- CDF_EPOCH value.
```

EPOCH1_STRING_LEN is defined in cdf.inc.

8.10 parse_EPOCH2

parse_EPOCH2 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produces by the encode_EPOCH2 function described in Section 8.5. If an illegal field is detected in the string, the value returned will be -1.0.

```
SUBROUTINE parse_EPOCH2 (ep_string, epoch)
CHARACTER ep_string*(EPOCH2_STRING_LEN)   ! In  -- The alternate date/time
                                          !        character string.
REAL*8    epoch                           ! Out -- CDF_EPOCH value.
```

EPOCH2_STRING_LEN is defined in cdf.inc.

8.11 parse_EPOCH3

parse_EPOCH3 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produces by the encode_EPOCH3 function described in Section 8.6. If an illegal field is detected in the string, the value returned will be -1.0.

```
SUBROUTINE parse_EPOCH3 (ep_string, epoch)
CHARACTER ep_string*(EPOCH3_STRING_LEN)   ! In  -- The alternate date/time
                                          !        character string.
REAL*8    epoch                           ! Out -- CDF_EPOCH value.
```

EPOCH3_STRING_LEN is defined in cdf.inc.

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]
IBM_PC_OVERFLOW	An operation involving a buffer greater than 64k bytes in size has been specified for PCs running 16-bit DOS/Windows 3.*. [Error]
ILLEGAL_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

Fortran Programming Summary

B.1 Standard Interface

```
SUBROUTINE CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority,
1                       id, status)
CHARACTER CDF_name                                                  ! in
INTEGER*4 num_dims                                                  ! in
INTEGER*4 dim_sizes(*)                                              ! in
INTEGER*4 encoding                                                  ! in
INTEGER*4 majority                                                  ! in
INTEGER*4 id                                                        ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_open (CDF_name, id, status)
CHARACTER CDF_name*(*)                                              ! in
INTEGER*4 id                                                        ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_doc (id, version, release, text, status)
INTEGER*4 id                                                        ! in
INTEGER*4 version                                                   ! out
INTEGER*4 release                                                   ! out
CHARACTER text*(CDF_DOCUMENT_LEN)                                   ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_inquire (id, num_dims, dim_sizes, encoding, majority,
1                        max_rec, num_vars, num_attrs, status)
INTEGER*4 id                                                        ! in
INTEGER*4 num_dims                                                  ! out
INTEGER*4 dim_sizes(CDF_MAX_DIMS)                                   ! out
INTEGER*4 encoding                                                  ! out
INTEGER*4 majority                                                  ! out
INTEGER*4 max_rec                                                   ! out
```

```
INTEGER*4 num_vars                                                     ! out
INTEGER*4 num_attrs                                                    ! out
INTEGER*4 status                                                       ! out

SUBROUTINE CDF_close (id, status)
INTEGER*4 id                                                           ! in
INTEGER*4 status                                                       ! out

SUBROUTINE CDF_delete (id, status)
INTEGER*4 id                                                           ! in
INTEGER*4 status                                                       ! out

SUBROUTINE CDF_error (status, text)
INTEGER*4 status                                                       ! in
CHARACTER text*(CDF_STATUSTEXT_LEN)                                    ! out

SUBROUTINE CDF_attr_create (id, attr_name, attr_scope, attr_num, status)
INTEGER*4 id                                                           ! in
CHARACTER attr_name*(*)                                                ! in
INTEGER*4 attr_scope                                                   ! in
INTEGER*4 attr_num                                                     ! out
INTEGER*4 status                                                       ! out

INTEGER*4 CDF_attr_num (id, attr_name)
INTEGER*4 id                                                           ! in
CHARACTER attr_name*(*)                                                ! in

SUBROUTINE CDF_attr_rename (id, attr_num, attr_name, status)
INTEGER*4 id                                                           ! in
INTEGER*4 attr_num                                                     ! in
CHARACTER attr_name*(*)                                                ! in
INTEGER*4 status                                                       ! out

SUBROUTINE CDF_attr_inquire (id, attr_num, attr_name, attr_scope, max_entry,
1                            status)
INTEGER*4 id                                                           ! in
INTEGER*4 attr_num                                                     ! in
CHARACTER attr_name*(CDF_ATTR_NAME_LEN)                                ! out
INTEGER*4 attr_scope                                                   ! out
INTEGER*4 max_entry                                                    ! out
INTEGER*4 status                                                       ! out

SUBROUTINE CDF_attr_entry_inquire (id, attr_num, entry_num, data_type,
1                                  num_elements, status)
INTEGER*4 id                                                           ! in
INTEGER*4 attr_num                                                     ! in
INTEGER*4 entry_num                                                    ! in
INTEGER*4 data_type                                                    ! out
INTEGER*4 num_elements                                                 ! out
INTEGER*4 status                                                       ! out
```

```
SUBROUTINE CDF_attr_put (id, attr_num, entry_num, data_type, num_elements,
1                        value, status)
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 entry_num                                                 ! in
INTEGER*4 data_type                                                 ! in
INTEGER*4 num_elements                                              ! in
<type>    value                                                     ! in
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_attr_get (id, attr_num, entry_num, value, status)
INTEGER*4 id                                                        ! in
INTEGER*4 attr_num                                                  ! in
INTEGER*4 entry_num                                                 ! in
<type>    value                                                     ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_var_create (id, var_name, data_type, num_elements,
1                          rec_variance, dim_variances, var_num, status)
INTEGER*4 id                                                        ! in
CHARACTER var_name                                                  ! in
INTEGER*4 data_type                                                 ! in
INTEGER*4 num_elements                                              ! in
INTEGER*4 rec_variance                                              ! in
INTEGER*4 dim_variances(*)                                          ! in
INTEGER*4 var_num                                                   ! out
INTEGER*4 status                                                    ! out

INTEGER*4 FUNCTION CDF_var_num (id, var_name)
INTEGER*4 id                                                        ! in
CHARACTER var_name*(*)                                              ! in

SUBROUTINE CDF_var_rename (id, var_num, var_name, status)
INTEGER*4 id                                                        ! in
INTEGER*4 var_num                                                   ! in
CHARACTER var_name*(*)                                              ! in
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_var_inquire (id, var_num, var_name, data_type, num_elements,
1                           rec_variance, dim_variances, status)
INTEGER*4 id                                                        ! in
INTEGER*4 var_num                                                   ! in
CHARACTER var_name*(CDF_VAR_NAME_LEN)                               ! out
INTEGER*4 data_type                                                 ! out
INTEGER*4 num_elements                                              ! out
INTEGER*4 rec_variance                                              ! out
INTEGER*4 dim_variances(CDF_MAX_DIMS)                               ! out
INTEGER*4 status                                                    ! out

SUBROUTINE CDF_var_put (id, var_num, rec_num, indices, value, status)
INTEGER*4 id                                                        ! in
```

```
INTEGER*4 var_num                                                    ! in
INTEGER*4 rec_num                                                    ! in
INTEGER*4 indices(*)                                                 ! in
<type>    value                                                      ! in
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_var_get (id, var_num, rec_num, indices, value, status)
INTEGER*4 id                                                         ! in
INTEGER*4 var_num                                                    ! in
INTEGER*4 rec_num                                                    ! in
INTEGER*4 indices(*)                                                 ! in
<type>    value                                                      ! out
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_var_hyper_put (id, var_num, rec_start, rec_count, rec_interval,
1                             indices, counts, intervals, buffer, status)
INTEGER*4 id                                                         ! in
INTEGER*4 var_num                                                    ! in
INTEGER*4 rec_start                                                  ! in
INTEGER*4 rec_count                                                  ! in
INTEGER*4 rec_interval                                               ! in
INTEGER*4 indices(*)                                                 ! in
INTEGER*4 counts(*)                                                  ! in
INTEGER*4 intervals(*)                                               ! in
<type>    buffer                                                     ! in
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_var_hyper_get (id, var_num, rec_start, rec_count, rec_interval,
1                             indices, counts, intervals, buffer, status)
INTEGER*4 id                                                         ! in
INTEGER*4 var_num                                                    ! in
INTEGER*4 rec_start                                                  ! in
INTEGER*4 rec_count                                                  ! in
INTEGER*4 rec_interval                                               ! in
INTEGER*4 indices(*)                                                 ! in
INTEGER*4 counts(*)                                                  ! in
INTEGER*4 intervals(*)                                               ! in
<type>    buffer                                                     ! out
INTEGER*4 status                                                     ! out

SUBROUTINE CDF_var_close (id, var_num)
INTEGER*4 id                                                         ! in
INTEGER*4 var_num                                                    ! in
```

B.2 Internal Interface

```
INTEGER*4 FUNCTION CDF_lib (fnc, ..., status)
INTEGER*4 fnc                                                        ! in
.
```

```
.
.
INTEGER*4 status                                             ! out

   CLOSE_
             CDF_
             rVAR_
             zVAR_

   CONFIRM_
             ATTR_                    INTEGER*4 attr_num                ! out
             ATTR_EXISTENCE_          CHARACTER attr_name*(*)           ! in
             CDF_                     INTEGER*4 id                      ! out
             CDF_ACCESS_
             CDF_CACHESIZE_           INTEGER*4 num_buffers             ! out
             CDF_DECODING_            INTEGER*4 decoding                ! out
             CDF_NAME_                CHARACTER CDF_name*(CDF_PATHNAME_LEN)
                                                                        ! out
             CDF_NEGtoPOSfp0_MODE_    INTEGER*4 mode                    ! out
             CDF_READONLY_MODE_       INTEGER*4 mode                    ! out
             CDF_STATUS_              INTEGER*4 status                  ! out
             CDF_zMODE_               INTEGER*4 mode                    ! out
             COMPRESS_CACHESIZE_      INTEGER*4 num_buffers             ! out
             CURgENTRY_EXISTENCE_
             CURrENTRY_EXISTENCE_
             CURzENTRY_EXISTENCE_
             gENTRY_                  INTEGER*4 entry_num               ! out
             gENTRY_EXISTENCE_        INTEGER*4 entry_num               ! in
             rENTRY_                  INTEGER*4 entry_num               ! out
             rENTRY_EXISTENCE_        INTEGER*4 entry_num               ! in
             rVAR_                    INTEGER*4 var_num                 ! out
             rVAR_CACHESIZE_          INTEGER*4 num_buffers             ! out
             rVAR_EXISTENCE_          CHARACTER var_name*(*)            ! in
             rVAR_PADVALUE_
             rVAR_RESERVEPERCENT_     INTEGER*4 percent                 ! out
             rVAR_SEQPOS_             INTEGER*4 rec_num                 ! out
                                      INTEGER*4 indices(CDF_MAX_DIMS)   ! out
             rVARs_DIMCOUNTS_         INTEGER*4 counts(CDF_MAX_DIMS)    ! out
             rVARs_DIMINDICES_        INTEGER*4 indices(CDF_MAX_DIMS)   ! out
             rVARs_DIMINTERVALS_      INTEGER*4 intervals(CDF_MAX_DIMS) ! out
             rVARs_RECCOUNT_          INTEGER*4 rec_count               ! out
             rVARs_RECINTERVAL_       INTEGER*4 rec_interval            ! out
             rVARs_RECNUMBER_         INTEGER*4 rec_num                 ! out
             STAGE_CACHESIZE_         INTEGER*4 num_buffers             ! out
             zENTRY_                  INTEGER*4 entry_num               ! out
             zENTRY_EXISTENCE_        INTEGER*4 entry_num               ! in
             zVAR_                    INTEGER*4 var_num                 ! out
             zVAR_CACHESIZE_          INTEGER*4 num_buffers             ! out
             zVAR_DIMCOUNTS_          INTEGER*4 counts(CDF_MAX_DIMS)    ! out
             zVAR_DIMINDICES_         INTEGER*4 indices(CDF_MAX_DIMS)   ! out
             zVAR_DIMINTERVALS_       INTEGER*4 intervals(CDF_MAX_DIMS) ! out
```

```
        zVAR_EXISTENCE_         CHARACTER var_name*(*)           ! in
        zVAR_PADVALUE_
        zVAR_RECCOUNT_          INTEGER*4 rec_count              ! out
        zVAR_RECINTERVAL_       INTEGER*4 rec_interval           ! out
        zVAR_RECNUMBER_         INTEGER*4 rec_num                ! out
        zVAR_RESERVEPERCENT_    INTEGER*4 percent                ! out
        zVAR_SEQPOS_            INTEGER*4 rec_num                ! out
                                INTEGER*4 indices(CDF_MAX_DIMS)  ! out

CREATE_
        ATTR_                   CHARACTER attr_name*(*)          ! in
                                INTEGER*4 scope                  ! in
                                INTEGER*4 attr_num               ! out

        CDF_                    CHARACTER CDF_name*(*)           ! in
                                INTEGER*4 num_dims               ! in
                                INTEGER*4 dim_sizes(*)           ! in
                                INTEGER*4 id                     ! out

        rVAR_                   CHARACTER var_name*(*)           ! in
                                INTEGER*4 data_type              ! in
                                INTEGER*4 num_elements           ! in
                                INTEGER*4 rec_vary               ! in
                                INTEGER*4 dim_varys              ! in
                                INTEGER*4 var_num                ! out

        zVAR_                   CHARACTER var_name*(*)           ! in
                                INTEGER*4 data_type              ! in
                                INTEGER*4 num_elements           ! in
                                INTEGER*4 num_dims               ! in
                                INTEGER*4 dim_sizes(*)           ! in
                                INTEGER*4 rec_vary               ! in
                                INTEGER*4 dim_varys              ! in
                                INTEGER*4 var_num                ! out

DELETE_
        ATTR_
        CDF_
        gENTRY_
        rENTRY_
        rVAR_
        rVAR_RECORDS_           INTEGER*4 first_record           ! in
                                INTEGER*4 last_record            ! in
        zENTRY_
        zVAR_
        zVAR_RECORDS_           INTEGER*4 first_record           ! in
                                INTEGER*4 last_record            ! in

GET_
        ATTR_MAXgENTRY_         INTEGER*4 max_entry              ! out
        ATTR_MAXrENTRY_         INTEGER*4 max_entry              ! out
```

```
ATTR_MAXzENTRY_        INTEGER*4 max_entry                ! out
ATTR_NAME_             CHARACTER attr_name*(CDF_ATTR_NAME_LEN)
                                                          ! out
ATTR_NUMBER_           CHARACTER attr_name*(*)            ! in
                       INTEGER*4 attr_num                 ! out
ATTR_NUMgENTRIES_      INTEGER*4 num_entries              ! out
ATTR_NUMrENTRIES_      INTEGER*4 num_entries              ! out
ATTR_NUMzENTRIES_      INTEGER*4 num_entries              ! out
ATTR_SCOPE_            INTEGER*4 scope                    ! out
CDF_COMPRESSION_       INTEGER*4 c_type                   ! out
                       INTEGER*4 c_parms(CDF_MAX_PARMS)   ! out
                       INTEGER*4 c_pct                    ! out
CDF_COPYRIGHT_         CHARACTER copy_right*(CDF_COPYRIGHT_LEN)
                                                          ! out
CDF_ENCODING_          INTEGER*4 encoding                 ! out
CDF_FORMAT_            INTEGER*4 format                   ! out
CDF_INCREMENT_         INTEGER*4 increment                ! out
CDF_INFO_              CHARACTER CDF_name*(*)             ! in
                       INTEGER*4 c_type                   ! out
                       INTEGER*4 c_parms(CDF_MAX_PARMS)   ! out
                       INTEGER*4 c_size                   ! out
                       INTEGER*4 u_size                   ! out
CDF_MAJORITY_          INTEGER*4 majority                 ! out
CDF_NUMATTRS_          INTEGER*4 num_attrs                ! out
CDF_NUMgATTRS_         INTEGER*4 num_attrs                ! out
CDF_NUMrVARS_          INTEGER*4 num_vars                 ! out
CDF_NUMvATTRS_         INTEGER*4 num_attrs                ! out
CDF_NUMzVARS_          INTEGER*4 num_vars                 ! out
CDF_RELEASE_           INTEGER*4 release                  ! out
CDF_VERSION_           INTEGER*4 version                  ! out
DATATYPE_SIZE_         INTEGER*4 data_type                ! in
                       INTEGER*4 num_bytes                ! out
gENTRY_DATA_           <type> value                       ! out
gENTRY_DATATYPE_       INTEGER*4 data_type                ! out
gENTRY_NUMELEMS_       INTEGER*4 num_elements             ! out
LIB_COPYRIGHT_         CHARACTER copy_right*(CDF_COPYRIGHT_LEN)
                                                          ! out
LIB_INCREMENT_         INTEGER*4 increment                ! out
LIB_RELEASE_           INTEGER*4 release                  ! out
LIB_subINCREMENT_      CHARACTER subincrement*1           ! out
LIB_VERSION_           INTEGER*4 version                  ! out
rENTRY_DATA_           <type> value                       ! out
rENTRY_DATATYPE_       INTEGER*4 data_type                ! out
rENTRY_NUMELEMS_       INTEGER*4 num_elements             ! out
rVAR_ALLOCATEDFROM_    INTEGER*4 start_record             ! in
                       INTEGER*4 next_record              ! out
rVAR_ALLOCATEDTO_      INTEGER*4 start_record             ! in
                       INTEGER*4 last_record              ! out
rVAR_BLOCKINGFACTOR_   INTEGER*4 blocking_factor          ! out
rVAR_COMPRESSION_      INTEGER*4 c_type                   ! out
                       INTEGER*4 c_parms(CDF_MAX_PARMS)   ! out
```

```
                        INTEGER*4 c_pct                    ! out
rVAR_DATA_              <type> value                       ! out
rVAR_DATATYPE_          INTEGER*4 data_type                ! out
rVAR_DIMVARYS_          INTEGER*4 dim_varys(CDF_MAX_DIMS)  ! out
rVAR_HYPERDATA_         <type> buffer                      ! out
rVAR_MAXallocREC_       INTEGER*4 max_rec                  ! out
rVAR_MAXREC_            INTEGER*4 max_rec                  ! out
rVAR_NAME_              CHARACTER var_name*(CDF_VAR_NAME_LEN)
                                                           ! out
rVAR_nINDEXENTRIES_     INTEGER*4 num_entries              ! out
rVAR_nINDEXLEVELS_      INTEGER*4 num_levels               ! out
rVAR_nINDEXRECORDS_     INTEGER*4 num_records              ! out
rVAR_NUMallocRECS_      INTEGER*4 num_records              ! out
rVAR_NUMBER_            CHARACTER var_name*(*)             ! in
                        INTEGER*4 var_num                  ! out
rVAR_NUMELEMS_          INTEGER*4 num_elements             ! out
rVAR_NUMRECS_           INTEGER*4 num_records              ! out
rVAR_PADVALUE_          <type> value                       ! out
rVAR_RECVARY_           INTEGER*4 rec_vary                 ! out
rVAR_SEQDATA_           <type> value                       ! out
rVAR_SPARSEARRAYS_      INTEGER*4 s_arrays_type            ! out
                        INTEGER*4 s_arrays_parms(CDF_MAX_PARMS)
                                                           ! out
                        INTEGER*4 s_arrays_pct             ! out
rVAR_SPARSERECORDS_     INTEGER*4 s_records_type           ! out
rVARs_DIMSIZES_         INTEGER*4 dim_sizes(CDF_MAX_PARMS) ! out
rVARs_MAXREC_           INTEGER*4 max_rec                  ! out
rVARs_NUMDIMS_          INTEGER*4 num_dims                 ! out
rVARs_RECDATA_          INTEGER*4 num_vars                 ! in
                        INTEGER*4 var_nums(*)              ! in
                        <type> buffer                      ! out
STATUS_TEXT_            CHARACTER text*(CDF_STATUSTEXT_LEN) ! out
zENTRY_DATATYPE_        INTEGER*4 data_type                ! out
zENTRY_DATA_            <type> value                       ! out
zENTRY_NUMELEMS_        INTEGER*4 num_elements             ! out
zVAR_ALLOCATEDFROM_     INTEGER*4 start_record             ! in
                        INTEGER*4 next_record              ! out
zVAR_ALLOCATEDTO_       INTEGER*4 start_record             ! in
                        INTEGER*4 last_record              ! out
zVAR_BLOCKINGFACTOR_    INTEGER*4 blocking_factor          ! out
zVAR_COMPRESSION_       INTEGER*4 c_type                   ! out
                        INTEGER*4 c_parms(CDF_MAX_PARMS)   ! out
                        INTEGER*4 c_pct                    ! out
zVAR_DATA_              <type> value                       ! out
zVAR_DATATYPE_          INTEGER*4 data_type                ! out
zVAR_DIMSIZES_          INTEGER*4 dim_sizes(CDF_MAX_DIMS)  ! out
zVAR_DIMVARYS_          INTEGER*4 dim_varys(CDF_MAX_DIMS)  ! out
zVAR_HYPERDATA_         <type> buffer                      ! out
zVAR_MAXallocREC_       INTEGER*4 max_rec                  ! out
zVAR_MAXREC_            INTEGER*4 max_rec                  ! out
zVAR_NAME_              CHARACTER var_name*(CDF_VAR_NAME_LEN)
```

```
                                                                  ! out
          zVAR_nINDEXENTRIES_      INTEGER*4 num_entries          ! out
          zVAR_nINDEXLEVELS_       INTEGER*4 num_levels           ! out
          zVAR_nINDEXRECORDS_      INTEGER*4 num_records          ! out
          zVAR_NUMallocRECS_       INTEGER*4 num_records          ! out
          zVAR_NUMBER_             CHARACTER var_name*(*)         ! in
                                   INTEGER*4 var_num              ! out
          zVAR_NUMDIMS_            INTEGER*4 num_dims             ! out
          zVAR_NUMELEMS_           INTEGER*4 num_elements         ! out
          zVAR_NUMRECS_            INTEGER*4 num_records          ! out
          zVAR_PADVALUE_           <type> value                   ! out
          zVAR_RECVARY_            INTEGER*4 rec_vary             ! out
          zVAR_SEQDATA_            <type> value                   ! out
          zVAR_SPARSEARRAYS_       INTEGER*4 s_arrays_type        ! out
                                   INTEGER*4 s_arrays_parms(CDF_MAX_PARMS)
                                                                  ! out
                                   INTEGER*4 s_arrays_pct         ! out
          zVAR_SPARSERECORDS_      INTEGER*4 s_records_type       ! out
          zVARs_MAXREC_            INTEGER*4 max_rec              ! out
          zVARs_RECDATA_           INTEGER*4 num_vars             ! in
                                   INTEGER*4 var_nums(*)          ! in
                                   <type> buffer                  ! out

  NULL_
                                   INTEGER*4 status               ! out

  OPEN_
          CDF_                     CHARACTER CDF_name*(*)         ! in
                                   INTEGER*4 id                   ! out

  PUT_
          ATTR_NAME_               CHARACTER attr_name*(*)        ! in
          ATTR_SCOPE_              INTEGER*4 scope                ! in
          CDF_COMPRESSION_         INTEGER*4 c_type               ! in
                                   INTEGER*4 c_parms(*)           ! in
          CDF_ENCODING_            INTEGER*4 encoding             ! in
          CDF_FORMAT_              INTEGER*4 format               ! in
          CDF_MAJORITY_            INTEGER*4 majority             ! in
          gENTRY_DATA_             INTEGER*4 data_type            ! in
                                   INTEGER*4 num_elements         ! in
                                   <type> value                   ! in
          gENTRY_DATASPEC_         INTEGER*4 data_type            ! in
                                   INTEGER*4 num_elements         ! in
          rENTRY_DATA_             INTEGER*4 data_type            ! in
                                   INTEGER*4 num_elements         ! in
                                   <type> value                   ! in
          rENTRY_DATASPEC_         INTEGER*4 data_type            ! in
                                   INTEGER*4 num_elements         ! in
          rVAR_ALLOCATEBLOCK_      INTEGER*4 first_record         ! in
                                   INTEGER*4 last_record          ! in
          rVAR_ALLOCATERECS_       INTEGER*4 num_records          ! in
```

```
        rVAR_BLOCKINGFACTOR_     INTEGER*4 blocking_factor          ! in
        rVAR_COMPRESSION_        INTEGER*4 c_type                   ! in
                                 INTEGER*4 c_parms(*)               ! in
        rVAR_DATA_               <type> value                       ! in
        rVAR_DATASPEC_           INTEGER*4 data_type                ! in
                                 INTEGER*4 num_elements             ! in
        rVAR_DIMVARYS_           INTEGER*4 dim_varys(*)             ! in
        rVAR_HYPERDATA_          <type> buffer                      ! in
        rVAR_INITIALRECS_        INTEGER*4 num_records              ! in
        rVAR_NAME_               CHARACTER var_name*(*)             ! in
        rVAR_PADVALUE_           <type> value                       ! in
        rVAR_RECVARY_            INTEGER*4 rec_vary                 ! in
        rVAR_SEQDATA_            <type> value                       ! in
        rVAR_SPARSEARRAYS_       INTEGER*4 s_arrays_type            ! in
                                 INTEGER*4 s_arrays_parms(*)        ! in
        rVAR_SPARSERECORDS_      INTEGER*4 s_records_type           ! in
        rVARs_RECDATA_           INTEGER*4 num_vars                 ! in
                                 INTEGER*4 var_nums(*)              ! in
                                 <type> buffer                      ! in
        zENTRY_DATA_             INTEGER*4 data_type                ! in
                                 INTEGER*4 num_elements             ! in
                                 <type> value                       ! in
        zENTRY_DATASPEC_         INTEGER*4 data_type                ! in
                                 INTEGER*4 num_elements             ! in
        zVAR_ALLOCATEBLOCK_      INTEGER*4 first_record             ! in
                                 INTEGER*4 last_record              ! in
        zVAR_ALLOCATERECS_       INTEGER*4 num_records              ! in
        zVAR_BLOCKINGFACTOR_     INTEGER*4 blocking_factor          ! in
        zVAR_COMPRESSION_        INTEGER*4 c_type                   ! in
                                 INTEGER*4 c_parms(*)               ! in
        zVAR_DATA_               <type> value                       ! in
        zVAR_DATASPEC_           INTEGER*4 data_type                ! in
                                 INTEGER*4 num_elements             ! in
        zVAR_DIMVARYS_           INTEGER*4 dim_varys(*)             ! in
        zVAR_INITIALRECS_        INTEGER*4 num_records              ! in
        zVAR_HYPERDATA_          <type> buffer                      ! in
        zVAR_NAME_               CHARACTER var_name*(*)             ! in
        zVAR_PADVALUE_           <type> value                       ! in
        zVAR_RECVARY_            INTEGER*4 rec_vary                 ! in
        zVAR_SEQDATA_            <type> value                       ! in
        zVAR_SPARSEARRAYS_       INTEGER*4 s_arrays_type            ! in
                                 INTEGER*4 s_arrays_parms(*)        ! in
        zVAR_SPARSERECORDS_      INTEGER*4 s_records_type           ! in
        zVARs_RECDATA_           INTEGER*4 num_vars                 ! in
                                 INTEGER*4 var_nums(*)              ! in
                                 <type> buffer                      ! in

SELECT_
        ATTR_                    INTEGER*4 attr_num                 ! in
        ATTR_NAME_               CHARACTER attr_name*(*)            ! in
        CDF_                     INTEGER*4 id                       ! in
```

```
CDF_CACHESIZE_          INTEGER*4 num_buffers            ! in
CDF_DECODING_           INTEGER*4 decoding               ! in
CDF_NEGtoPOSfp0_MODE_   INTEGER*4 mode                   ! in
CDF_READONLY_MODE_      INTEGER*4 mode                   ! in
CDF_SCRATCHDIR_         CHARACTER dir_name*(*)           ! in
CDF_STATUS_             INTEGER*4 status                 ! in
CDF_zMODE_              INTEGER*4 mode                   ! in
COMPRESS_CACHESIZE_     INTEGER*4 num_buffers            ! in
gENTRY_                 INTEGER*4 entry_num              ! in
rENTRY_                 INTEGER*4 entry_num              ! in
rENTRY_NAME_            CHARACTER var_name*(*)           ! in
rVAR_                   INTEGER*4 var_num                ! in
rVAR_CACHESIZE_         INTEGER*4 num_buffers            ! in
rVAR_NAME_              CHARACTER var_name*(*)           ! in
rVAR_RESERVEPERCENT_    INTEGER*4 percent                ! in
rVAR_SEQPOS_            INTEGER*4 rec_num                ! in
                        INTEGER*4 indices(*)             ! in
rVARs_CACHESIZE_        INTEGER*4 num_buffers            ! in
rVARs_DIMCOUNTS_        INTEGER*4 counts(*)              ! in
rVARs_DIMINDICES_       INTEGER*4 indices(*)             ! in
rVARs_DIMINTERVALS_     INTEGER*4 intervals(*)           ! in
rVARs_RECCOUNT_         INTEGER*4 rec_count              ! in
rVARs_RECINTERVAL_      INTEGER*4 rec_interval           ! in
rVARs_RECNUMBER_        INTEGER*4 rec_num                ! in
STAGE_CACHESIZE_        INTEGER*4 num_buffers            ! in
zENTRY_                 INTEGER*4 entry_num              ! in
zENTRY_NAME_            CHARACTER var_name*(*)           ! in
zVAR_                   INTEGER*4 var_num                ! in
zVAR_CACHESIZE_         INTEGER*4 num_buffers            ! in
zVAR_NAME_              CHARACTER var_name*(*)           ! in
zVAR_DIMCOUNTS_         INTEGER*4 counts(*)              ! in
zVAR_DIMINDICES_        INTEGER*4 indices(*)             ! in
zVAR_DIMINTERVALS_      INTEGER*4 intervals(*)           ! in
zVAR_RECCOUNT_          INTEGER*4 rec_count              ! in
zVAR_RECINTERVAL_       INTEGER*4 rec_interval           ! in
zVAR_RECNUMBER_         INTEGER*4 rec_num                ! in
zVAR_RESERVEPERCENT_    INTEGER*4 percent                ! in
zVAR_SEQPOS_            INTEGER*4 rec_num                ! in
                        INTEGER*4 indices(*)             ! in
zVARs_CACHESIZE_        INTEGER*4 num_buffers            ! in
zVARs_RECNUMBER_        INTEGER*4 rec_num                ! in
```

B.3 EPOCH Utility Routines

```
SUBROUTINE compute_EPOCH (year, month, day, hour, minute, second, msec, epoch)
INTEGER*4 year                                                        ! in
INTEGER*4 month                                                       ! in
INTEGER*4 day                                                         ! in
INTEGER*4 hour                                                        ! in
```

```
INTEGER*4 minute                                                        ! in
INTEGER*4 second                                                        ! in
INTEGER*4 msec                                                          ! in
REAL*8    epoch                                                         ! out

SUBROUTINE EPOCH_breakdown (epoch, year, month, day, hour, minute, second,
1                            msec)
REAL*8    epoch                                                         ! in
INTEGER*4 year                                                          ! out
INTEGER*4 month                                                         ! out
INTEGER*4 day                                                           ! out
INTEGER*4 hour                                                          ! out
INTEGER*4 minute                                                        ! out
INTEGER*4 second                                                        ! out
INTEGER*4 msec                                                          ! out

SUBROUTINE encode_EPOCH (epoch, ep_string)
REAL*8    epoch                                                         ! in
CHARACTER ep_string*(EPOCH_STRING_LEN)                                  ! out

SUBROUTINE encode_EPOCH1 (epoch, ep_string)
REAL*8    epoch                                                         ! in
CHARACTER ep_string*(EPOCH1_STRING_LEN)                                 ! out

SUBROUTINE encode_EPOCH2 (epoch, ep_string)
REAL*8    epoch                                                         ! in
CHARACTER ep_string*(EPOCH2_STRING_LEN)                                 ! out

SUBROUTINE encode_EPOCHx (epoch, format, encoded)
REAL*8    epoch                                                         ! in
CHARACTER format(EPOCHx_FORMAT_MAX)                                     ! in
CHARACTER encoded(EPOCHx_STRING_MAX)                                    ! out

SUBROUTINE parse_EPOCH (ep_string, epoch)
CHARACTER ep_string*(EPOCH_STRING_LEN)                                  ! in
REAL*8    epoch                                                         ! out

SUBROUTINE parse_EPOCH1 (ep_string, epoch)
CHARACTER ep_string*(EPOCH1_STRING_LEN)                                 ! in
REAL*8    epoch                                                         ! out

SUBROUTINE parse_EPOCH2 (ep_string, epoch)
CHARACTER ep_string*(EPOCH2_STRING_LEN)                                 ! in
REAL*8    epoch                                                         ! out

SUBROUTINE parse_EPOCH3 (ep_string, epoch)
CHARACTER ep_string*(EPOCH3_STRING_LEN)                                 ! in
REAL*8    epoch                                                         ! out
```

Index