

# **CDF**

## **Fortran Reference Manual**

Version 3.2, October 22, 2008

Space Physics Data Facility  
NASA / Goddard Space Flight Center

Copyright © 2008  
Space Physics Data Facility  
NASA/Goddard Space Flight Center  
Greenbelt, Maryland 20771 (U.S.A.)

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

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

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

Internet - [cdfsupport@listserv.gsfc.nasa.gov](mailto:cdfsupport@listserv.gsfc.nasa.gov)

# Contents

<b><u>1</u></b>	<b><u>Compiling</u></b> .....	<b>1</b>
1.1	<u>VMS/OpenVMS Systems</u> .....	2
1.2	<u>UNIX Systems</u> .....	2
1.3	<u>Windows NT/2000/XP Systems, Digital Visual Fortran</u> .....	2
<b><u>2</u></b>	<b><u>Linking</u></b> .....	<b>5</b>
2.1	<u>VAX/VMS &amp; VAX/OpenVMS Systems</u> .....	5
2.2	<u>DEC Alpha/OpenVMS Systems</u> .....	5
2.3	<u>UNIX Systems</u> .....	6
2.3.1	<u>Combining the Compile and Link</u> .....	6
2.4	<u>Windows NT/2000/XP Systems, Digital Visual Fortran</u> .....	6
<b><u>3</u></b>	<b><u>Linking Shared CDF Library</u></b> .....	<b>9</b>
3.1	<u>VAX (VMS &amp; OpenVMS)</u> .....	9
3.2	<u>DEC Alpha (OpenVMS)</u> .....	10
3.3	<u>SUN (SOLARIS)</u> .....	10
3.4	<u>HP 9000 (HP-UX)</u> .....	11
3.5	<u>IBM RS6000 (AIX)</u> .....	11
3.6	<u>DEC Alpha (OSF/1)</u> .....	11
3.7	<u>SGi (IRIX 6.x)</u> .....	11
3.8	<u>Linux (PC &amp; Power PC)</u> .....	11
3.9	<u>Windows (NT/2000/XP)</u> .....	12
<b><u>4</u></b>	<b><u>Programming Interface</u></b> .....	<b>13</b>
4.1	<u>Argument Passing</u> .....	13
4.2	<u>Item Referencing</u> .....	14
4.3	<u>Status Code Constants</u> .....	14
4.4	<u>CDF Formats</u> .....	14
4.5	<u>CDF Data Types</u> .....	14
4.6	<u>Data Encodings</u> .....	15
4.7	<u>Data Decodings</u> .....	16
4.8	<u>Variable Majorities</u> .....	17
4.9	<u>Record/Dimension Variances</u> .....	18
4.10	<u>Compressions</u> .....	18
4.11	<u>Sparseness</u> .....	19
4.11.1	<u>Sparse Records</u> .....	19
4.11.2	<u>Sparse Arrays</u> .....	19
4.12	<u>Attribute Scopes</u> .....	19
4.13	<u>Read-Only Modes</u> .....	20
4.14	<u>zModes</u> .....	20
4.15	<u>-0.0 to 0.0 Modes</u> .....	20
4.16	<u>Operational Limits</u> .....	20
4.17	<u>Limits of Names and Other Character Strings</u> .....	21
4.18	<u>Backward File Compatibility with CDF 2.7</u> .....	21
4.19	<u>Checksum</u> .....	22
<b><u>5</u></b>	<b><u>Standard Interface (Original)</u></b> .....	<b>25</b>

<a href="#">5.1</a>	<a href="#">CDF attr create</a>	25
<a href="#">5.1.1</a>	<a href="#">Example(s)</a>	26
<a href="#">5.2</a>	<a href="#">CDF attr entry inquire</a>	26
<a href="#">5.2.1</a>	<a href="#">Example(s)</a>	27
<a href="#">5.3</a>	<a href="#">CDF attr get</a>	28
<a href="#">5.3.1</a>	<a href="#">Example(s)</a>	28
<a href="#">5.4</a>	<a href="#">CDF attr inquire</a>	29
<a href="#">5.4.1</a>	<a href="#">Example(s)</a>	30
<a href="#">5.5</a>	<a href="#">CDF attr num</a>	31
<a href="#">5.5.1</a>	<a href="#">Example(s)</a>	31
<a href="#">5.6</a>	<a href="#">CDF attr put</a>	32
<a href="#">5.6.1</a>	<a href="#">Example(s)</a>	33
<a href="#">5.7</a>	<a href="#">CDF attr rename</a>	33
<a href="#">5.7.1</a>	<a href="#">Example(s)</a>	34
<a href="#">5.8</a>	<a href="#">CDF close</a>	34
<a href="#">5.8.1</a>	<a href="#">Example(s)</a>	35
<a href="#">5.9</a>	<a href="#">CDF create</a>	35
<a href="#">5.9.1</a>	<a href="#">Example(s)</a>	36
<a href="#">5.10</a>	<a href="#">CDF delete</a>	37
<a href="#">5.10.1</a>	<a href="#">Example(s)</a>	37
<a href="#">5.11</a>	<a href="#">CDF doc</a>	37
<a href="#">5.11.1</a>	<a href="#">Example(s)</a>	38
<a href="#">5.12</a>	<a href="#">CDF error</a>	39
<a href="#">5.12.1</a>	<a href="#">Example(s)</a>	39
<a href="#">5.13</a>	<a href="#">CDF getrvarsrecorddata</a>	40
<a href="#">5.13.1</a>	<a href="#">Example(s)</a>	41
<a href="#">5.14</a>	<a href="#">CDF getzvarsrecorddata</a>	42
<a href="#">5.14.1</a>	<a href="#">Example(s)</a>	42
<a href="#">5.15</a>	<a href="#">CDF inquire</a>	44
<a href="#">5.15.1</a>	<a href="#">Example(s)</a>	45
<a href="#">5.16</a>	<a href="#">CDF open</a>	45
<a href="#">5.16.1</a>	<a href="#">Example(s)</a>	46
<a href="#">5.17</a>	<a href="#">CDF putrvarsrecorddata</a>	46
<a href="#">5.17.1</a>	<a href="#">Example(s)</a>	47
<a href="#">5.18</a>	<a href="#">CDF putzvarsrecorddata</a>	48
<a href="#">5.18.1</a>	<a href="#">Example(s)</a>	49
<a href="#">5.19</a>	<a href="#">CDF var close</a>	50
<a href="#">5.19.1</a>	<a href="#">Example(s)</a>	51
<a href="#">5.20</a>	<a href="#">CDF var create</a>	51
<a href="#">5.20.1</a>	<a href="#">Example(s)</a>	52
<a href="#">5.21</a>	<a href="#">CDF var get</a>	53
<a href="#">5.21.1</a>	<a href="#">Example(s)</a>	53
<a href="#">5.22</a>	<a href="#">CDF var hyper get</a>	54
<a href="#">5.22.1</a>	<a href="#">Example(s)</a>	55
<a href="#">5.23</a>	<a href="#">CDF var hyper put</a>	56
<a href="#">5.23.1</a>	<a href="#">Example(s)</a>	57
<a href="#">5.24</a>	<a href="#">CDF var inquire</a>	58
<a href="#">5.24.1</a>	<a href="#">Example(s)</a>	59
<a href="#">5.25</a>	<a href="#">CDF var num</a>	59
<a href="#">5.25.1</a>	<a href="#">Example(s)</a>	60
<a href="#">5.26</a>	<a href="#">CDF var put</a>	61
<a href="#">5.26.1</a>	<a href="#">Example(s)</a>	61
<a href="#">5.27</a>	<a href="#">CDF var rename</a>	62
<a href="#">5.27.1</a>	<a href="#">Example(s)</a>	62

## **[6](#) [Standard Interface \(New\)](#).....65**

6.1	<a href="#">Library</a> .....	65
6.1.1	<a href="#">CDF get datatype size</a> .....	66
6.1.2	<a href="#">CDF get lib copyright</a> .....	66
6.1.3	<a href="#">CDF get lib version</a> .....	67
6.1.4	<a href="#">CDF get status text</a> .....	68
6.2	<a href="#">CDF</a> .....	69
6.2.1	<a href="#">CDF close cdf</a> .....	69
6.2.2	<a href="#">CDF create cdf</a> .....	70
6.2.3	<a href="#">CDF delete cdf</a> .....	71
6.2.4	<a href="#">CDF get cachesize</a> .....	72
6.2.5	<a href="#">CDF get checksum</a> .....	73
6.2.6	<a href="#">CDF get compress cachesize</a> .....	74
6.2.7	<a href="#">CDF get compression</a> .....	75
6.2.8	<a href="#">CDF get compression info</a> .....	76
6.2.9	<a href="#">CDF get copyright</a> .....	76
6.2.10	<a href="#">CDF get decoding</a> .....	77
6.2.11	<a href="#">CDF get encoding</a> .....	78
6.2.12	<a href="#">CDF get format</a> .....	79
6.2.13	<a href="#">CDF get majority</a> .....	80
6.2.14	<a href="#">CDF get name</a> .....	81
6.2.15	<a href="#">CDF get negtoposfp0 mode</a> .....	81
6.2.16	<a href="#">CDF get readonly mode</a> .....	82
6.2.17	<a href="#">CDF get stage cachesize</a> .....	83
6.2.18	<a href="#">CDF get version</a> .....	84
6.2.19	<a href="#">CDF get zmode</a> .....	85
6.2.20	<a href="#">CDF inquire cdf</a> .....	86
6.2.21	<a href="#">CDF open cdf</a> .....	87
6.2.22	<a href="#">CDF set cachesize</a> .....	88
6.2.23	<a href="#">CDF set checksum</a> .....	89
6.2.24	<a href="#">CDF set compress cachesize</a> .....	90
6.2.25	<a href="#">CDF set compression</a> .....	91
6.2.26	<a href="#">CDF set decoding</a> .....	92
6.2.27	<a href="#">CDF set encoding</a> .....	93
6.2.28	<a href="#">CDF set format</a> .....	93
6.2.29	<a href="#">CDF set majority</a> .....	94
6.2.30	<a href="#">CDF set negtoposfp0 mode</a> .....	95
6.2.31	<a href="#">CDF set readonly mode</a> .....	96
6.2.32	<a href="#">CDF set stage cachesize</a> .....	97
6.2.33	<a href="#">CDF set zmode</a> .....	98
6.3	<a href="#">Variable</a> .....	98
6.3.1	<a href="#">CDF close zvar</a> .....	98
6.3.2	<a href="#">CDF confirm zvar existence</a> .....	99
6.3.3	<a href="#">CDF confirm zvar padvalue exist</a> .....	100
6.3.4	<a href="#">CDF create zvar</a> .....	101
6.3.5	<a href="#">CDF delete zvar</a> .....	103
6.3.6	<a href="#">CDF delete zvar recs</a> .....	104
6.3.7	<a href="#">CDF get num zvars</a> .....	105
6.3.8	<a href="#">CDF get var num</a> .....	106
6.3.9	<a href="#">CDF get vars maxwrittenrecnums</a> .....	107
6.3.10	<a href="#">CDF get zvar allocrecs</a> .....	108
6.3.11	<a href="#">CDF get zvar blockingfactor</a> .....	109
6.3.12	<a href="#">CDF get zvar cachesize</a> .....	110
6.3.13	<a href="#">CDF get zvar compression</a> .....	111
6.3.14	<a href="#">CDF get zvar data</a> .....	112
6.3.15	<a href="#">CDF get zvar datatype</a> .....	113
6.3.16	<a href="#">CDF get zvar dimsizes</a> .....	114

6.3.17	<a href="#">CDF get zvar dimvariances</a>	115
6.3.18	<a href="#">CDF get zvar maxallocrecnum</a>	116
6.3.19	<a href="#">CDF get zvar maxwrittenrecnum</a>	117
6.3.20	<a href="#">CDF get zvar name</a>	118
6.3.21	<a href="#">CDF get zvar numdims</a>	118
6.3.22	<a href="#">CDF get zvar numelems</a>	119
6.3.23	<a href="#">CDF get zvar numrecs written</a>	120
6.3.24	<a href="#">CDF get zvar padvalue</a>	121
6.3.25	<a href="#">CDF get zvar recorddata</a>	122
6.3.26	<a href="#">CDF get zvar recvariance</a>	123
6.3.27	<a href="#">CDF get zvar reservepercent</a>	124
6.3.28	<a href="#">CDF get zvar seqdata</a>	125
6.3.29	<a href="#">CDF get zvar seqpos</a>	126
6.3.30	<a href="#">CDF get zvars maxwrittenrecnum</a>	127
6.3.31	<a href="#">CDF get zvar sparserecords</a>	128
6.3.32	<a href="#">CDF get zvars recorddata</a>	129
6.3.33	<a href="#">CDF hyper get zvar data</a>	130
6.3.34	<a href="#">CDF hyper put zvar data</a>	132
6.3.35	<a href="#">CDF inquire zvar</a>	134
6.3.36	<a href="#">CDF put zvar data</a>	136
6.3.37	<a href="#">CDF put zvar recorddata</a>	137
6.3.38	<a href="#">CDF put zvar seqdata</a>	138
6.3.39	<a href="#">CDF put zvars recorddata</a>	139
6.3.40	<a href="#">CDF rename zvar</a>	142
6.3.41	<a href="#">CDF set zvar allocblockrecs</a>	143
6.3.42	<a href="#">CDF set zvar allocrecs</a>	144
6.3.43	<a href="#">CDF set zvar blockingfactor</a>	144
6.3.44	<a href="#">CDF set zvar cachesize</a>	145
6.3.45	<a href="#">CDF set zvar compression</a>	146
6.3.46	<a href="#">CDF set zvar dataspec</a>	147
6.3.47	<a href="#">CDF set zvar dimvariances</a>	148
6.3.48	<a href="#">CDF set zvar initialrecs</a>	149
6.3.49	<a href="#">CDF set zvar padvalue</a>	150
6.3.50	<a href="#">CDF set zvar recvariance</a>	151
6.3.51	<a href="#">CDF set zvar reservepercent</a>	152
6.3.52	<a href="#">CDF set zvars cachesize</a>	153
6.3.53	<a href="#">CDF set zvar seqpos</a>	154
6.3.54	<a href="#">CDF set zvar sparserecords</a>	155
6.4	<a href="#">Attributes/Entries</a>	156
6.4.1	<a href="#">CDF confirm attr existence</a>	156
6.4.2	<a href="#">CDF confirm gentry existence</a>	157
6.4.3	<a href="#">CDF confirm reentry existence</a>	158
6.4.4	<a href="#">CDF confirm zentry existence</a>	158
6.4.5	<a href="#">CDF create attr</a>	159
6.4.6	<a href="#">CDF delete attr</a>	161
6.4.7	<a href="#">CDF delete attr gentry</a>	161
6.4.8	<a href="#">CDF delete attr reentry</a>	162
6.4.9	<a href="#">CDF delete attr zentry</a>	163
6.4.10	<a href="#">CDF get attr gentry</a>	164
6.4.11	<a href="#">CDF get attr gentry datatype</a>	165
6.4.12	<a href="#">CDF get attr gentry numelems</a>	166
6.4.13	<a href="#">CDF get attr max gentry</a>	167
6.4.14	<a href="#">CDF get attr max reentry</a>	168
6.4.15	<a href="#">CDF get attr max zentry</a>	169
6.4.16	<a href="#">CDF get attr name</a>	170
6.4.17	<a href="#">CDF get attr num</a>	171

6.4.18	<a href="#">CDF get attr num gentries</a>	172
6.4.19	<a href="#">CDF get attr num rentries</a>	173
6.4.20	<a href="#">CDF get attr num zentries</a>	174
6.4.21	<a href="#">CDF get attr reentry</a>	174
6.4.22	<a href="#">CDF get attr reentry datatype</a>	176
6.4.23	<a href="#">CDF get attr reentry numelems</a>	177
6.4.24	<a href="#">CDF get attr scope</a>	178
6.4.25	<a href="#">CDF get attr zentry</a>	179
6.4.26	<a href="#">CDF get attr zentry datatype</a>	180
6.4.27	<a href="#">CDF get attr zentry numelems</a>	181
6.4.28	<a href="#">CDF get num attrs</a>	182
6.4.29	<a href="#">CDF get num gattrs</a>	183
6.4.30	<a href="#">CDF get num vattrs</a>	184
6.4.31	<a href="#">CDF inquire attr</a>	185
6.4.32	<a href="#">CDF inquire attr gentry</a>	187
6.4.33	<a href="#">CDF inquire attr reentry</a>	188
6.4.34	<a href="#">CDF inquire attr zentry</a>	190
6.4.35	<a href="#">CDF put attr gentry</a>	191
6.4.36	<a href="#">CDF put attr reentry</a>	192
6.4.37	<a href="#">CDF put attr zentry</a>	194
6.4.38	<a href="#">CDF rename attr</a>	195
6.4.39	<a href="#">CDF set attr gentry dataspec</a>	196
6.4.40	<a href="#">CDF set attr reentry dataspec</a>	197
6.4.41	<a href="#">CDF set attr scope</a>	198
6.4.42	<a href="#">CDF set attr zentry dataspec</a>	199

## **7 Internal Interface – CDF lib .....201**

7.1	<a href="#">Example(s)</a>	201
7.2	<a href="#">Current Objects/States (Items)</a>	203
7.3	<a href="#">Returned Status</a>	206
7.4	<a href="#">Indentation/Style</a>	207
7.5	<a href="#">Syntax</a>	207
7.5.1	<a href="#">Macintosh, MPW Fortran</a>	208
7.6	<a href="#">Operations</a>	209
7.7	<a href="#">More Examples</a>	266
7.7.1	<a href="#">Creation</a>	266
7.7.2	<a href="#">zVariable Creation (Character Data Type)</a>	267
7.7.3	<a href="#">Hyper Read with Subsampling</a>	267
7.7.4	<a href="#">Attribute Renaming</a>	269
7.7.5	<a href="#">Sequential Access</a>	269
7.7.6	<a href="#">Attribute rEntry Writes</a>	270
7.7.7	<a href="#">Multiple zVariable Write</a>	270

## **8 Interpreting CDF Status Codes .....272**

## **9 EPOCH Utility Routines .....274**

9.1	<a href="#">compute EPOCH</a>	274
9.2	<a href="#">EPOCH breakdown</a>	275
9.3	<a href="#">encode EPOCH</a>	275
9.4	<a href="#">encode EPOCH1</a>	275
9.5	<a href="#">encode EPOCH2</a>	276
9.6	<a href="#">encode EPOCH3</a>	276
9.7	<a href="#">encode EPOCHx</a>	276
9.8	<a href="#">parse EPOCH</a>	277

<a href="#">9.9</a>	<a href="#">parse EPOCH1</a> .....	277
<a href="#">9.10</a>	<a href="#">parse EPOCH2</a> .....	278
<a href="#">9.11</a>	<a href="#">parse EPOCH3</a> .....	278
<a href="#">9.12</a>	<a href="#">compute EPOCH16</a> .....	278
<a href="#">9.13</a>	<a href="#">EPOCH16 breakdown</a> .....	279
<a href="#">9.14</a>	<a href="#">encode EPOCH16</a> .....	279
<a href="#">9.15</a>	<a href="#">encode EPOCH16 1</a> .....	279
<a href="#">9.16</a>	<a href="#">encode EPOCH16 2</a> .....	280
<a href="#">9.17</a>	<a href="#">encode EPOCH16 3</a> .....	280
<a href="#">9.18</a>	<a href="#">encode EPOCH16 x</a> .....	280
<a href="#">9.19</a>	<a href="#">parse EPOCH16</a> .....	281
<a href="#">9.20</a>	<a href="#">parse EPOCH16 1</a> .....	282
<a href="#">9.21</a>	<a href="#">parse EPOCH16 2</a> .....	282
<a href="#">9.22</a>	<a href="#">parse EPOCH16 3</a> .....	282



# Chapter 1

## 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 files, DEFINITIONS.COM, provided with the CDF distribution. On UNIX systems (including Mac OS X) an environment variable, CDF\_INC, that serves the same purpose is defined in the definitions files 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 files on those systems. The location of cdf.inc 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 files 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 files 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 files 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 files declares the FUNCTIONS 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) that 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.<sup>1</sup>

### Digital Visual Fortran<sup>1</sup>

On Windows NT/2000/XP systems using Digital Visual Fortran, the following lines would be included at the top of each routine/source files:

```
.  
. (PROGRAM, SUBROUTINE, or FUNCTION statement)  
.
```

---

<sup>1</sup> Normally, you need to run DFVARS.BAT in bin directory to set up the proper environment from the command prompt.

```
INCLUDE 'cdfdvf.inc'  
INCLUDE 'cdfdf.inc'
```

The include files `cdfdvf.inc` contains an `INTERFACE` statement for each subroutine/function in the CDF library. Including this files is absolutely essential no matter if you are using the Internal Interface (CDF lib) or Standard Interface (e.g., CDF create, etc.) `cdfdvf.inc` is located in the same directory as `cdf.inc`. The include file `cdfdf.inc` is similar to `cdfdf.inc`, with some statements commented out for Digital Visual Fortran compiler.

## 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:<sup>2</sup>

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

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 Windows NT/2000/XP Systems, Digital Visual Fortran

An example of the command to compile a source file on Windows NT/95/98 systems using Digital Visual Fortran would be as follows:<sup>3</sup>

```
> DF /c /iface:nomixed_strfilesn_arg /nowarn /optimize:0 /I<inc-path> <source-name>.f
```

---

<sup>2</sup> The name of the Fortran compiler may be different depending on the flavor of UNIX being used.

<sup>3</sup> This example assumes you have properly set the MS-DOS environment variables used by the Digital Visual Fortran compiler.

where <source-name>.f is the name of the source file being compiled (the .f extension is required) and <inc-path> is the file name of the directory containing cdfdvf.inc and cdfdf.inc. You will need to know where on your system cdfdvf.inc and cdfdf.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 /iface:nomixed str len arg option specifies that Fortran string arguments will have their string lengths appended to the end of the argument list by the compiler.

The /optimize:0 option specifies that no code optimization is done. We had a problem using the default optimization.

The /nowarn option specifies that no warning messages will be given.

You can run the batch files, DFVARS.BAT, came with the Digital Visual Fortran, to set them up.



# Chapter 2

## 2 Linking

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

### 2.1 VAX/VMS & VAX/OpenVMS Systems

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

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

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

It may also be necessary to specify SYSS\$LIBRARY:VAXCTRL/LIBRARY at the end of the LINK command if your system does not properly define LNK\$LIBRARY (or LNK\$LIBRARY\_1, etc.).

### 2.2 DEC Alpha/OpenVMS Systems

---

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

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

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

where <object-file(s)> is your application's object module(s) (the .OBJ extension is not necessary) and <crtl> is VAXCRTLD 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 to link your application with the CDF library (libcdf.a) on UNIX flavored systems 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 Windows NT/2000/XP Systems, Digital Visual 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 Visual 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 Windows NT/95/98 systems using Digital Visual Fortran and Microsoft Visual C++ would be as follows:<sup>2</sup>

---

<sup>2</sup> This example assumes you have properly set the MS-DOS environment variables (e.g., LIB should be set to include directories that contain C's LIBC.LIB and Fortran's DFOR.LIB.)

```
> LINK <objs> <lib-path>libcdf.lib /out:<name.exe> /nodefaultlib:libcd
```

where <objs> is your application's object module(s) (the .obj extension is necessary); <name.exe> is the name of the executable file to be created 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.

The /nodefaultlib:libcd option specifies that the LIBCD.LIB is to be ignored during the library search for resolving references.





# Chapter 3

## 3 Linking Shared CDF Library

A shareable version of the CDF library is also available on VMS systems, some flavors of UNIX<sup>1</sup>, Windows NT/95/98<sup>2</sup> and Macintosh.<sup>3</sup> The shared version is put in the same directory as the non-shared version and is named as follows:

<u>Machine/Operating System</u>	<u>Shared CDF Library</u>
VAX (VMS & OpenVMS)	LIBCDF.EXE
DEC Alpha (OpenVMS)	LIBCDF.EXE
Sun (SOLARIS)	libcdf.so
HP 9000 (HP-UX) <sup>4</sup>	libcdf.sl
IBM RS6000 (AIX) <sup>4</sup>	libcdf.o
DEC Alpha (OSF/1)	libcdf.so
SGi (6.x)	libcdf.so
Linux (PC & Power PC)	libcdf.so
Windows NT/2000/XP	dlldcf.dll
Macintosh OS X <sup>4</sup>	libcdf.dylib

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
```

---

<sup>1</sup> On UNIX systems, when executing a program linked to the shared CDF library, the environment variable LD\_LIBRARY\_PATH must be set to include the directory containing libcdf.so or libcdf.sl.

<sup>2</sup> When executing a program linked to the dynamically linked CDF library (DLL), the environment variable PATH must be set to include the directory containing dlldcf.dll.

<sup>3</sup> On Mac systems, when executing a program linked to the shared CDF library, dlldcf.ppc or dlldcf.68k must be copied into System's Extension folder.

<sup>4</sup> Not yet tested. Contact [CDFsupport@listserv.gsfc.nasa.gov](mailto:CDFsupport@listserv.gsfc.nasa.gov) to coordinate the test.

```

SYS$SHARE:VAXCTRL/SHAREABLE
<Control-Z>
$ DEASSIGN CDF$LIBCDFEXE

```

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

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

```

$ LINK <object-file(s)>, SYS$INPUT:/OPTIONS
SYS$SHARE:LIBCDF/SHAREABLE
SYS$SHARE:VAXCTRL/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 VAXCTRL if your CDF distribution is built for a default double-precision floating-point representation of G\_FLOAT or VAXCTRLD for a default of D\_FLOAT or VAXCTRLT for a default of IEEE\_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 (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}.

### 3.4 HP 9000 (HP-UX)

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

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

### 3.5 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}.

### 3.6 DEC Alpha (OSF/1)

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

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

### 3.7 SGI (IRIX 6.x)

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

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

### 3.8 Linux (PC & Power PC)

```
% g77 -o <exe-file> <object-file(s)>.o ${CDF_LIB}/libcdf.so -lm -lc
```

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

### 3.9 Windows (NT/2000/XP)

```
% link /out:<exe-file>.exe <object-file(s)>.obj <lib-path>dllcdf.lib  
/nodefaultlib:libcd
```

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

# Chapter 4

## 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 INCLUDED 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

These constants are of type INTEGER\*4.

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

## 4.4 CDF Formats

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

## 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_EPOCH16	two 8-byte, floating point.

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

**NOTE:** When using a DEC Alpha running OSF/1 keep in mind that a long is 8 bytes and that an int is 4 bytes. Use int C variables with the CDF data types CDF\_INT4 and CDF\_UINT4 rather than long C variables.

**NOTE:** When using an PC (MS-DOS) keep in mind that an int is 2 bytes and that a long is 4 bytes. Use long C variables with the CDF data types CDF\_INT4 and CDF\_UINT4 rather than int C variables.

## 4.6 Data Encodings

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

HOST_ENCODING	Indicates host machine data representation (native). This is the default encoding, and it will provide the greatest performance when reading/writing on a machine of the same type.
NETWORK_ENCODING	Indicates network transportable data representation (XDR).
VAX_ENCODING	Indicates VAX data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.
ALPHAVMSd_ENCODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.

ALPHAVMSg_ENCODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's G_FLOAT representation.
ALPHAVMSi_ENCODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in IEEE representation.
ALPHAOSF1_ENCODING	Indicates DEC Alpha running OSF/1 data representation.
SUN_ENCODING	Indicates SUN data representation.
SGi_ENCODING	Indicates Silicon Graphics Iris and Power Series data representation.
DECSTATION_ENCODING	Indicates DECstation data representation.
IBMRS_ENCODING	Indicates IBMRS data representation (IBM RS6000 series).
HP_ENCODING	Indicates HP data representation (HP 9000 series).
PC_ENCODING	Indicates PC data representation.
NeXT_ENCODING	Indicates NeXT data representation.
MAC_ENCODING	Indicates Macintosh data representation.

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

When inquiring the encoding of a CDF, either NETWORK\_ENCODING or a specific machine encoding will be returned. (HOST\_ENCODING is never returned.)

## 4.7 Data Decodings

A CDF's decoding affects how its attribute entry and variable data values are passed out to a calling application. The decoding for a CDF may be selected and reselected any number of times while the CDF is open. Selecting a decoding does not affect how the values are stored in the CDF file(s) - only how 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. This is the default majority.
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 multi-dimensional 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`

Gnu's "zip" compression.<sup>1</sup> There is one parameter.

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

## 4.11 Sparseness

### 4.11.1 Sparse Records

The following types of sparse records for variables are supported.

`NO_SPARSERECORDS`

No sparse records.

`PAD_SPARSERECORDS`

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

`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.<sup>2</sup>

`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).

---

<sup>1</sup> Disabled for PC running 16-bit DOS/Windows 3.x.

<sup>2</sup> Obviously, sparse arrays are not yet supported.

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

### 4.13 Read-Only Modes

Once a CDF has been opened, it may be placed into a read-only mode to prevent accidental modification (such as when the CDF is simply being browsed). Read-only mode is selected via the Internal Interface using the <SELECT\_CDF\_READONLY\_MODE\_> operation. When read-only mode is set, all metadata is read into memory for future reference. This improves overall metadata access performance but is extra overhead if metadata is not needed. Note that if the CDF is modified while not in read-only mode, subsequently setting read-only mode in the same session will not prevent future modifications to the CDF.

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

### 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 systems being used (including logical names on VMS systems and environment variables on UNIX systems).
CDF_VAR_NAME_LEN256	Maximum length of a variable name.
CDF_ATTR_NAME_LEN256	Maximum length of an attribute name.
CDF_COPYRIGHT_LEN	Maximum length of the CDF copyright text.
CDF_STATUSTEXT_LEN	Maximum length of the explanation text for a status code.

## 4.18 Backward File Compatibility with CDF 2.7

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

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

The following example uses the Internal Interface routine to create two CDF files: "MY\_TEST1.cdf" is a V3.1 file while "MY\_TEST2.cdf" a V2.7 file. Alternatively, the Standard Interface routine CDF\_create\_CDF can be used for the file creation.

```
.
.
include 'cdf.inc'
```

```

integer*4    id1, id2          /* CDF identifier. */
integer*4    status           /* Returned status code. */
integer*4    numDims = 0      /* Number of dimensions. */
integer*4    dimSizes[1] = {0} /* Dimension sizes. */
.
.
status = CDF_lib (CREATE_, CDF_, "MY_TEST1", numDims, dimSizes, id1,
1
NULL_, status)
if (status .lt. CDF_OK) call UserStatusHandler (status)
.
.
call CDF_set_FileBackward(BACKWARDFILEon)
status = CDF_lib (CREATE_, CDF_, "MY_TEST2", numDims, dimSizes, id2,
1
NULL_, status)
if (status .NE. CDF_OK) call UserStatusHandler (status)
.
.

```

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

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

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

```

include 'cdf.inc'
.
.
integer*4    flag             /* CDF identifier. */
.
flag = CDF_get_FileBackward()

```

## 4.19 Checksum

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

If the checksum bit is turned on, a 16-byte signature message (a.k.a. message digest) is computed from the entire file and appended to the end of the file when the file is closed (after any create/write/update activities). Every time such

file is open, other than the normal steps for opening a CDF file, this signature, serving as the authentic checksum, is used for file integrity check by comparing it to the re-computed checksum from the current file. If the checksums match, the file's data integrity is verified. Otherwise, an error message is issued. Currently, the valid checksum modes are: **NO\_CHECKSUM** and **MD5\_CHECKSUM**, both defined in `cdf.h`. With **MD5\_CHECKSUM**, the **MD5** algorithm is used for the checksum computation. The checksum operation can be applied to CDF files that were created with V2.6 or later.

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

See Section 6.2.5 and 6.2.23 for the Standards Interface functions and Section 7.6 for the Internal Interface functions. The environment variable method requires no function calls (and thus no code change is involved for existing applications). The environment variable **CDF\_CHECKSUM** on all Unix platforms and Windows, or **CDF\$CHECKSUM** on Open/VMS, is used to control the checksum option. If its value is set to "**MD5**", all new CDF files will have their checksum bit set with a signature message produced by the MD5 algorithm. If the environment variable is not set or its value is set to anything else, no checksum is set for the new files.

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

```
.
.
include 'cdf.inc'
.
.
integer*4    id1, id2                /* CDF identifier. */
integer*4    status                 /* Returned status code. */
integer*4    numDims = 0            /* Number of dimensions. */
integer*4    dimSizes[1] = {0}     /* Dimension sizes. */
integer*4    checksum               /* Number of dimensions. */
.
.
status = CDF_lib (CREATE_, CDF_, "MY_TEST1", numDims, dimSizes, id1,
1              NULL_, status)
if (status .NE. CDF_OK) call UserStatusHandler (status)

checksum = MD5_CHECKSUM
status = CDFlib (SELECT_, CDF_, id1,
1              PUT_, CDF_CHECKSUM_, checksum,
2              NULL_, status)
if (status .NE. CDF_OK) UserStatusHandler (status)
.
.
status = CDFlib (OPEN_, CDF_, "MY_TEST2", id2,
1              NULL_);
if (status .NE. CDF_OK) UserStatusHandler (status)
.
.
checksum = NO_CHECKSUM
status = CDFlib (SELECT_, CDF_, id2,
1              PUT_, CDF_CHECKSUM_, checksum,
2              NULL_, status)
if (status .NE. CDF_OK) UserStatusHandler (status)
```

Alternatively, the Standard Interface function **CDF\_set\_Checksum** can be used for the same purpose.

The following example uses the Internal Interface to get the checksum mode used in a CDF.

```
.  
.
#include "cdf.h"
.
.
CDFid      id;                /* CDF identifier. */
CDFstatus  status;           /* Returned status code. */
long       checksum;         /* Checksum code. */
.
.
status = CDFlib (OPEN_, CDF_, "MY_TEST1", &id,
                NULL_);
if (status .NE. CDF_OK) UserStatusHandler (status);
.
.
status = CDFlib (SELECT_, CDF_, id,
                GET_, CDF_CHECKSUM_, &checksum,
                NULL_);
if (status .NE. CDF_OK) UserStatusHandler (status)
if (checksum .EQ. MD5_CHECKSUM) THEN
.....
ENDIF
.
.
```

Alternatively, the Standard Interface function **CDF\_get\_Checksum** can be used for the same purpose.



# Chapter 5

## 5 Standard Interface (Original)

The following sections describe the original Standard Interface routines callable from Fortran applications. Most functions return a status code of type INTEGER\*4 (see Chapter 8). The Internal Interface is described in Chapter 7. An application can use both interfaces when necessary.

These routines have been available since earlier CDF versions. Very limited access to zVariables is available here and there is no access to entries associated with zVariable. While they are still supported in the V3.\* library, a new set of Standard Interface routines is made available to complement this limited list. Those routines are described in the Chapter 6.

### 5.1 CDF\_attr\_create

```
SUBROUTINE CDF_attr_create (
```

```
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_LEN256 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 8 explains how to interpret status codes.

### 5.1.1 Example(s)

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

```

.
.
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)
.
.

```

## 5.2 CDF\_attr\_entry\_inquire

SUBROUTINE CDF\_attr\_entry\_inquire (

```

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 5.4). 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 5.5).
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 rEntry).
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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

## 5.2.1 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 has variable scope, the entry 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_LEN256) ! 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

```

## 5.3 CDF\_attr\_get

SUBROUTINE CDF\_attr\_get (

```

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 5.5).
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 function 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 memory at address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 5.3.1 Example(s)

The following example displays the value of the UNITS attribute for the rEntry corresponding to the PRES\_LVL rVariable (but only if the data type is CDF\_CHAR).

```

.
.
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
.
.

```

## 5.4 CDF\_attr\_inquire

SUBROUTINE CDF\_attr\_inquire (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,     ! in -- Attribute number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN256), ! 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 (Section 5.2).

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 5.5).
attr_name	The attribute's name. This character string must be large enough to hold CDF_ATTR_NAME_LEN256 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 7). If no entries exist for the attribute, then a value of zero (0) will be passed back.
status	The completion status code. Chapter 8 explains how to interpret status codes.

## 5.4.1 Example(s)

The following example displays the name of each attribute in a CDF. The number of attributes in the CDF is first determined using the function 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_LEN256)! 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
.
.
```

## 5.5 CDF\_attr\_num

```
INTEGER*4 FUNCTION CDF_attr_num (
```

```

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 (of type INTEGER\*4) 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_LEN256 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.)

### 5.5.1 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. 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 5.7.

```

.
.
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)

```

## 5.6 CDF\_attr\_put

SUBROUTINE CDF\_attr\_put (

```

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 this entry.
INTEGER*4 num_elements, ! in -- Number of elements (of the data type).
<type>    value,       ! out -- Value (<type> is dependent 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 5.5).
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 specified 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
value	The value(s) to write. The entry value is written to the CDF from memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.



num\_elements elements of the data type data\_type will be written to the CDF starting from memory address value.

### 5.6.1 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 the variable scope attribute 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)
.
.
```

### 5.7 CDF\_attr\_rename

SUBROUTINE CDF\_attr\_rename (

```
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 5.5).
attr_name	The new attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.
status	The completion status code. Chapter 8 explains how to interpret status codes.

## 5.7.1 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)
.
.
```

## 5.8 CDF\_close

```
SUBROUTINE CDF_close (
```

```
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 8 explains how to interpret status codes.

## 5.8.1            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)
.
.
```

## 5.9            CDF\_create

```
SUBROUTINE CDF_create (
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). For 0-dimensional rVariables this argument is ignored (but must be present).
encoding	The encoding for variable data and attribute entry data. Specify one of the encodings described in Section 4.6.
majority	The majority for variable data. Specify one of the majorities described in Section 4.8.
id	The identifier for the created CDF. This identifier must be used in all subsequent operations on the CDF.
status	The completion status code. Chapter 8 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 5.8).

## 5.9.1 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.

## 5.10 CDF\_delete

SUBROUTINE CDF\_delete (

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

The arguments to 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 8 explains how to interpret status codes.

### 5.10.1 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
.  
.
```

## 5.11 CDF\_doc

```
SUBROUTINE CDF_doc (
```

```
INTEGER*4    id,                ! in -- CDF identifier.
INTEGER*4    version,          ! out -- Version number.
INTEGER*4    release,          ! out -- Release number.
CHARACTER    copy_right*(CDF_COPYRIGHT_LEN), ! out -- Copyright.
INTEGER*4    status)           ! out -- Completion status
```

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

The arguments to 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.
copy_right	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 newline character after each line of the copyright notice.
status	The completion status code. Chapter 8 explains how to interpret status codes.

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

### 5.11.1 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
                             ! ina 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_CHARACTER= CDF_COPYRIGHT_LEN
    DO WHILE (copyright(last_char:last_char) .EQ. ' ')
        last_CHARACTER= last_CHARACTER- 1
    END DO
    lf = CHAR(10)
    start_CHARACTER= 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_CHARACTER= i + 1
        END IF
    END DO
END IF
.
.

```

## 5.12 CDF\_error

SUBROUTINE CDF\_error (

```

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 8 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.

### 5.12.1 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_wet1', id, status)
IF (status .LT. CDF_WARN) THEN ! INFO and WARNING codes ignored.
  CALL CDF_error (status, text)
  last_CHARACTER= CDF_STATUSTEXT_LEN
  DO WHILE (text(last_char:last_char) .EQ. ' ')
    last_CHARACTER= last_CHARACTER- 1
  END DO
  WRITE (6,101) text(1:last_char)
101  FORMAT (' ', 'ERROR> ', A)
END IF
.
.

```

## 5.13 CDF\_getrvarsrecorddata

SUBROUTINE CDF\_getrvarsrecorddata(

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 num_var            ! in -- Number of rVariables.
INTEGER*4 var_nums(*)       ! in -- rVariable numbers.
INTEGER*4 rec_num           ! in -- Record number.
<type> buffer                ! out -- First variable buffer in a common block (<type> depends
                           !         on the data type of the rVariable).
INTEGER*4 status            ! out -- Completion status.

```

CDF\_getrvarsrecorddata is used to read a full record data at a specific record number for a selected group of rVariables in a CDF. It expects that the data buffer for each rVariable is big enough to hold a full physical record<sup>1</sup> data and properly put in a common block. No space is needed for each rVariable's non-variant dimensional elements. Retrieved record data from the variable group is filled into respective rVariable's buffer.

The arguments to CDF\_getrvarsrecorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDF_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the rVariables in the group involved this read operation.
var_nums	The numbers of the rVariables involved for which to read a whole record data.
rec_num	The record number at which to read the whole record data for the group of rVariables.
buffer	The first variable buffer to read in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.

<sup>1</sup> Physical record is explained in the Primer chapter in the CDF User's Guide.



### 5.13.1 Example(s)

The following example will read an entire single record data for a group of rVariables. The CDF's rVariables are 2-dimensional with sizes [2,2]. The rVariables involved in the read are **Time**, **Longitude**, **Latitude** and **Temperature**. The record to read is 5. Since the dimension variances for **Time** are [NONVARY,NONVARY], a scalar variable of INTEGER\*4 is allocated for its data type CDF\_INT4. For **Longitude**, a 1-dimensional array of REAL\*4 is allocated as its dimension variances are [VARY,NONVARY] with data type CDF\_REAL4. A similar allocation is done for **Latitude** for its [NONVARY,VARY] dimension variances and CDF\_REAL4 data type. For **Temperature**, a 2-dimensional array of REAL\*4 is allocated due to its [VARY,VARY] dimension variances and CDF\_REAL4 data type.

```
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4    id                ! CDF identifier.
INTEGER*4    status            ! Returned status code.
INTEGER*4    num_var           ! Number of rVariables.
INTEGER*4    var_nums(4)      ! rVariable numbers in CDF.
INTEGER*4    rec_num          ! Record number to read.
INTEGER*4    time              ! Datatype: INT4.
                                ! Rec/dim variances: T/FF.
REAL*4       longitude(2)     ! Datatype: REAL4.
                                ! Rec/dim variances: T/TF.
REAL*4       latitude(2)      ! Datatype: REAL4.
                                ! Rec/dim variances: T/FT.
REAL*4       temperature(2,2) ! Datatype: REAL4.
                                ! Rec/dim variances: T/TT.
COMMON /BLK/time, longitude, latitude, temperature
.
.
num_var = 4                ! Number of rVariables
rec_num = 5                ! Record number to read
var_nums(1) = CDF_var_num (id, 'Time') ! rVariable number
IF (var_nums(1) .LT. 1)    ! If less than one (1),
1   CALL UserStatusHandler (var_nums(1)) ! then it is actually a
                                ! warning/error code.
var_nums(2) = CDF_var_num (id, 'Longitude')
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

var_nums(3) = CDF_var_num (id, 'Latitude')
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

var_nums(4) = CDF_var_num (id, 'Temperature')
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

CALL CDF_getrvarsrecorddata (id, num_var, var_nums, rec_num,
1   time, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <GET\_rVARs\_RECDDATA\_>.

## 5.14 CDF\_getzvarsrecddata

SUBROUTINE CDF\_getzvarsrecddata(

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 num_var      ! in -- Number of zVariables.
INTEGER*4 var_nums(*)  ! in -- zVariable numbers.
INTEGER*4 rec_num      ! in -- Record number.
<type> buffer          ! out -- First variable buffer in a common block (<type> depends
                       !         on the data type of the zVariable).
INTEGER*4 status       ! out -- Completion status.

```

CDF\_getzvarsrecddata is used to read a full record data at a specific record number for a selected group of zVariables in a CDF. It expects that the data buffer for each zVariable is big enough to hold a full physical record<sup>2</sup> data and properly put in a common block. No space is needed for each zVariable's non-variant dimensional elements. Retrieved record data from the variable group is filled into respective zVariable's buffer.

The arguments to CDF\_getzvarsrecddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDF_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the zVariables in the group involved this read operation.
var_nums	The numbers of the zVariables involved for which to read a whole record data.
rec_num	The record number at which to read the whole record data for the group of zVariables.
buffer	The first variable buffer to read in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.

### 5.14.1 Example(s)

The following example will read an entire single record data for a group of zVariables. The zVariables involved in the read are **Time**, **Longitude**, **Delta**, **Temperature** and **NAME**. The record to read is 4. Since **Temperature** is 0-dimensional with CDF\_FLOAT data type, a scalar variable of REAL\*4 is allocated. For **Longitude**, a 1-dimensional array of INTEGER\*2 (size [3]) is given for its dimension variance [VARY] and data type CDF\_INT2. Similar data variables are provided for **Longitude** and **Time**. They both are 2-dimensional array of INTEGER\*4 (sizes [3,2]) for their dimension variances [VARY,VARY] and data type either CDF\_INT4 or CDF\_UINT4. For **NAME**, a 1-dimensional array of CHARACTER\*10 (size [2]) is allocated due to its [VARY] dimension variance and CDF\_CHAR data type with the number of element 10.

<sup>2</sup> Physical record is explained in the Primer chapter in the CDF User's Guide.

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4    id            ! CDF identifier.
INTEGER*4    status       ! Returned status code.
INTEGER*4    num_var      ! Number of zVariables.
INTEGER*4    var_nums(5)  ! zVariable numbers in CDF.
INTEGER*4    rec_num      ! Record number to write.
INTEGER*4    time(3,2)    ! Datatype: UINT4.
                ! Rec/dim variances: T/TT.
INTEGER*4    delta(3,2)   ! Datatype: INT4 .
                ! Rec/dim variances: T/TT.
INTEGER*2    longitude(3) ! Datatype: INT2.
                ! Rec/dim variances: T/T.
REAL*4       temperature  ! Datatype: FLOAT.
                ! Rec/dim variances: T/.
CHARACTER*10 name(2)      ! Datatype: CHAR/10.
                ! Rec/dim variances: T/T.
COMMON /BLK/delta, time, temperature, longitude, name
.
.
num_var = 5            ! Number of zVariables
rec_num = 4            ! Record number to read

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Delta', var_nums(1),
1             NULL_, status)           ! zVariable number
IF (var_nums(1) .LT. 1)                ! If less than one (1),
x CALL UserStatusHandler (var_nums(1)) ! then it is actually a
                ! warning/error code.

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Time', var_nums(2),
1             NULL_, status)
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Longitude', var_nums(3),
1             NULL_, status)
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Temperature', var_nums(4),
1             NULL_, status)
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'NAME', var_nums(5),
1             NULL_, status)
IF (var_nums(5) .LT. 1) CALL UserStatusHandler (var_nums(5))

CALL CDF_getzvarsrecorddata (id, num_var, var_nums, rec_num,
1             delta, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if

such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <GET\_,zVARs\_RECADATA\_>.

## 5.15 CDF\_inquire

SUBROUTINE CDF\_inquire(

```
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 for the rVariables in the CDF.
dim_sizes	The dimension sizes of the rVariables in the CDF. dim_sizes is a 1-dimensional array containing one element per dimension. Each element of dim_sizes receives the corresponding dimension size. For 0-dimensional rVariables this argument is ignored (but must be present).
encoding	The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.
majority	The majority of the variable data. The majorities are defined in Section 4.8.
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 7).
num_vars	The number of rVariables in the CDF.
num_attrs	The number of attributes in the CDF.
status	The completion status code. Chapter 8 explains how to interpret status codes.

## 5.15.1 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)
.
.
```

## 5.16 CDF\_open

SUBROUTINE CDF\_open (

```
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 cannot 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 8 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 5.8).

### 5.16.1 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)
.
.

```

## 5.17 CDF\_putrvarsrecorddata

SUBROUTINE CDF\_putrvarsrecorddata(

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 num_var            ! in -- Number of rVariables.
INTEGER*4 var_nums(*)       ! in -- rVariable numbers.
INTEGER*4 rec_num           ! in -- Record number.
<type> buffer                ! in -- First variable buffer in a common block (<type> depends
                             !      on the data type of the rVariable).
INTEGER*4 status            ! out -- Completion status.

```

CDF\_putrvarsrecorddata is used to write a full record data at a specific record number for a selected group of rVariables in a CDF. It expects that the data buffer for each zVariable is big enough to contain a full physical record data and properly put in a common block. No space is expected for each rVariable's non-variant dimensional elements. Record data from each buffer is written to its respective rVariable.

The arguments to CDF\_putrvarsrecorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, CDF_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the rVariables in the group involved this write operation.
var_nums	The numbers of the rVariables involved for which to write a whole record data.
rec_num	The record number at which to write the whole record data for the group of rVariables.
buffer	The first variable buffer to write in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.

### 5.17.1 Example(s)

The following example will write an entire single record data for a group of rVariables. The CDF's rVariables are 2-dimensional with sizes [2,2]. The rVariables involved in the write are **Time**, **Longitude**, **Latitude** and **Temperature**. The record to write is 5. Since the dimension variances for **Time** are [NONVARY, NONVARY], a scalar variable of INTEGER\*4 is allocated for its data type CDF\_INT4. For **Longitude**, a 1-dimensional array of REAL\*4 is allocated as its dimension variances are [VARY, NONVARY] with data type CDF\_REAL4. A similar allocation is done for **Latitude** for its [NONVARY, VARY] dimension variances and CDF\_REAL4 data type. For **Temperature**, a 2-dimensional array of REAL\*4 is allocated due to its [VARY, VARY] dimension variances and CDF\_REAL4 data type.

```

INCLUDE '<path>cdf.inc'
.
INTEGER*4    id                ! CDF identifier.
INTEGER*4    status            ! Returned status code.
INTEGER*4    num_var           ! Number of rVariables.
INTEGER*4    var_nums(4)      ! rVariable numbers in CDF.
INTEGER*4    rec_num          ! Record number to write.
INTEGER*4    time /123/      ! Datatype: INT4.
                                ! Rec/dim variances: T/FF.
REAL*4       longitude(2)     ! Datatype: REAL4.
1            /100.01, -100.02/ ! Rec/dim variances: T/TF.
REAL*4       latitude(2)      ! Datatype: REAL4.
1            /23.45, -54.32/  ! Rec/dim variances: T/FT.
REAL*4       temperature(2,2) ! Datatype: REAL4.
1            /20.0, 40.0,     ! Rec/dim variances: T/TT.
2            30.0, 50.0/
COMMON /BLK/time, longitude, latitude, temperature
.
num_var = 4                    ! Number of rVariables
rec_num = 5                    ! Record number to write
var_nums(1) = CDF_var_num (id, 'Time') ! rVariable number
IF (var_nums(1) .LT. 1)        ! If less than one (1),
1    CALL UserStatusHandler (var_nums(1)) ! then it is actually a
                                ! warning/error code.
var_nums(2) = CDF_var_num (id, 'Longitude')
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

```

```

var_nums(3) = CDF_var_num (id, 'Latitude')
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

var_nums(4) = CDF_var_num (id, 'Temperature')
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

CALL CDF_putrvarsrecorddata (id, num_var, var_nums, rec_num,
1                             time, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <PUT\_rVARs\_RECDDATA\_>.

## 5.18 CDF\_putzvarsrecorddata

SUBROUTINE CDF\_putzvarsrecorddata(

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 num_var            ! in -- Number of zVariables.
INTEGER*4 var_nums(*)        ! in -- zVariable numbers.
INTEGER*4 rec_num            ! in -- Record number.
<type> buffer                ! in -- First variable buffer in a common block (<type> depends
                             !      on the data type of the zVariable).
INTEGER*4 status             ! out -- Completion status.

```

CDF\_putzvarsrecorddata is used to write a full record data at a specific record number for a selected group of zVariables in a CDF. It expects that the data buffer for each zVariable is big enough to contain a full physical record data and properly put in a common block. No space is expected for each zVariable's non-variant dimensional elements. Record data from each buffer is written to its respective zVariable.

The arguments to CDF\_putzvarsrecorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, Cdf_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the zVariables in the group involved this write operation.
var_nums	The numbers of the zVariables involved for which to write a whole record data.
rec_num	The record number at which to write the whole record data for the group of zVariables.
buffer	The first variable buffer to write in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.



## 5.18.1 Example(s)

The following example will write an entire single record data for a group of zVariables. The zVariables involved in the write are **Time**, **Longitude**, **Delta**, **Temperature** and **NAME**. The record to write is 4. Since **Temperature** is 0-dimensional with **CDF\_FLOAT** data type, a scalar variable of REAL\*4 is allocated. For **Longitude**, a 1-dimensional array of INTEGER\*2 (size [3]) is given for its dimension variance [**VARY**] and data type **CDF\_INT2**. Similar data variables are provided for **Longitude** and **Time**. They both are 2-dimensional array of INTEGER\*4 (sizes [3,2]) for their dimension variances [**VARY,VARY**] and data type either **CDF\_INT4** or **CDF\_UINT4**. For **NAME**, a 1-dimensional array of CHARACTER\*10 (size [2]) is allocated due to its [**VARY**] dimension variance and **CDF\_CHAR** data type with the number of element 10.

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4    id                ! CDF identifier.
INTEGER*4    status            ! Returned status code.
INTEGER*4    num_var           ! Number of zVariables.
INTEGER*4    var_nums(5)       ! zVariable numbers in CDF.
INTEGER*4    rec_num           ! Record number to write.
INTEGER*4    time(3,2)        ! Datatype: UINT4.
1            /10, 20,          ! Rec/dim variances: T/TT.
2            30, 40,
3            50, 60/
INTEGER*4    delta(3,2)       ! Datatype: INT4 .
1            /1, 2,            ! Rec/dim variances: T/TT.
2            5, 6,
3            9, 10/
INTEGER*2    longitude(3)     ! Datatype: INT2.
1            /10, 20, 30/      ! Rec/dim variances: T/T.
REAL*4       temperature      ! Datatype: FLOAT.
1            /1234.56/         ! Rec/dim variances: T/.
CHARACTER*10 name(2)          ! Datatype: CHAR/10.
1            /'ABCDEFGHIJ',    ! Rec/dim variances: T/T.
2            '12345678'/

COMMON /BLK/delta, time, temperature, longitude, name
.
.
num_var = 5                    ! Number of zVariables
rec_num = 4                    ! Record number to write

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Delta', var_nums(1),
1            NULL_, status)    ! zVariable number
IF (var_nums(1) .LT. 1)       ! If less than one (1),
x CALL UserStatusHandler (var_nums(1)) ! then it is actually a
! warning/error code.

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Time', var_nums(2),
1            NULL_, status)
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Longitude', var_nums(3),
1            NULL_, status)
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

```

```

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Temperature', var_nums(4),
1          NULL_, status)
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'NAME', var_nums(5),
1          NULL_, status)
IF (var_nums(5) .LT. 1) CALL UserStatusHandler (var_nums(5))

CALL CDF_putzvarsrecorndata (id, num_var, var_nums, rec_num,
1          delta, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <PUT\_, zVARs\_RECADATA\_>.

## 5.19 CDF\_var\_close

```

SUBROUTINE CDF_var_close (
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 5.25).
status	The completion status code. Chapter 8 explains how to interpret status codes.

## 5.19.1 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)
.
.
```

## 5.20 CDF\_var\_create

```
SUBROUTINE CDF_var_create (
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.
```

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_LEN256 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 rVariable's record variance. Specify one of the variances defined in Section 4.9.
dim_variances	The rVariable'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 8 explains how to interpret status codes.

## 5.20.1 Example(s)

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

```
.
.
INCLUDE '<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)
.
.

```

## 5.21 CDF\_var\_get

SUBROUTINE CDF\_var\_get (

```

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> is dependent 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 (see Section 5.22).

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 5.25).
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 at memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 5.21.1 Example(s)

The following example will read and hold an entire record of data from an rVariable. The CDF's rVariables are 3-dimensional with sizes [180,91,10]. For this rVariable the record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF\_REAL4.

```

.
.
INCLUDE '<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
END DO
.
.

```

## 5.22 CDF\_var\_hyper\_get

SUBROUTINE CDF\_var\_hyper\_get (

```

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,             ! in -- Buffer of values (<type> is dependent 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\_inquire can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to 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 5.25).
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. If there are zero (0) dimensions, this argument is ignored (but must be present).
counts	The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. 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 to hold the values. 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 values are read from the CDF and placed into memory starting at address buffer.  <b>WARNING:</b> 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 8 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 CDF's variable majority is ROW\_MAJOR. For the rVariable the record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF\_REAL4. This example is similar to the example in Section 5.21 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\_hyper\_put

```
SUBROUTINE CDF_var_hyper_put (
```

```

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> is dependent 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 can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to 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 5.25).
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 <sup>3</sup> (e.g., An interval of 2 means write to every other record).
indices	The indices (within each record) at which to start writing. Each element of indices specifies the corresponding dimension index. If there are zero (0) dimensions, this argument is ignored (but must be present).
counts	The number of values along each dimension to write. Each element of count specifies the corresponding dimension count. For 0-dimensional rVariables this argument is ignored (but must be present).
intervals	For each dimension the interval between values for subsampling <sup>4</sup> (e.g., an interval of 2 means write to every other value). intervals is a 1-dimensional array containing one element per rVariable dimension. Each element of intervals specifies the corresponding dimension interval. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).
buffer	The buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address buffer are written to the CDF.  <b>WARNING:</b> 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 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 8 explains how to interpret status codes.

### 5.23.1 Example(s)

The following example writes values to the rVariable LATITUDE of a CDF whose rVariables are 2-dimensional with dimension sizes [360,181]. For LATITUDE the record variance is NOVARY, the dimension variances are [NOVARY,VARY], and the data type is CDF\_INT2. This example is similar to the example in Section 5.26 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.

```

<sup>3</sup> "Subsampling" is not the best term to use when writing data, but you should know what we mean.

<sup>4</sup> Again, not the best term.

```

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)
.
.

```

## 5.24 CDF\_var\_inquire

SUBROUTINE CDF\_var\_inquire (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- rVariable number.
CHARACTER var_name*(CDF_VAR_NAME_LEN256), ! 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 5.25).
var_name	The rVariable's name. This character string must be large enough to hold CDF_VAR_NAME_LEN256 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 rVariable value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. (Each value consists of the entire string.) For all other data types, this will always be one (1) - multiple elements at each value are not allowed for non-character data types.
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 8 explains how to interpret status codes.

### 5.24.1 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_LEN256) ! 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)
.
.

```

## 5.25 CDF\_var\_num

```

INTEGER*4 FUNCTION CDF_var_num (

```

```

INTEGER*4 id,           ! in-- CDF identifier.
CHARACTER var_name*(*); ! in-- Variable name.

```

CDF\_var\_num is used to determine the number associated with a given rVariable or zVariable name. If the variable 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 (of type INTEGER\*4) 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.
VarName	The name of the variable, either rVariable or zVariable, for which to search. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.

CDF\_var\_num may be used as an embedded function call when a variable number is needed. CDF\_var\_num is declared in cdf.inc. (Fortran functions must be declared so that the returned value is interpreted correctly.)

## 5.25.1 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_LEN256) ! 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 5.24.

## 5.26 CDF\_var\_put

SUBROUTINE CDF\_var\_put (

```
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> is dependent 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 (see Section 5.23).

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 5.25).
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 memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 5.26.1 Example(s)

The following example writes values to the rVariable named LATITUDE in a CDF whose rVariables are 2-dimensional with dimension sizes [360,181]. For LATITUDE the record variance is NOVARY, the dimension variances are [NOVARY,VARY], and the data type is CDF\_INT2.

```
.
.
INCLUDE '<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.)

## 5.27 CDF\_var\_rename

```

SUBROUTINE CDF_var_rename (

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 5.25).
var_name	The new rVariable name. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 5.27.1 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
.
.
```





# Chapter 6

## 6 Standard Interface (New)

The following sections describe the new set of Standard Interface routines callable from Fortran applications. Most subroutines return a status code of type INTEGER\*4 (see Chapter 8). The Internal Interface is described in Chapter 7. An application can use either or both interfaces when necessary.

Previously, the Standard Interface only provided a very limited functionality within the CDF library. For example, it could not handle zVariables and vAttribute zEntries (they were only accessible via the Internal Interface). Since V3.1, the Standard Interface has been expanded to include many new operations that are previously only available through the Internal Interface.

The original Standard Interface functions<sup>1</sup> and subroutines<sup>2</sup>, described in Chapter 5, in the previous library version are still available and work the same way as before. To encourage the use of zVariables, the preferred variable type over the rVariables in the CDF, new subroutines are explicitly added to the library to handle zVariables, their data as well as entries in the variable-scoped attributes. The original Standard Interface functions/subroutines can be used to operate the rVariables and their associated rEntries. The Internal Interface needs to be called to operate the functions/items that are not available from the new Standard Interface.

A naming convention is adopted by the new Standard Interface subroutines to separate the operations on zVariable, as well as entry, i.e., gEntry, rEntry and zEntry.

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

### 6.1 Library

The functions in this section are related to the library being used for the CDF operations and are common for any CDF entity, i.e., CDFs, variables, attributes and entries.

---

<sup>1</sup> They are: CDF\_attr\_Num and CDF\_var\_Num.

<sup>2</sup> They are: CDF\_create, CDF\_open, CDF\_doc, CDF\_inquire, CDF\_close, CDF\_delete, CDF\_attr\_Create, CDF\_attr\_Rename, CDF\_attr\_Inquire, CDF\_attr\_Entry\_Inquire, CDF\_attr\_Put, CDF\_attr\_Get, CDF\_var\_Create, CDF\_var\_Rename, CDF\_var\_Inquire, CDF\_var\_Put, CDF\_var\_Get, CDF\_var\_Hyper\_Put, CDF\_var\_Hyper\_Get, CDF\_var\_Close, CDF\_getrVarsRecordData, CDF\_getzVarsRecordData, CDF\_putrVarsRecordData and CDF\_putzVarsRecordData.

## 6.1.1 CDF\_get\_datatype\_size

SUBROUTINE CDF\_get\_datatype\_size (

```
INTEGER*4 data_type,      ! in -- CDF data type.
INTEGER*4 size,          ! out -- Size in bytes.
INTEGER*4 status)       ! out -- Completion status
```

CDF\_get\_datatype\_size acquires the size (in bytes) of an element of the specified CDF data type

The arguments to CDF\_get\_datatype\_size are defined as follows:

data\_type    The CDF data type.

size         Size in bytes of that data type.

status       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.1.1.1. Example(s)

The following example acquires the size (in bytes) of CDF data type CDF\_INT4 (it should be 4 bytes).

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 size           ! Size of the data type.
INTEGER*4 status        ! Returned status code.
.
.
CALL CDF_get_datatype_size (CDF_INT4, size, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.1.2 CDF\_get\_lib\_copyright

SUBROUTINE CDF\_get\_lib\_copyright (

```
CHARACTER copyright*(*), ! out -- CDF library copyright notice.
INTEGER*4 status)       ! out -- Completion status
```

CDF\_get\_lib\_copyright acquires the copyright notice of the CDF library being used.

The arguments to CDF\_get\_lib\_copyright are defined as follows:

copyright    The copyright notice from the CDF library.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.1.2.1. Example(s)

The following example acquires the CDF library's copyright notice.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
CHARACTER copyright*(CDF_COPYRIGHT_LEN)        ! Copyright notice.
INTEGER*4 status                                ! Returned status code.

.
.
CALL CDF_get_lib_copyright (copyright, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.1.3 CDF\_get\_lib\_version

```
SUBROUTINE CDF_get_lib_version (
INTEGER*4 version,                ! out -- CDF library version.
INTEGER*4 release,                ! out -- CDF library release.
INTEGER*4 increment,              ! out -- CDF library increment.
CHARACTER sub_increment*(*) ! out -- CDF library sub-increment..
INTEGER*4 status)                ! out -- Completion status.
```

CDF\_get\_lib\_version acquires the version and release information from the CDF library being used.

The arguments to CDF\_get\_lib\_version are defined as follows:

version        The CDF library version.

release        The CDF library release.

increment      The CDF library increment.

sub\_increment        The CDF library sub-increment.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.1.3.1. Example(s)

The following example acquires the CDF library's version/release information.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 version           ! Library version.
INTEGER*4 release           ! Library release.
INTEGER*4 increment         ! Library increment.
CHARACTER sub_increment*1   ! Library sub-increment.
INTEGER*4 status            ! Returned status code.

.
.
CALL CDF_get_lib_version (version, release, increment,
1 sub_increment, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.1.4 CDF\_get\_status\_text

```
SUBROUTINE CDF_get_status_text (
INTEGER*4 status_id,          ! in -- CDF status identifier.
CHARACTER text*(*),          ! out -- Status text description.
INTEGER*4 status)            ! out -- Completion status
```

CDF\_get\_status\_text is used to inquire the explanation of a given status code (not just error codes). Chapter 8 explains how to interpret status codes and Appendix A lists all of the possible status codes.

The arguments to CDF\_get\_status\_text are defined as follows:

status_id	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.
status	The status of checking.

### 6.1.4.1. Example(s)

The following example displays the explanation text if an error code is returned from a call to CDF\_open\_cdf.

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

```

INTEGER*4 id                               ! CDF identifier.
INTEGER*4 status1, status2                 ! 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_cdf ('giss_wet1', id, status1)
IF (status1 .LT. CDF_WARN) THEN           ! INFO and WARNING codes ignored.
  CALL CDF_get_status_text (status1, text, status2)
  last_CHARACTER= CDF_STATUSTEXT_LEN
  DO WHILE (text(last_char:last_char) .EQ. ' ')
    last_CHARACTER= last_CHARACTER- 1
  END DO
  WRITE (6,101) text(1:last_char)
101  FORMAT (' ', 'ERROR> ', A)
END IF
.
.

```

## 6.2 CDF

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

### 6.2.1 CDF\_close\_cdf

```

SUBROUTINE CDF_close_cdf(
.
.
.
INTEGER*4 id,                               ! in -- CDF identifier.
INTEGER*4 status)                           ! out -- Completion status

```

CDF\_close\_cdf 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. This routine is identical to the original Standard Interface routine CDF\_close.

**NOTE:** You must close a CDF with CDF\_close\_cdf 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\_cdf, the CDF's cache buffers are left unflushed.

The arguments to CDF\_close\_cdf are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.1.1. 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_cdf (id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.2.2 CDF\_create\_cdf

```
SUBROUTINE CDF_create_cdf (
CHARACTER  CDF_name*(*),      ! in -- CDF file name.
INTEGER*4  status)           ! out -- Completion status
```

CDF\_create\_cdf creates a CDF as defined by the arguments. This function provides the simplest form of CDF creation without the number of dimensions, dimension sizes, encoding and majority arguments required in the original Standard Interface routine, CDF\_create, or the similar process from the Internal Interface CDF\_lib routine. The created CDF will have zero (0) dimension (thus no dimension sizes) and use the default encoding (HOST\_ENCODING) and majority (ROW\_MAJOR), specified in the configuration file of your CDF distribution. This routine should be used to create CDFs that will have only zVariables, or rVariables with no dimensionality. Use CDF\_create or CDF\_lib routine to create CDFs to hold rVariables with dimensions. 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\_cdf, delete it with CDF\_delete, and then recreate it with CDF\_create\_cdf. If the existing CDF is corrupted, the call to CDF\_open\_cdf 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\_cdf 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). |
|          | <b>UNIX:</b> File names are case-sensitive.                                                                                                                                                                                                                                                                                              |
| 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 8 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_cdf` must be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 6.2.1).

### 6.2.2.1. Example(s)

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

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
.
.
CALL CDF_create_cdf ('test1', id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.2.3 CDF\_delete\_cdf

```
SUBROUTINE CDF_delete_cdf(
```

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

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

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

The arguments to `CDF_delete_cdf` are defined as follows:

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

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.3.1. 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_cdf ('test2', id, status)
IF (status .LT. CDF_OK) THEN            ! INFO status codes ignored.
    CALL UserStatusHandler (status)
ELSE
    CALL CDF_delete_cdf (id, status)
    IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

## 6.2.4 CDF\_get\_cachesize

```
SUBROUTINE CDF_get_cachesize (
INTEGER*4 id,                            ! in -- CDF identifier.
INTEGER*4 num_buffers,                 ! out -- Number of cache buffers.
INTEGER*4 status)                       ! out -- Completion status
```

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

The arguments to CDF\_get\_cachesize are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

num\_buffers        The number of cache buffers.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.4.1. Example(s)



The following example acquires the number of cache buffers used for a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 num_buffers       ! Number of cache buffers.
INTEGER*4 status            ! Returned status code.

.
.
CALL CDF_get_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.5 CDF\_get\_checksum

```
SUBROUTINE CDF_get_checksum (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 checksum,         ! out -- Checksum mode.
INTEGER*4 status)           ! out -- Completion status
```

CDF\_get\_checksum acquires the checksum mode of a CDF file. Refer to Section 4.19 for the description of checksum.

The arguments to CDF\_get\_checksum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
checksum	The checksum mode.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.5.1. Example(s)

The following example acquires the checksum mode for a CDF.

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

```

.
.
CALL CDF_get_checksum(id, checksum, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.6 CDF\_get\_compress\_cachesize

SUBROUTINE CDF\_get\_compress\_cachesize (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 num_buffers, ! out -- Number of cache buffers.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_compress\_cachesize acquires the number of cache buffers used for the compression scratch CDF file. Refer to the CDF User's Guide for the description of caching scheme used by the CDF library.

The arguments to CDF\_get\_compress\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
num_buffers	The number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.6.1. Example(s)

The following example acquires the number of cache buffers used for the compression scratch CDF file.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 num_buffers ! Number of cache buffers.
INTEGER*4 status      ! Returned status code.
.
.
CALL CDF_get_compress_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.7 CDF\_get\_compression

SUBROUTINE CDF\_get\_compression (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 compress_type, ! out -- Compression type.
INTEGER*4 compress_parms(*), ! out -- Compression parameters.
INTEGER*4 compress_percent, ! out -- Compression percentage.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_compression acquires the compression information of the CDF. It returns the compression type (method) and, if compressed, the compression parameters and compression rate. CDF compression types/parameters are described in Section 4.10.

The arguments to CDF\_get\_compression are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
compress_type	The compression type.
compress_parms	The compression parameters.
compress_percent	The compression percentage.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.7.1. Example(s)

The following example acquires the compression information from a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 compress_type ! Compression type.
INTEGER*4 compress_parms(CDF_MAX_DIMS) ! Compression parameters.
INTEGER*4 compress_percent ! Compression percentage.
INTEGER*4 status      ! Returned status code.

.
.
CALL CDF_get_compression (id, compress_type, compress_parms,
1                          compress_percent, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.8 CDF\_get\_compression\_info

SUBROUTINE CDF\_get\_compression\_info (

```
char *CDFname,           ! in -- CDF name. */
INTEGER*4 compress_type, ! out -- Compression type.
INTEGER*4 compress_parms(*), ! out -- Compression parameters.
INTEGER*4 compress_percent, ! out -- Compression percentage.
INTEGER*4 status)       ! out -- Completion status
```

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

The arguments to CDFgetCompressionInfo are defined as follows:

CDFname	The pathname of a CDF file without the .cdf file extension.
compress_type	The compression type.
compress_parms	The compression parameters.
compress_percent	The compression percentage.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.8.1. Example(s)

The following example acquires the compression information from a CDF named "MYCDF.cdf".

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 compress_type           ! Compression type.
INTEGER*4 compress_parms(CDF_MAX_DIMS) ! Compression parameters.
INTEGER*4 compress_percent       ! Compression percentage.
INTEGER*4 status                 ! Returned status code.

.
.
CALL CDF_get_compression_info ('MYCDF', id, compress_type, compress_parms,
1 compress_percent, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.9 CDF\_get\_copyright

```

SUBROUTINE CDF_get_copyright (
INTEGER*4 id,           ! in -- CDF identifier.
CHARACTER copyright*(*), ! out -- Copyright notice.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_copyright acquires the copyright notice in a CDF.

The arguments to CDF\_get\_copyright are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
copyright	The copyright notice.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.9.1. Example(s)

The following example acquires the copyright notice from a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
CHARACTER copyright*(CDF_COPYRIGHT_LEN) ! Copyright.
INTEGER*4 status       ! Returned status code.
.
.
CALL CDF_get_copyright (id, copyright, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.10 CDF\_get\_decoding

```

SUBROUTINE CDF_get_decoding (
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 decoding,    ! out -- CDF decoding.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_decoding acquires the decoding for the data in a CDF. The decodings are described in Section 4.7.

The arguments to CDF\_get\_decoding are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

decoding    The decoding.

status       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.10.1. Example(s)

The following example acquires the decoding code for a CDF.

```

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

.
.
CALL CDF_get_decoding (id, decoding, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.11 CDF\_get\_encoding

SUBROUTINE CDF\_get\_encoding (

```

INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 decoding,            ! out -- CDF encoding.
INTEGER*4 status)              ! out -- Completion status

```

CDF\_get\_encoding acquires the encoding code used for the data in a CDF. The encodings are described in Section 4.6.

The arguments to CDF\_get\_encoding are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

encoding    The encoding.

status       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.11.1. Example(s)

The following example acquires the encoding code used in a CDF.

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

.
.
CALL CDF_get_encoding (id, encoding, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.12 CDF\_get\_format

```
SUBROUTINE CDF_get_format (
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 format,      ! out -- CDF format.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_format acquires the file format, single or multi-file, of the CDF. The formats are described in Section 4.4.

The arguments to CDF\_get\_format are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- format       The format.
- status       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.12.1. Example(s)

The following example acquires the file format for a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 format       ! Format.
```

```

INTEGER*4 status          ! Returned status code.

.
.
CALL CDF_get_format (id, format, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.13 CDF\_get\_majority

```

SUBROUTINE CDF_get_majority (

```

```

INTEGER*4 id,             ! in -- CDF identifier.
INTEGER*4 majority,      ! out -- Variable majority.
INTEGER*4 status)        ! out -- Completion status

```

CDF\_get\_majority acquires the variable majority, row or column-major, of the CDF. The majorities are described in Section 4.8.

The arguments to CDF\_get\_majority are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
majority	The variable majority of the CDF.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.13.1. Example(s)

The following example acquires the variable majority of a CDF.

```

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

.
.
CALL CDF_get_majority (id, majority, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```



## 6.2.14 CDF\_get\_name

```
SUBROUTINE CDF_get_name (  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 name,        ! out -- CDF name.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_name acquires the name of the specified CDF.

The arguments to CDF\_get\_name are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
name	The name of the CDF.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.14.1. Example(s)

The following example acquires the name of a CDF.

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

## 6.2.15 CDF\_get\_negtoposfp0\_mode

```
SUBROUTINE CDF_get_negtoposfp0_mode (  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 negtoposfp0, ! out -- -0.0 to 0.0 mode.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_negtoposfp0\_mode acquires -0.0 to 0.0 mode of the CDF. You can use CDF\_set\_negtoposfp0\_mode subroutine to set the mode. The -0.0 to 0.0 modes are described in Section 4.15.

The arguments to CDF\_get\_negtoposfp0\_mode are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- negtoposfp0 The -0.0 to 0.0 mode of the CDF.
- status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.15.1. Example(s)

The following example acquires the -0.0 to 0.0 mode of a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                    ! CDF identifier.
INTEGER*4 negtoposfp0          ! -0.0 to 0.0 mode.
INTEGER*4 status                ! Returned status code.
.
.
CALL CDF_get_negtoposfp0_mode (id, negtoposfp0, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.16 CDF\_get\_readonly\_mode

SUBROUTINE CDF\_get\_readonly\_mode (

```

INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 readonly,             ! out -- Read-only mode of the CDF.
INTEGER*4 status)               ! out -- Completion status

```

CDF\_get\_readonly\_mode acquires the read-only mode for a CDF. You can use CDF\_set\_readonly\_mode to set the mode. The read-only modes are described in Section 4.13.

The arguments to CDF\_get\_readonly\_mode are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- readonly     The read-only mode.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.16.1. Example(s)

The following example acquires the read-only mode of a CDF.

```
.  
.br/>INCLUDE '<path>cdf.inc'  
.br/>.br/>INTEGER*4 id            ! CDF identifier.  
INTEGER*4 readonly     ! Read-only mode.  
INTEGER*4 status       ! Returned status code.  
  
.br/>.br/>CALL CDF_get_readonly_mode (id, readonly, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.br/>.
```

## 6.2.17 CDF\_get\_stage\_cachesize

```
SUBROUTINE CDF_get_stage_cachesize (  
  
INTEGER*4 id,            ! in -- CDF identifier.  
INTEGER*4 num_buffers,   ! out -- Number of cache buffers.  
INTEGER*4 status)       ! out -- Completion status
```

CDF\_get\_stage\_cachesize inquires the number of cache buffers being used for the staging scratch file a CDF. Refer to the CDF User's Guide for the description of the caching scheme used by the CDF library.

The arguments to CDF\_get\_stage\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
num_buffers	Number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.17.1. Example(s)

The following example acquires the number of cache size buffers used for the staging scratch file for a CDF.

```
.  
.
```

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_buffers  ! Number of cache buffers.
.
.
CALL CDF_get_stage_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.18 CDF\_get\_version

```

SUBROUTINE CDF_get_version (

```

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 version,     ! out -- CDF version number.
INTEGER*4 release,     ! out -- CDF release number within the version.
INTEGER*4 increment,   ! out -- CDF increment number within the release.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_version inquires the version/release information for a CDF file. This information reflects the CDF library that was used to create the CDF file.

The arguments to CDF\_get\_version are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
version	CDF version number.
release	CDF release number within the version.
increment	CDF increment number within the release.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.18.1. Example(s)

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

```

.
.
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.
INTEGER*4 increment       ! CDF increment number.
.
.
CALL CDF_get_version (id, version, release, increment, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.19 CDF\_get\_zmode

SUBROUTINE CDF\_get\_zmode (

```

INTEGER*4 id,              ! in -- CDF identifier.
INTEGER*4 zmode,          ! out -- CDF zMode.
INTEGER*4 status)         ! out -- Completion status

```

CDF\_get\_zmode inquires the zMode for a CDF file. The zModes are described in Section 4.14.

The arguments to CDF\_get\_zmode are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
zmode	CDF zMode.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.19.1. Example(s)

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

```

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

```

## 6.2.20 CDF\_inquire\_cdf

SUBROUTINE CDF\_inquire\_cdf (

```
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_rrec,        ! out -- Maximum record number in the CDF, rVariables.
INTEGER*4 num_rvars,      ! out -- Number of rVariables in the CDF.
INTEGER*4 max_zrec,      ! out -- Maximum record number in the CDF, zVariables.
INTEGER*4 num_zvars,     ! out -- Number of zVariables in the CDF.
INTEGER*4 num_attrs,     ! out -- Number of attributes in the CDF.
INTEGER*4 status)       ! out -- Completion status
```

CDF\_inquire\_cdf inquires the basic characteristics of a CDF. This subroutine expands the original Standard Interface subroutine CDF\_inquire by acquiring extra information regarding the zVariables. An application needs to know the number of rVariable dimensions and their sizes before it can access rVariable data. For zVariables, use CDF\_get\_zvar\_numdims and CDF\_get\_zvar\_dimsizes subroutines to acquire each individual zVariable's dimensions and its sizes. 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\_cdf are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
num_dims	The number of dimensions for the rVariables in the CDF.
dim_sizes	The dimension sizes of the rVariables in the CDF. dim_sizes is a 1-dimensional array containing one element per dimension. Each element of dim_sizes receives the corresponding dimension size. For 0-dimensional rVariables this argument is ignored (but must be present).
encoding	The encoding of the variable data and attribute entry data. The encodings are defined in Section 4.6.
majority	The majority of the variable data. The majorities are defined in Section 4.8.
max_rrec	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_rrec is the largest of these. Some rVariables may have fewer records actually written
num_rvars	The number of rVariables in the CDF.
max_zrec	The maximum record number written to a zVariable in the CDF. Note that the maximum record number written is also kept separately for each zVariable in the CDF. The value of max_zrec is the largest of these. Some zVariables may have fewer records actually written. CDF_get_zvar_maxwrittenrecnum (Section 6.3.19) can be used to inquire the maximum record written for an individual zVariable.
num_zvars	The number of zVariables in the CDF.
num_attrs	The number of attributes in the CDF.

status

The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.20.1. Example(s)

The following example inquires 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_rrec         ! Maximum record number among rVariables.
INTEGER*4 num_rvars        ! Number of rVariables in CDF.
INTEGER*4 max_zrec         ! Maximum record number among zVariables.
INTEGER*4 num_zvars        ! Number of zVariables in CDF.
INTEGER*4 num_attrs        ! Number of attributes in CDF.
.
.
CALL CDF_inquire_cdf (id, num_dims, dim_sizes, encoding, majority,
.
.
.
                                max_rrec, num_rvars, max_zrec, num_zvars, num_attrs,
.
                                status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.2.21 CDF\_open\_cdf

SUBROUTINE CDF\_open\_cdf(

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

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

The arguments to CDF\_open\_cdf 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 8 explains how to interpret status codes.

**NOTE:** `CDF_close_cdf` must be used to close the CDF before your application exits to ensure that the CDF will be correctly written to disk (see Section 6.2.1).

### 6.2.21.1. 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 (CDF_name, id, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.22 CDF\_set\_cachesize

```
SUBROUTINE CDF_set_cachesize (
INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 num_buffers,         ! in -- Number of cache buffers.
INTEGER*4 status)               ! out -- Completion status
```

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

The arguments to `CDF_set_cachesize` are defined as follows:

**id**                    The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create_cdf` or `CDF_open_cdf`.



num\_buffers        The number of cache buffers.

status            The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.22.1. Example(s)

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

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 num_buffers                 ! Number of cache buffers.
INTEGER*4 status                       ! Returned status code.

.
.
num_buffers = 10
CALL CDF_set_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.23 CDF\_set\_checksum

```

SUBROUTINE CDF_set_checksum (
INTEGER*4 id,                            ! in -- CDF identifier.
INTEGER*4 checksum,                    ! in -- Checksum mode.
INTEGER*4 status)                      ! out -- Completion status

```

CDF\_set\_checksum specifies the checksum mode of a CDF file. Refer to Section 4.19 for the description of checksum.

The arguments to CDF\_set\_checksum are defined as follows:

id                The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

checksum        The checksum mode.

status          The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.23.1. Example(s)

The following example sets checksum mode for a CDF.

```

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

.
.
checksum = MD5_CHECKSUM
CALL CDF_set_checksum (id, checksum, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.24 CDF\_set\_compress\_cachesize

```

SUBROUTINE CDF_set_compress_cachesize (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 num_buffers,      ! in -- Number of cache buffers.
INTEGER*4 status)           ! out -- Completion status

```

CDF\_set\_compress\_cachesize specifies the number of cache buffers used for the compression scratch CDF file. Refer to the CDF User's Guide for the description of caching scheme used by the CDF library.

The arguments to CDF\_set\_compress\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
num_buffers	The number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.24.1. Example(s)

The following example sets the number of cache buffers to 10 to be used for the compression scratch CDF file.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.

```

```

INTEGER*4 num_buffers           ! Number of cache buffers.
INTEGER*4 status                ! Returned status code.

.
.
num_buffers = 10
CALL CDF_set_compress_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.25 CDF\_set\_compression

SUBROUTINE CDF\_set\_compression (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 compress_type,    ! in -- Compression type.
INTEGER*4 compress_parms(*), ! in -- Compression parameters.
INTEGER*4 status)           ! out -- Completion status

```

CDF\_set\_compression specifies the compression information of the CDF. It returns the compression type (method) and, if compressed, the compression parameters and compression rate. CDF compression types/parameters are described in Section 4.10.

The arguments to CDF\_set\_compression are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
compress_type	The compression type.
compress_parms	The compression parameters.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.25.1. Example(s)

The following example uses GZIP.9 compression for a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 compress_type     ! Compression type.
INTEGER*4 compress_parms(CDF_MAX_DIMS) ! Compression parameters.
INTEGER*4 status            ! Returned status code.
.
.

```

```

.
compress_type = GZIP_COMPRESSION
compress_parms(1) = 9
CALL CDF_set_compression (id, compress_type, compress_parms,
1                          status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.26 CDF\_set\_decoding

SUBROUTINE CDF\_set\_decoding (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 decoding,    ! in -- CDF decoding.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_decoding specifies the decoding for the data in a CDF. The decodings are described in Section 4.7.

The arguments to CDF\_set\_decoding are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
decoding	The decoding.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.26.1. Example(s)

The following example sets the decoding to NETWORK\_DECODING for a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 decoding     ! Decoding.
INTEGER*4 status       ! Returned status code.
.
.
decoding = NETWORK_DECODING
CALL CDF_set_decoding (id, decoding, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.27 CDF\_set\_encoding

```
SUBROUTINE CDF_set_encoding (  
  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 decoding,    ! in -- CDF encoding.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_encoding specifies the encoding code used for the data in a CDF. The encodings are described in Section 4.6.

The arguments to CDF\_set\_encoding are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
encoding	The encoding.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.27.1. Example(s)

The following example sets the encoding code to NETWORK\_ENCODING to be used for a CDF.

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

## 6.2.28 CDF\_set\_format

```
SUBROUTINE CDF_set_format (  
  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 format,      ! in -- CDF format.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_format specifies the file format, single or multi-file, of the CDF. The formats are described in Section 4.4.

The arguments to CDF\_set\_format are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
format	The format.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.28.1. Example(s)

The following example sets the file format to MULTI\_FILE\_FORMAT for a CDF.

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

.
.
format = MULTI_FILE_FORMAT
CALL CDF_set_format (id, format, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.29 CDF\_set\_majority

SUBROUTINE CDF\_set\_majority (

INTEGER*4 id,	! in -- CDF identifier.
INTEGER*4 majority,	! in -- Variable majority.
INTEGER*4 status)	! out -- Completion status

CDF\_set\_majority specifies the variable majority, row or column-major, of the CDF. The majorities are described in Section 4.8.

The arguments to CDF\_set\_majority are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
majority	The variable majority of the CDF.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.29.1. Example(s)

The following example sets the variable majority to ROW\_MAJOR for a CDF.

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

.
.
majority = ROW_MAJOR
CALL CDF_set_majority (id, majority, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.2.30 CDF\_set\_negtoposfp0\_mode

SUBROUTINE CDF\_set\_negtoposfp0\_mode (

```
INTEGER*4 id,            ! in -- CDF identifier.
INTEGER*4 negtoposfp0, ! in -- -0.0 to 0.0 mode.
INTEGER*4 status)       ! out -- Completion status
```

CDF\_set\_negtoposfp0\_mode specifies -0.0 to 0.0 mode of the CDF. You can use CDF\_get\_negtoposfp0\_mode subroutine to check the mode. The -0.0 to 0.0 modes are described in Section 4.15.

The arguments to CDF\_set\_negtoposfp0\_mode are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

negtoposfp0 The -0.0 to 0.0 mode of the CDF.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.30.1. Example(s)

The following example sets the -0.0 to 0.0 mode to NEGtoPOSfp0off for a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 negtoposfp0 ! -0.0 to 0.0 mode.
INTEGER*4 status       ! Returned status code.

.
.
negtoposfp0 = NEGtoPOSfp0off
CALL CDF_set_negtoposfp0_mode (id, negtoposfp0, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.31 CDF\_set\_readonly\_mode

SUBROUTINE CDF\_set\_readonly\_mode (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 readonly,    ! in -- Read-only mode of the CDF.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_readonly\_mode specifies the read-only mode for a CDF. You can use CDF\_get\_readonly\_mode to check the mode. The read-only modes are described in Section 4.13.

The arguments to CDF\_set\_readonly\_mode are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
readonly	The read-only mode.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.31.1. Example(s)

The following example sets the read-only mode to READONLYoff (to allow read/write) for a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 readonly     ! Read-only mode.

```



```

INTEGER*4 status      ! Returned status code.

.
.
readonly = READONLYoff
CALL CDF_set_readonly_mode (id, readonly, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.32 CDF\_set\_stage\_cachesize

```

SUBROUTINE CDF_set_stage_cachesize (

```

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 num_buffers, ! in -- Number of cache buffers.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_set\_stage\_cachesize respecifies the number of cache buffers being used for the staging scratch file a CDF. Refer to the CDF User's Guide for the description of the caching scheme used by the CDF library.

The arguments to CDF\_set\_stage\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
num_buffers	Number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.32.1. Example(s)

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

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 status     ! Returned status code.
INTEGER*4 num_buffers ! Number of cache buffers.
.
.
num_buffers = 10
CALL CDF_set_stage_cachesize (id, rec_number, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.2.33 CDF\_set\_zmode

```
SUBROUTINE CDF_set_zmode (  
  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 zmode,       ! in -- zMode.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_zmode respecifies the zMode for a CDF file. The zModes are described in Section 4.14.

The arguments to CDF\_set\_zmode are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
zmode	CDF zMode.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.2.33.1. Example(s)

The following example sets zMode to zMODEon2, all rVariables are viewed as zVariables with NOVARY dimensions being eliminated, for a CDF.

```
.  
.   
INCLUDE '<path>cdf.inc'  
.   
.   
INTEGER*4 id           ! CDF identifier.  
INTEGER*4 status       ! Returned status code.  
.   
.   
CALL CDF_set_zmode (id, zMODEon2, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.   
.
```

## 6.3 Variable

This section provides the variable-specific functions. A variable is identified by its unique name in a CDF or a variable number in either rVariable or zVariable group. To operate a variable, the CDF it resides in must be open.

### 6.3.1 CDF\_close\_zvar

```
SUBROUTINE CDF_close_zvar (
```

```
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 var_num,     ! in -- zVariable identifier.  
INTEGER*4 status)      ! out -- Completion status
```

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

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

The arguments to CDF\_close\_zvar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The variable number for the open zVariable's file. This identifier must have been initialized by a call to CDF_create_zvar or CDF_get_var_num.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.1.1. Example(s)

The following example closes an open zVariable "MY\_VAR" in a CDF.

```
.  
.   
INCLUDE '<path>cdf.inc'  
.   
.   
INTEGER*4 id           ! CDF identifier.  
INTEGER*4 var_num      ! Variable identifier.  
INTEGER*4 status       ! Returned status code.  
.   
.   
var_num = CDF_get_var_num(id, 'MY_VAR')  
IF (var_num .LT. 0) CALL UserQuit(..)  
.   
.   
CALL CDF_close_zvar (id, var_num, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.   
.
```

### 6.3.2 CDF\_confirm\_zvar\_existence

```
INTEGER*4 FUNCTION CDF_confirm_zvar_existence (  
  
INTEGER*4 id,           ! in -- CDF identifier.
```

CHARACTER var\_name\*(\*) ! in -- Variable name.

CDF\_confirm\_zvar\_existence confirms the existence of a zVariable with the specified name in a CDF. If the zVariable does not exist, an error code will be returned.

The arguments to CDF\_confirm\_zvar\_existence are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

var\_name     The variable name.

### 6.3.2.1. Example(s)

The following example will check the existence of zVariable “MY\_VAR” in a CDF.

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

### 6.3.3 CDF\_confirm\_zvar\_padvalue\_exist

INTEGER\*4 FUNCTION CDF\_confirm\_zvar\_padvalue\_exist (

INTEGER\*4 id,            ! in -- CDF identifier.  
INTEGER\*4 var\_num)        ! in -- Variable number.

CDF\_confirm\_zvar\_padvalue\_exist confirms the existence of an explicitly specified pad value for the specified zVariable in a CDF. If an explicit pad value has not been specified, the informational status code NO\_PADVALUE\_SPECIFIED will be returned.

The arguments to CDF\_confirm\_zvar\_padvalue\_exist are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

var\_num     The variable number.

### 6.3.3.1. Example(s)

The following example will check the existence of the pad value for zVariable “MY\_VAR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 var_num      ! Variable number.
INTEGER*4 status       ! Returned status code.
.
.
var_num = CDF_get_var_num(id, 'MY_VAR')
IF (var_num .LT. 1) CALL UserQuit(...)
Status = CDF_confirm_zvar_padvalue_exist (id, var_num)
IF (status .NE. NO_PADVALUE_SPECIFIED) THEN
.
.
END IF
.
.
```

### 6.3.4 CDF\_create\_zvar

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

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

The arguments to CDF\_create\_zvar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_name	The name of the zVariable to create. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
data_type	The data type of the new zVariable. 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.
num_dims	The zVariable's number of dimension.
dim_sizes	The zVariable's dimension sizes. Each element of dim_sizes specifies the number of values in corresponding dimension. For 0-dimensional zVariables this argument is ignored (but must be present).
rec_variance	The zVariable's record variance. Specify one of the variances defined in Section 4.9.
dim_variances	The zVariable'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 zVariables this argument is ignored (but must be present).
var_num	The number assigned to the new zVariable. This number must be used in subsequent CDF function calls when referring to this zVariable. An existing zVariable's number may be determined with the CDF_get_var_num function.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.4.1. Example(s)

The following example will create several zVariables in a CDF. In this case, EPOCH is a 0-dimensional of CDF\_EPOCH data type, LAT a 1-dimensional of 2 elements of CDF\_INT2 data type, LON a 2-dimensional with 2 by 3 of CDF\_INT2 data type and TMP a 2 dimensional with 2 by 3 of CDF\_REAL4 data type.

```

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

INTEGER*4 EPOCH_rec_var    ! EPOCH record variance.
INTEGER*4 LAT_rec_var      ! LAT record variance.
INTEGER*4 LON_rec_var      ! LON record variance.
INTEGER*4 TMP_rec_var      ! 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 zVariable number.
INTEGER*4 LON_var_num      ! LON zVariable number.
INTEGER*4 TMP_var_num      ! TMP zVariable number.
INTEGER*4 num_dims_EPOCH, num_dims_LAT, num_dims_LON,
1      num_dims_TEMP      ! Number of dimensions.
INTEGER*4 dim_sizes_EPOCH(1), dim_sizes_LAT(1),
1      dim_sizes_LON(2), dim_sizes_TEMP(2)

```

```

                                ! Dimesion sizes.

DATA num_dims_EPOCH/0/, num_dims_LAT/1/,
1   num_dims_LON/2/, num_dims_TEMP/2/

DATA dim_sizes_EPOCH/1/, dim_sizes_LAT/3/,
1   dim_sizes_LON/2,3/, dim_sizes_TEMP/2,3/

DATA EPOCH_rec_vary/VARY/, LAT_rec_vary/NOVARY/,
1   LON_rec_vary/NOVARY/, TMP_rec_vary/VARY/

DATA EPOCH_dim_varys/NOVARY/, LAT_dim_varys/VARY/,
1   LON_dim_varys/VARY,VARY/, TMP_dim_varys/VARY,VARY/
.
.
CALL CDF_create_zvar (id, 'EPOCH', CDF_EPOCH, 1, num_dims_EPOCH,
1   dim_sizes_EPOCH,
2   EPOCH_rec_vary, EPOCH_dim_varys, POCH_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_create_zvar (id, 'LATITUDE', CDF_INT2, 1, num_dims_LAT,
1   dim_sizes_LAT,
2   LAT_rec_vary, LAT_dim_varys, LAT_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_create_zvar (id, 'LONGITUDE', CDF_INT2, 1, num_dims_LON,
1   dim_sizes_LON,
2   LON_rec_vary, LON_dim_varys, LON_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

CALL CDF_create_zvar (id, 'TEMPERATURE', CDF_REAL4, 1, num_dims_TEMP,
1   dim_sizes_TEMP,
2   TMP_rec_vary, TMP_dim_varys, TMP_var_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.5 CDF\_delete\_zvar

SUBROUTINE CDF\_delete\_zvar (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 var_num,    ! in -- zVariable number.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_delete\_zvar deletes the specified zVariable from a CDF

The arguments to CDF\_delete\_zvar are defined as follows:

**id**            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

var\_num    The zVariable number.

status     The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.5.1. Example(s)

The following example will delete the zVariable “MY\_VAR” in a CDF.

```

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

```

### 6.3.6 CDF\_delete\_zvar\_recs

```

SUBROUTINE CDF_delete_zvar_recs (
INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 var_num,              ! in -- zVariable number.
INTEGER*4 start_rec,            ! in -- Starting record number.
INTEGER*4 end_rec,              ! in -- Ending record number.
INTEGER*4 status)               ! out -- Completion status

```

CDF\_delete\_zvar\_recs deletes a range of data records from the specified zVariable in a CDF

The arguments to CDF\_delete\_zvar\_recs are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

var\_num      The zVariable number.

start\_rec    The starting record number to delete.

end\_rec      The ending record number to delete.

status       The completion status code. Chapter 8 explains how to interpret status codes.



### 6.3.6.1. Example(s)

The following example will delete 10 records (from record number 10 to 19) from the zVariable “MY\_VAR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
.
.
CALL CDF_delete_zvar_recs (id, CDF_get_var_num(id, 'MY_VAR'), 10, 19, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.3.7 CDF\_get\_num\_zvars

```
SUBROUTINE CDF_get_num_zvars (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 vars,              ! out -- Number of zVariables.
INTEGER*4 status)            ! out -- Completion status
```

CDF\_get\_num\_zvars acquires the total number of zVariables in a CDF.

The arguments to CDF\_get\_num\_zvars are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- vars         The number of zVariables.
- status       The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.7.1. Example(s)

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

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

```

INTEGER*4 vars          ! zVariables.
INTEGER*4 status       ! Returned status code.

.
.
CALL CDF_get_num_zvars (id, vars, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.8 CDF\_get\_var\_num

```

INTEGER*4 FUNCTION CDF_get_var_num (

```

```

INTEGER*4 id,           ! in-- CDF identifier.
CHARACTER var_name*(*) ! in-- Variable name.

```

CDF\_get\_var\_num is used to determine the number associated with the specified variable name. If the Variable is found, CDF\_get\_var\_num returns its number - which will be equal to or greater than one (1). If an error occurs (e.g., the Variable does not exist in the CDF), an error code (of type INTEGER\*4) is returned. Error codes are less than zero (0).

Initially, this function can only handle rVariables. As the variable name is unique in a CDF file, no two variables, either rVariable or zVariable can have the same name. This function is now extended to include zVariable. The variable number it returns represents the number in either the rVariable group or zVariable group wherever the variable exists.

The arguments to CDF\_get\_var\_num are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
varName	The name of the Variable for which to search. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.

CDF\_get\_var\_num may be used as an embedded function call when a Variable number is needed. CDF\_get\_var\_num is declared in cdf.inc. (Fortran functions must be declared so that the returned value is interpreted correctly.)

#### 6.3.8.1. Example(s)

In the following example CDF\_get\_var\_num is used as an embedded function call when inquiring about an rVariable and a zVariable.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status      ! Returned status code.
CHARACTER var_name1*(CDF_VAR_NAME_LEN256) ! rVariable name.

```

```

CHARACTER var_name2*(CDF_VAR_NAME_LEN256)    ! zVariable name.

INTEGER*4 data_type1, data_type1            ! Data type of the rVariable.
INTEGER*4 num_elems1, num_elems2           ! Number of elements (of the
                                           ! data type).

INTEGER*4 rec_vary1, rec_vary2             ! Record variance.
INTEGER*4 num_dims2                        ! Number of dimensions
INTEGER*4 dim_sizes2(CDF_MAX_DIMS)        ! Dimension sizes
INTEGER*4 dim_variances1(CDF_MAX_DIMS)    ! Dimension variances.
INTEGER*4 dim_variances2(CDF_MAX_DIMS)    ! Dimension variances..
.
CALL CDF_var_inquire (id, CDF_get_var_num(id,'LATITUDE'), var_name1,
1                      data_type1, num_elems1, rec_vary1, dim_variances1,
2                      status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
CALL CDF_inquire_zvar (id, CDF_get_var_num(id,'LONGITUDE'), var_name1,
1                      data_type2, num_elems2, num_dims2, dim_sizes2,
2                      rec_vary2, dim_variances2, 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\_get\_var\_num would have returned an error code. Passing that error code to CDF\_inquire\_rvar as an rVariable number would have resulted in CDF\_inquire\_rvar 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 5.24.

### 6.3.9 CDF\_get\_vars\_maxwrittenrecnums

```

SUBROUTINE CDF_get_vars_maxwrittenrecnums (

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 rvars_maxrec,     ! out -- Maximum record number among rVariables.
INTEGER*4 zvars_maxrec,     ! out -- Maximum record number among zVariables.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_get\_vars\_maxwrittenrecnums inquires the maximum written record numbers among all rVariables and zVariables in a CDF.

The arguments to CDF\_get\_vars\_maxwrittenrecnums are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
rvars_maxrec	Maximum record number among rVariables.
zvars_maxrec	Maximum record number among zVariables.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.9.1. Example(s)

The following example acquires the maximum record numbers from all rVariables and zVariables in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 rvars_maxrec ! Maximum record number among rVariables.
INTEGER*4 zvars_maxrec ! Maximum record number among zVariables.
.
.
CALL CDF_get_vars_maxwrittenrecnums (id, rvars_maxrec, zvars_maxrec, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.3.10 CDF\_get\_zvar\_allocrecs

```
SUBROUTINE CDF_get_zvar_allocrecs (
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_recs,   ! out -- Number of allocated records.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_get\_zvar\_allocrecs inquires the number of records allocated for the specified zVariable in a CDF. Refer to the CDF User's Guide for the description of allocating variable records in a single-file CDF.

The arguments to CDF\_get\_zvar\_allocrecs are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- var\_num      The zVariable number.
- Num\_recs    The number of records allocated for the variable.
- status      The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.10.1. Example(s)

The following example acquires the number of records allocated for zVariable "MY\_VAR" in a CDF.

```
.
.
```

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 num_recs    ! Number of allocated records.
INTEGER*4 status      ! Returned status code.

.
.
CALL CDF_get_zvar_allocrecs (id, CDF_get_var_num(id, 'MY_VAR'),
1 num_recs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.11 CDF\_get\_zvar\_blockingfactor

SUBROUTINE CDF\_get\_zvar\_blockingfactor (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 var_num,    ! in -- zVariable number.
INTEGER*4 bf,         ! out -- Variable blocking factor.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_zvar\_blockingfactor inquires the blocking factor for the specified zVariable in a CDF. Refer to the CDF User's Guide for the description of the blocking factor.

The arguments to CDF\_get\_zvar\_blockingfactor are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
bf	The blocking factor of the variable.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.11.1. Example(s)

The following example acquires the blocking factor for zVariable "MY\_VAR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 bf          ! Blocking factor.

```

```

INTEGER*4 status      ! Returned status code.

.
.
CALL CDF_get_zvar_blockingfactor (id, CDF_get_var_num(id, 'MY_VAR'),
1                                bf, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.12 CDF\_get\_zvar\_cachesize

```

SUBROUTINE CDF_get_zvar_cachesize (

```

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 var_num,    ! in -- zVariable number.
INTEGER*4 num_buffers, ! out -- Variable number of cache buffers.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_zvar\_cachesize inquires the number of cache buffers being for the specified zVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User's Guide for the description about caching scheme used by the CDF library.

The arguments to CDF\_get\_zvar\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
num_buffers	The number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.12.1. Example(s)

The following example acquires the number of cache buffers used for zVariable "MY\_VAR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 num_buffers ! Number of cache buffers.
INTEGER*4 status      ! Returned status code.

.
.
CALL CDF_get_zvar_cachesize (id, CDF_get_var_num(id, 'MY_VAR'),

```

```

1                                num_buffers, status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
  .
  .

```

### 6.3.13 CDF\_get\_zvar\_compression

SUBROUTINE CDF\_get\_zvar\_compression (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 compress_type,    ! out -- Compression type.
INTEGER*4 compress_parms,   ! out -- Compression parameters.
INTEGER*4 compress_percent, ! out -- Compression percentage.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_get\_zvar\_compression inquires the compression type/parameters of the specified zVariable in a CDF. Refer to Section 4.10 for the description of the CDF supported compression types/parameters.

The arguments to CDF\_get\_zvar\_compression are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
compress_type	The compression type.
compress_parms	The compression parameters.
compress_percent	The compression percentage.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.13.1. Example(s)

The following example acquires the compression type/parameters for zVariable “MY\_VAR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 ctype             ! Compression type.
INTEGER*4 cparms(CDF_MAX_DIMS) ! Compression parameters.
INTEGER*4 cpercent         ! Compression percentage.
INTEGER*4 status           ! Returned status code.
.

```

```

      .
      CALL CDF_get_zvar_compression (id, CDF_get_var_num(id, 'MY_VAR'),
1      ctype, cparms, cpercent, status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
      .
      .

```

### 6.3.14 CDF\_get\_zvar\_data

SUBROUTINE CDF\_get\_zvar\_data (

```

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

```

CDF\_get\_zvar\_data is used to read a single value from a zVariable. CDF\_hyper\_get\_zvar\_data may be used to read more than one zVariable values with a single call (see Section 6.3.33).

The arguments to CDF\_get\_zvar\_data are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
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 zVariables this argument is ignored (but must be present).
value	The value read. This buffer must be large enough to hold the value. CDF_inquire_zvar would be used to determine the zVariable's data type and number of elements (of that data type) at each value. The value is read from the CDF and placed at memory address value.  <b>WARNING:</b> 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.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.14.1. Example(s)

The following example reads and hold an entire record of data from zVariable “Temperature” in a CDF. This zVariable is 3-dimensional with sizes [180,91,10]. 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        ! zVariable number.
INTEGER*4 rec_num      ! Record number.
INTEGER*4 d1, d2, d3   ! Dimension index values.
.
.
var_n = CDF_get_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_get_zvar_data (id, var_n, rec_num, indices, tmp(d1,d2,d3),
1      status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    END DO
  END DO
END DO
END DO
.
.

```

### 6.3.15 CDF\_get\_zvar\_datatype

SUBROUTINE CDF\_get\_zvar\_datatype (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,      ! in -- zVariable number.
INTEGER*4 data_type,    ! out -- Data type.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvar\_datatype is used to acquire the data type of the specified zVariable in a CDF. Refer to Section 4.5 for the description of the CDF data types.

The arguments to CDF\_get\_zvar\_datatype are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
----	--------------------------------------------------------------------------------------------------------------------

var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
data_type	The data type of the variable data.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.15.1. Example(s)

The following example acquires the data type of zVariable “Temperature” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 data_type    ! Data type.
.
.
CALL CDF_get_zvar_datatype (id, CDF_get_var_num (id, 'Temperature'),
1                          data_type, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.16 CDF\_get\_zvar\_dimsizes

SUBROUTINE CDF\_get\_zvar\_dimsizes (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 dim_sizes(*), ! out -- Dimension sizes.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvar\_dimsizes acquires the size of each dimension for the specified zVariable in a CDF. For 0-dimensional zVariables, this operation is not applicable.

The arguments to CDF\_get\_zvar\_dimsizes are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	zVariable number.
dim_sizes	Dimension sizes.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.16.1. Example(s)

The following example acquires the dimension sizes for zVariable “MY\_VAR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 dim_sizes(CDF_MAX_DIMS) ! Dimension sizes.
.
.
CALL CDF_get_zvar_dimsizes (id, CDF_get_var_num(id, 'MY_VAR'), dim_sizes,
1                          status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.3.17 CDF\_get\_zvar\_dimvariances

```
SUBROUTINE CDF_get_zvar_dimvariances (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 dim_varys(*),     ! out -- Dimension variances.
INTEGER*4 status)           ! out -- Completion status
```

CDF\_get\_zvar\_dimvariances acquires the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. Refer to Section 4.9 for the description of the CDF variable’s dimension variances.

The arguments to CDF\_get\_zvar\_dimvariances are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
dim_varys	The dimension variances.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.17.1. Example(s)

The following example acquires the dimension variances for zVariable “Temperature” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 dim_varys(CDF_MAX_DIMS)! Dimension variances.
.
.
CALL CDF_get_zvar_dimvariances (id, CDF_get_var_num (id, 'Temperature'),
1 dim_varys, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.18 CDF\_get\_zvar\_maxallocrecnum

SUBROUTINE CDF\_get\_zvar\_maxallocrecnum (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 rec_num,         ! out -- Maximum allocated record number.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_get\_zvar\_maxallocrecnum acquires the maximum record number allocated for the specified zVariable in a CDF.

The arguments to CDF\_get\_zvar\_maxallocrecnum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
rec_num	The maximum record number allocated.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.18.1. Example(s)

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

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 rec_num          ! Maximum allocated record number.

```

```

.
.
CALL CDF_get_zvar_maxallocrecnum (id, CDF_get_var_num (id, 'Temperature'),
1                                rec_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.19 CDF\_get\_zvar\_maxwrittenrecnum

SUBROUTINE CDF\_get\_zvar\_maxwrittenrecnum (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 rec_num,     ! out -- Maximum written record number.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvar\_maxwrittenrecnum acquires the maximum record number written for the specified zVariable in a CDF.

The arguments to CDF\_get\_zvar\_maxwrittenrecnum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
rec_num	The maximum record number written.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.19.1. Example(s)

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

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 rec_num      ! Maximum written record number.
.
.
CALL CDF_get_zvar_maxwrittenrecnum (id, CDF_get_var_num (id, 'Temperature'),
1                                rec_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.20 CDF\_get\_zvar\_name

```
SUBROUTINE CDF_get_zvar_name (  
  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 var_num,     ! in -- zVariable number.  
CHARACTER var_name*(*), ! out -- zVariable name.  
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_zvar\_name acquires the name of the specified zVariable, by its number, in a CDF.

The arguments to CDF\_get\_zvar\_name are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
var_name	The name of the variable.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.20.1. Example(s)

The following example acquires the name of the zVariable, numbered 2 in the zVariable group, in a CDF.

```
.  
.  
INCLUDE '<path>cdf.inc'  
.  
.  
INTEGER*4 id           ! CDF identifier.  
INTEGER*4 status       ! Returned status code.  
INTEGER*4 var_num      ! zVariable number.  
INTEGER*4 var_name*(CDF_VAR_NAME_LEN256) ! zVariable name.  
.  
.  
rec_num = 2  
CALL CDF_get_zvar_name (id, var_num, var_name, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.  
.
```

## 6.3.21 CDF\_get\_zvar\_numdims

```
SUBROUTINE CDF_get_zvar_numdims (  
  

```

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_dims,   ! out -- Number of dimensions.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_zvar\_numdims acquires the number of dimensions for the specified zVariable in a CDF.

The arguments to CDF\_get\_zvar\_numdims are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	zVariable number.
num_dims	Number of dimensions.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.21.1. Example(s)

The following example acquires the number of dimensions for zVariable “MY\_VAR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_dims     ! Dimension sizes.
.
.
CALL CDF_get_zvar_numdims (id, CDF_get_var_num(id, 'MY_VAR'), num_dims,
1                          status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.22 CDF\_get\_zvar\_numelems

SUBROUTINE CDF\_get\_zvar\_numelems (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_elems,   ! out -- Number of elements.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_zvar\_numelems acquires the number of elements for each data value of the specified zVariable in a CDF. For character data type (CDF\_CHAR and CDF\_UCHAR), the number of elements is the number of characters in the string. (Each value consists of the entire string.) For other data types, the number of elements will always be one (1).

The arguments to CDF\_get\_zvar\_numelems are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
num_elems	The number of elements.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.22.1. Example(s)

The following example acquires the number of elements for the data values for zVariable “Temperature” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_elems    ! Number of elements.
.
.
CALL CDF_get_zvar_numelems (id, CDF_get_var_num (id, 'Temperature'),
1  num_elems, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.3.23 CDF\_get\_zvar\_numrecs\_written

SUBROUTINE CDF\_get\_zvar\_numrecs (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_records, ! out -- Number of written records.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_zvar\_numrecs\_written acquires the number of records written for the specified zVariable in a CDF. This number may not correspond to the maximum record written if the zVariable has sparse records.

The arguments to CDF\_get\_zvar\_numrecs\_written are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
----	--------------------------------------------------------------------------------------------------------------------



var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
num_records	The number of written records.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.23.1. Example(s)

The following example acquires the number of written records for zVariable “Temperature” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_records  ! Number of written records.
.
.
CALL CDF_get_zvar_numrecs_written (id, CDF_get_var_num (id, 'Temperature'),
1                                num_records, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.24 CDF\_get\_zvar\_padvalue

SUBROUTINE CDF\_get\_zvar\_padvalue (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
<type> pad_value,      ! out -- Pad value.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvar\_padvalue acquires the pad value of the specified zVariable in a CDF. If a pad value has not been explicitly specified for the zVariable through CDF\_set\_zvar\_padvalue or something similar from the Internal Interface function, the informational status code NO\_PADVALUE\_SPECIFIED will be returned and the default pad value for the variable’s data type will be placed in the pad value buffer provided.

The arguments to CDF\_get\_zvar\_padvalue are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
pad_value	The pad value.

status                    The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.24.1. Example(s)

The following example acquires acquire the pad value from zVariable “MY\_VAR”, a CDF\_INT4 type variable in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                    ! CDF identifier.
INTEGER*4 status                ! Returned status code.
INTEGER*4 pad_value             ! Pad value.
.
.
CALL CDF_get_zvar_padvalue (id, CDF_get_var_num (id, 'MY_VAR'),
1                                pad_value, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.3.25 CDF\_get\_zvar\_recorddata

```
SUBROUTINE CDF_get_zvar_recorddata (
INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 var_num,              ! in -- zVariable number.
INTEGER*4 rec_num,              ! in -- Record number.
<type> buffer,                  ! out -- Record data buffer.
INTEGER*4 status)               ! out -- Completion status
```

CDF\_get\_zvar\_recorddata acquires an entire record at a given record number for the specified zVariable in a CDF. The buffer should be large enough to hold the entire data values for the variable. The retrieved data values in the buffer are in the order that corresponds to the variable majority defined for the CDF.

The arguments to CDF\_get\_zvar\_recorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
rec_num	The record number of the zVariable from which to read.
buffer	The record buffer to hold the data values from an entire record.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.25.1. Example(s)

The following example acquires an entire record, at numbered 5, for zVariable “MY\_VAR”, a 2-dimensional variable (2 by 3) of CDF\_INT4 data type, in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 buffer(2,3)  ! Record buffer.
.
.
CALL CDF_get_zvar_recorddata (id, CDF_get_var_num (id, 'MY_VAR'), 5,
1      buffer, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.3.26 CDF\_get\_zvar\_recvariance

```
SUBROUTINE CDF_get_zvar_recvariance (
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 rec_vary,    ! out -- Record variance.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_zvar\_recvariance acquires the record variance of the specified zVariable in a CDF. Refer to Section 4.9 for the description of the CDF variable’s record variance.

The arguments to CDF\_get\_zvar\_recvariance are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
rec_vary	The record variance.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.26.1. Example(s)

The following example acquires the record variance for zVariable “Temperature” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 rec_vary          ! Record variance.
.
.
CALL CDF_get_zvar_recvariance (id, CDF_get_var_num (id, 'Temperature'),
1                               rec_vary, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.27 CDF\_get\_zvar\_reservepercent

SUBROUTINE CDF\_get\_zvar\_reservepercent (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 res_percent,      ! out -- Reserve percentage.
INTEGER*4 status)           ! out -- Completion status

```

CDF\_get\_zvar\_reservepercent acquires the reserve percentage being used for the specified zVariable in a CDF. This operation only applies to compressed zVariables. Refer to the CDF User's Guide for the description of the reserve scheme used by the CDF library.

The arguments to CDF\_get\_zvar\_reservepercent are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
res_percent	The reserve percentage.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.27.1. Example(s)

The following example acquires the reserve percentage for the compressed zVariable "Temperature" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.

```

```

INTEGER*4 status          ! Returned status code.
INTEGER*4 res_percent     ! Reserve percentage.
.
.
CALL CDF_get_zvar_reservepercent (id, CDF_get_var_num (id, 'Temperature'),
1                                res_percent, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.28 CDF\_get\_zvar\_seqdata

SUBROUTINE CDF\_get\_zvar\_seqdata (

```

INTEGER*4 id,             ! in -- CDF identifier.
INTEGER*4 var_num,       ! in -- zVariable number.
<type> value,            ! out -- Data value.
INTEGER*4 status)        ! out -- Completion status

```

CDF\_get\_zvar\_seqdata reads one data value at the current sequential value for the specified zVariable in a CDF. After the read, the current sequential value is automatically incremented to the next value. An error is returned if the current sequential value is past the last record of the zVariable. Use CDF\_set\_zvar\_seqpos and CDF\_get\_zvar\_seqpos subroutine calls to set and get the current sequential value (position) for the variable.

The arguments to CDF\_get\_zvar\_seqdata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
value	The data value buffer.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.28.1. Example(s)

The following example reads two data values from the beginning of record (numbered 2) from a zVariable, a 2-dimensional CDF\_INT4 type variable, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id              ! CDF identifier.
INTEGER*4 status          ! Returned status code.
INTEGER*4 var_num         ! Variable number.
INTEGER*4 value1, value2  ! Variable data values.
INTEGER*4 rec_num         ! Record number.
INTEGER*4 indices(2)     ! Dimension indices.

```

```

.
.
rec_num = 2
indices(1) = 0
indices(2) = 0
CALL CDF_set_zvar_seqpos (id, var_num, rec_num, indices, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
CALL CDF_get_zvar_seqdata (id, var_num, value1, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
CALL CDF_get_zvar_seqdata (id, var_num, value2, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.29 CDF\_get\_zvar\_seqpos

SUBROUTINE CDF\_get\_zvar\_seqpos (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 rec_num,     ! out -- Record number.
INTEGER*4 indices(*), ! out -- Indices in a record.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_zvar\_seqpos acquires the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDF\_get\_zvar\_seqdata subroutine to get the data value.

The arguments to CDF\_get\_zvar\_seqpos are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
rec_num	The record number.
Indices	The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored, but must be presented.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.29.1. Example(s)

The following example inquires the location for the current sequential value, the record number and indices within it, from a 2-dimensional zVariable “MY\_VAR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'

```

```

.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.

INTEGER*4 rec_num     ! Record number.
INTEGER*4 indices(2)  ! Dimension indices.
.
.
CALL CDF_get_zvar_seqpos (id, CDF_get_var_num(id, 'MY_VAR'), rec_num,
1      indices, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.30 CDF\_get\_zvars\_maxwrittenrecnum

SUBROUTINE CDF\_get\_zvars\_maxwrittenrecnum (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 rec_num,     ! out -- Maximum record number.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvars\_maxwrittenrecnum acquires the maximum written record number among all of the zVariables in a CDF. A value of zero (0) indicates that zVariables contain no records. The maximum record number for an individual zVariable may be acquired using the CDF\_get\_zvar\_maxwrittenrecnum function call.

The arguments to CDF\_get\_zvars\_maxwrittenrecnum are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
rec_num	The maximum record number among all zVariables.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.30.1. Example(s)

The following example acquires the maximum written record number among all zVariables in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 rec_num     ! Record number.
.
.
CALL CDF_get_zvars_maxwrittenrecnum (id, rec_num, status)

```

```

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

```

### 6.3.31 CDF\_get\_zvar\_sparserecords

SUBROUTINE CDF\_get\_zvar\_sparserecords (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 srecords_type, ! out -- Sparse records type.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_zvar\_sparserecords acquires the sparse records type of the specified zVariable in a CDF. Refer to Section 4.11 for the description of the sparse records.

The arguments to CDF\_get\_zvar\_sparserecords are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	zVariable number.
srecords_type	Sparse records type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.31.1. Example(s)

The following example inquires the sparse records type for zVariable 'MY\_VAR' in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 srecords_type ! Sparse records type.
  INTEGER*4 num_dims   ! Dimension sizes.
.
.
CALL CDF_get_zvar_sparserecords (id, CDF_get_var_num(id, "MY_VAR"),
1 srecords_type, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```



### 6.3.32 CDF\_get\_zvars\_recorddata

SUBROUTINE CDF\_get\_zvars\_recorddata(

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 num_var,     ! in -- Number of zVariables.
INTEGER*4 var_nums(*), ! in -- zVariable numbers.
INTEGER*4 rec_num,     ! in -- Record number.
<type> buffer,        ! out -- First variable buffer in a common block (<type> depends
                      !         on the data type of the zVariable).
INTEGER*4 status       ! out -- Completion status.
```

CDF\_get\_zvars\_recorddata is used to read a full record data at a specific record number for a selected group of zVariables in a CDF. It expects that the data buffer for each zVariable is big enough to hold a full physical record<sup>3</sup> data and properly put in a common block. No space is needed for each zVariable's non-variant dimensional elements. Retrieved record data from the variable group is filled into respective zVariable's buffer.

The arguments to CDF\_get\_zvars\_recorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, Cdf_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the zVariables in the group involved this read operation.
var_nums	The numbers of the zVariables involved for which to read a whole record data.
rec_num	The record number at which to read the whole record data for the group of zVariables.
buffer	The first variable buffer to read in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.

#### 6.3.32.1. Example(s)

The following example will read an entire single record data for a group of zVariables. The zVariables involved in the read are **Time**, **Longitude**, **Delta**, **Temperature** and **NAME**. The record to read is **4**. Since **Temperature** is 0-dimensional with **CDF\_FLOAT** data type, a scalar variable of REAL\*4 is allocated. For **Longitude**, a 1-dimensional array of INTEGER\*2 (size [3]) is given for its dimension variance [VARY] and data type **CDF\_INT2**. Similar data variables are provided for **Longitude** and **Time**. They both are 2-dimensional array of INTEGER\*4 (sizes [3,2]) for their dimension variances [VARY,VARY] and data type either **CDF\_INT4** or **CDF\_UINT4**. For **NAME**, a 1-dimensional array of CHARACTER\*10 (size [2]) is allocated due to its [VARY] dimension variance and **CDF\_CHAR** data type with the number of element 10.

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

```
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_var      ! Number of zVariables.
INTEGER*4 var_nums(5) ! zVariable numbers in CDF.
INTEGER*4 rec_num      ! Record number to write.
```

<sup>3</sup> Physical record is explained in the Primer chapter in the CDF User's Guide.

```

INTEGER*4    time(3,2)      ! Datatype: UINT4.
                                ! Rec/dim variances: T/TT.
INTEGER*4    delta(3,2)    ! Datatype: INT4 .
                                ! Rec/dim variances: T/TT.
INTEGER*2    longitude(3)  ! Datatype: INT2.
                                ! Rec/dim variances: T/T.
REAL*4       temperature   ! Datatype: FLOAT.
                                ! Rec/dim variances: T/.
CHARACTER*10 name(2)       ! Datatype: CHAR/10.
                                ! Rec/dim variances: T/T.
COMMON /BLK/delta, time, temperature, longitude, name
.
.
num_var = 5                    ! Number of zVariables
rec_num = 4                    ! Record number to read

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Delta', var_nums(1),
1                NULL_, status) ! zVariable number
IF (var_nums(1) .LT. 1)        ! If less than one (1),
x CALL UserStatusHandler (var_nums(1)) ! then it is actually a
                                ! warning/error code.

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Time', var_nums(2),
1                NULL_, status)
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Longitude', var_nums(3),
1                NULL_, status)
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Temperature', var_nums(4),
1                NULL_, status)
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'NAME', var_nums(5),
1                NULL_, status)
IF (var_nums(5) .LT. 1) CALL UserStatusHandler (var_nums(5))

CALL CDF_get_zvars_reorddata (id, num_var, var_nums, rec_num,
1                time, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <GET\_, zVARs\_RECDDATA\_>.

### 6.3.33 CDF\_hyper\_get\_zvar\_data

```

SUBROUTINE CDF_hyper_get_zvar_data (

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable 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,      ! in -- Buffer of values (<type> is dependent on the data type of the zVariable).
INTEGER*4 status)      ! out -- Completion status

```

CDF\_hyper\_get\_zvar\_data is used to read a buffer of one or more values from a zVariable. It is important to know the variable majority of the CDF before using CDF\_hyper\_get\_zvar\_data because the values placed into the buffer will be in that majority. CDF\_inquire can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to CDF\_hyper\_get\_zvar\_data are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
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. If there are zero (0) dimensions, this argument is ignored (but must be present).
counts	The number of values along each dimension to read. Each element of counts specifies the corresponding dimension count. For 0-dimensional zVariables 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 zVariables, this argument is ignored (but must be present).
buffer	The buffer of values read. The majority of the values in this buffer will be the same as that of the CDF. This buffer must be large to hold the values. CDF_var_inquire would be used to determine the zVariable's data type and number of elements (of that data type) at each value. The values are read from the CDF and placed into memory starting at address buffer.
	<b>WARNING:</b> 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.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.33.1. Example(s)

The following example reads an entire record of data from zVariable “Temperature” in a CDF. This zVariable is 3-dimensional with sizes [180,91,10] and CDF’s variable majority is ROW\_MAJOR. The record variance is VARY, the dimension variances are [VARY,VARY,VARY], and the data type is CDF\_REAL4. This example is similar to the example in Section 6.3.33 except that it uses a single call to CDF\_hyper\_get\_zvar\_data rather than numerous calls to CDF\_get\_zvar\_data.

```
.
.
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_get_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_hyper_get_zvar_data (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.

### 6.3.34 CDF\_hyper\_put\_zvar\_data

```
SUBROUTINE CDF_hyper_put_zvar_data (

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable 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> is dependent on the data type of the zVariable).
INTEGER*4 status)        ! out -- Completion status

```

CDF\_hyper\_put\_zvar\_data is used to write a buffer of one or more values to a zVariable. It is important to know the variable majority of the CDF before using CDF\_hyper\_put\_zvar\_data because the values in the buffer to be written must be in the same majority. CDF\_inquire can be used to determine the default variable majority of a CDF distribution. The Concepts chapter in the CDF User's Guide describes the variable majorities.

The arguments to CDF\_hyper\_put\_zvar\_data are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to write. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
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 <sup>4</sup> (e.g., An interval of 2 means write to every other record).
indices	The indices (within each record) at which to start writing. Each element of indices specifies the corresponding dimension index. If there are zero (0) dimensions, this argument is ignored (but must be present).
counts	The number of values along each dimension to write. Each element of count specifies the corresponding dimension count. For 0-dimensional zVariables this argument is ignored (but must be present).
intervals	For each dimension the interval between values for subsampling <sup>5</sup> (e.g., an interval of 2 means write to every other value). intervals is a 1-dimensional array containing one element per zVariable dimension. Each element of intervals specifies the corresponding dimension interval. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).
buffer	The buffer of values to write. The majority of the values in this buffer must be the same as that of the CDF. The values starting at memory address buffer are written to the CDF.  <b>WARNING:</b> 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 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 8 explains how to interpret status codes.

### 6.3.34.1. Example(s)

<sup>4</sup> "Subsampling" is not the best term to use when writing data, but you should know what we mean.

<sup>5</sup> Again, not the best term.

The following example writes values to the zVariable LATITUDE of a CDF. This zVariable is 2-dimensional with dimension sizes [360,181]. 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 6.3.34 except that it uses a single call to CDF\_hyper\_put\_zvar\_data rather than numerous calls to CDF\_put\_zvar\_data.

```

.
.
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             ! zVariable 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_get_var_num (id, 'LATITUDE')
IF (var_n .LT. 1) CALL UserStatusHandler (var_n) ! If less than one (1),
! then not a zVariable
! number but rather a
! warning/error code

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

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

```

### 6.3.35 CDF\_inquire\_zvar

SUBROUTINE CDF\_inquire\_zvar (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,           ! in -- zVariable number.
CHARACTER var_name*(CDF_VAR_NAME_LEN256), ! out -- zVariable name.
INTEGER*4 data_type,        ! out -- Data type.
INTEGER*4 num_elements,     ! out -- Number of elements (of the data type).
INTEGER*4 num_dims,         ! out -- Number of dimensions.
INTEGER*4 dim_sizes(CDF_MAX_DIMS), ! out -- Dimension sizes.
INTEGER*4 rec_variance,     ! out -- Record variance.

```

```

INTEGER*4  dim_variances(CDF_MAX_DIMS),      ! out -- Dimension variances.
INTEGER*4  status)                          ! out -- Completion status

```

CDF\_inquire\_zvar is used to inquire about the specified zVariable. This subroutine would normally be used before reading zVariable values (with CDF\_get\_zvar\_data or CDF\_hyper\_get\_zvar\_data) to determine the data type and number of elements (of that data type).

The arguments to CDF\_inquire\_zvar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open.
var_num	The number of the zVariable to inquire. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
var_name	The zVariable's name. This character string must be large enough to hold CDF_VAR_NAME_LEN256 characters and will be blank padded if necessary.
data_type	The data type of the zVariable. The data types are defined in Section 4.5.
num_elements	The number of elements of the data type at each zVariable value. For character data types (CDF_CHAR and CDF_UCHAR), this is the number of characters in the string. (Each value consists of the entire string.) For all other data types, this will always be one (1) - multiple elements at each value are not allowed for non-character data types.
num_dims	The number of dimensions.
dim_sizes	The dimension sizes. It is a 1-dimensional array, containing one element per dimension. Each element of dimSizes receives the corresponding dimension size. For 0-dimensional zVariable this argument is ignored (but must be present).
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 zVariable this argument is ignored (but must be present).
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.35.1. Example(s)

The following example inquires about a zVariable named HEAT\_FLUX in a CDF. Note that the zVariable name returned by CDF\_inquire\_zvar will be the same as that passed in to CDF\_get\_var\_num.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
CHARACTER var_name*(CDF_VAR_NAME_LEN256) ! zVariable 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).
INTEGER*4 num_dims                      ! Number of dimensions.
INTEGER*4 dim_sizes(CDF_MAX_DIMS)     ! Dimension sizes (allocate to
                                        ! allow the maximum number of
                                        ! dimensions).
.
.
CALL CDF_inquire_zvar (id, CDF_get_var_num(id,'HEAT_FLUX'), var_name,
1      data_type, num_elems, rec_vary, dim_varys,
2      num_dims, dim_sizes, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.36 CDF\_put\_zvar\_data

SUBROUTINE CDF\_put\_zvar\_data (

```

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

```

CDF\_put\_zvar\_data is used to write a single value for a zVariable. CDF\_hyper\_put\_zvar\_data may be used to write more than one zVariable values with a single call (see Section 6.3.34).

The arguments to CDF\_put\_zvar\_data are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to write. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
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 zVariables this argument is ignored (but must be present).
value	The value to write. This buffer must be large enough to hold the value. CDF_inquire_zvar would be used to determine the zVariable's data type and number of elements (of that data type) at each value. The value is written to the CDF.

**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.



status                    The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.36.1. Example(s)

The following example writes an entire record of data to zVariable “Temperature”. This zVariable is 3-dimensional with sizes [180,91,10]. 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                ! zVariable number.
INTEGER*4 rec_num              ! Record number.
INTEGER*4 d1, d2, d3          ! Dimension index values.
.
.
var_n = CDF_get_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
. filled tmp array
.
DO d1 = 1, 180
  indices(1) = d1
  DO d2 = 1, 91
    indices(2) = d2
    DO d3 = 1, 10
      indices(3) = d3
      CALL CDF_put_zvar_data (id, var_n, rec_num, indices, tmp(d1,d2,d3),
1                                                          status)
      IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    END DO
  END DO
END DO
.
.
```

### 6.3.37            CDF\_put\_zvar\_recorddata

```
SUBROUTINE CDF_put_zvar_recorddata (

INTEGER*4 id,                    ! in -- CDF identifier.
INTEGER*4 var_num,              ! in -- zVariable number.
```

```

INTEGER*4 rec_num,          ! in -- Record number.
<type> buffer,             ! in -- Record data buffer.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_put\_zvar\_recorddata writes an entire record at a given record number for the specified zVariable in a CDF. The buffer should be large enough to hold the entire data values for the variable. The written data values in the buffer are in the order that corresponds to the variable majority defined for the CDF.

The arguments to CDF\_put\_zvar\_recorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to write. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
rec_num	The record number of the zVariable to which to write.
buffer	The record buffer to hold the data values for an entire record.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.37.1. Example(s)

The following example writes an entire record (numbered 5) for zVariable “MY\_VAR”, a 2-dimensional variable (2 by 3) of CDF\_INT4 data type, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 status      ! Returned status code.
INTEGER*4 buffer(2,3) ! Record buffer.
.
. fill buffer array
CALL CDF_put_zvar_recorddata (id, CDF_get_var_num (id, 'MY_VAR'), 5,
1  buffer, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.38 CDF\_put\_zvar\_seqdata

SUBROUTINE CDF\_put\_zvar\_seqdata (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
<type> value,         ! in -- Data value.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_put\_zvar\_seqdata writes one data value at the current sequential value for the specified zVariable in a CDF. After the read, the current sequential value is automatically incremented to the next value. An error is returned if the current sequential value is past the last record of the zVariable. Use CDF\_get\_zvar\_seqpos and CDF\_set\_zvar\_seqpos subroutine calls to get and set the current sequential value (position) for the variable.

The arguments to CDF\_put\_zvar\_seqdata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
value	The data value.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.38.1. Example(s)

The following example writes two data values from the beginning of record (numbered 2) to a zVariable, a 2-dimensional CDF\_INT4 type variable, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 var_num      ! Variable number.
INTEGER*4 value1, value2 ! Variable data values.
INTEGER*4 rec_num      ! Record number.
INTEGER*4 indices(2)   ! Dimension indices.
.
.
rec_num = 2
indices(1) = 0
indices(2) = 0
CALL CDF_set_zvar_seqpos (id, var_num, rec_num, indices, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
value1 = 10
value2 = 20
CALL CDF_put_zvar_seqdata (id, var_num, value1, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
CALL CDF_put_zvar_seqdata (id, var_num, value2, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.39 CDF\_put\_zvars\_recorddata

```

SUBROUTINE CDF_put_zvars_recorddata(

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 num_var,     ! in -- Number of zVariables.
INTEGER*4 var_nums(*), ! in -- zVariable numbers.
INTEGER*4 rec_num,     ! in -- Record number.
<type> buffer,        ! in -- First variable buffer in a common block (<type> depends
                       !           on the data type of the zVariable).
INTEGER*4 status)      ! out -- Completion status.

```

CDF\_put\_zvars\_recorddata is used to write a full record data at a specific record number for a selected group of zVariables in a CDF. It expects that the data buffer for each zVariable is big enough to contain a full physical record data and properly put in a common block. No space is expected for each zVariable's non-variant dimensional elements. Record data from each buffer is written to its respective zVariable.

The arguments to CDF\_put\_zvars\_recorddata are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDFcreate, Cdf_open or a similar CDF creation or opening functionality from the Internal Interface.
num_vars	The number of the zVariables in the group involved this write operation.
var_nums	The numbers of the zVariables involved for which to write a whole record data.
rec_num	The record number at which to write the whole record data for the group of zVariables.
buffer	The first variable buffer to write in a common block. The number of buffers should match to the num_var argument. Each buffer should hold a full physical record data.

### 6.3.39.1. Example(s)

The following example will write an entire single record data for a group of zVariables. The zVariables involved in the write are **Time**, **Longitude**, **Delta**, **Temperature** and **NAME**. The record to write is **4**. Since **Temperature** is 0-dimensional with **CDF\_FLOAT** data type, a scalar variable of REAL\*4 is allocated. For **Longitude**, a 1-dimensional array of INTEGER\*2 (size [3]) is given for its dimension variance [**VARY**] and data type **CDF\_INT2**. Similar data variables are provided for **Longitude** and **Time**. They both are 2-dimensional array of INTEGER\*4 (sizes [3,2]) for their dimension variances [**VARY,VARY**] and data type either **CDF\_INT4** or **CDF\_UINT4**. For **NAME**, a 1-dimensional array of CHARACTER\*10 (size [2]) is allocated due to its [**VARY**] dimension variance and **CDF\_CHAR** data type with the number of element 10.

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 num_var      ! Number of zVariables.
INTEGER*4 var_nums(5)  ! zVariable numbers in CDF.
INTEGER*4 rec_num      ! Record number to write.
INTEGER*4 time(3,2)    ! Datatype: UINT4.
1          /10, 20,     ! Rec/dim variances: T/TT.
2          30, 40,
3          50, 60/
INTEGER*4 delta(3,2)   ! Datatype: INT4 .

```

```

1          /1, 2,          ! Rec/dim variances: T/TT.
2          5, 6,
3          9, 10/
INTEGER*2  longitude(3)   ! Datatype: INT2.
1          /10, 20, 30/   ! Rec/dim variances: T/T.
REAL*4     temperature    ! Datatype: FLOAT.
1          /1234.56/     ! Rec/dim variances: T/.
CHARACTER*10 name(2)      ! Datatype: CHAR/10.
1          /'ABCDEFGHIJ', ! Rec/dim variances: T/T.
2          '12345678'/

```

```
COMMON /BLK/delta, time, temperature, longitude, name
```

```

.
.
num_var = 5          ! Number of zVariables
rec_num = 4         ! Record number to write

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Delta', var_nums(1),
1          NULL_, status)          ! zVariable number
IF (var_nums(1) .LT. 1)          ! If less than one (1),
x CALL UserStatusHandler (var_nums(1)) ! then it is actually a
! warning/error code.

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Time', var_nums(2),
1          NULL_, status)
IF (var_nums(2) .LT. 1) CALL UserStatusHandler (var_nums(2))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Longitude', var_nums(3),
1          NULL_, status)
IF (var_nums(3) .LT. 1) CALL UserStatusHandler (var_nums(3))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'Temperature', var_nums(4),
1          NULL_, status)
IF (var_nums(4) .LT. 1) CALL UserStatusHandler (var_nums(4))

status = CDF_LIB (GET_, zVAR_NUMBER_, 'NAME', var_nums(5),
1          NULL_, status)
IF (var_nums(5) .LT. 1) CALL UserStatusHandler (var_nums(5))

CALL CDF_put_zvars_reorddata (id, num_var, var_nums, rec_num,
1          time, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

Note that the ordering of the variable data buffer in the COMMON block BLK is very important. Always arrange data buffer in the order in such way that the variables with the bigger data types come in front of the variables with the smaller data types. They should be in this ordering: 8-byte, 4-byte, 2-byte, and 1-byte. Unexpected results may return if such ordering is not followed. This function can be a replacement for the similar functionality provided from the Internal Interface as <PUT\_, zVARS\_RECADATA\_>.

## 6.3.40 CDF\_rename\_zvar

SUBROUTINE CDF\_rename\_zvar (

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

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

The arguments to CDF\_rename\_zvar are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to rename. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
var_name	The new zVariable name. This may be at most CDF_VAR_NAME_LEN256 characters. Variable names are case-sensitive.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.40.1. Example(s)

In the following example the zVariable named TEMPERATURE is renamed to TMP (if it exists). Note that if CDF\_get\_var\_num returns a value less than one (1) then that value is not a zVariable 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     ! zVariable number.
.
.
var_num = CDF_get_var_num (id, 'TEMPERATURE')
IF (var_num .LT. 1) THEN
  IF (var_num .NE. NO_SUCH_VAR) CALL UserStatusHandler (var_num)
ELSE
  CALL CDF_rename_zvar (id, var_num, 'TMP', status)
  IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
END IF
.
.
```

### 6.3.41 CDF\_set\_zvar\_allocblockrecs

SUBROUTINE CDF\_set\_zvar\_allocblockrecs (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,      ! in -- zVariable number.
INTEGER*4 first_rec,    ! in -- First record number to allocate.
INTEGER*4 last_rec,     ! in -- Last record number to allocate.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_zvar\_allocblockrecs specifies a range records to allocate for the specified zVariable in a CDF. This operation is only applicable to uncompressed variables in single-file CDFs. Refer to the CDF User's Guide for the description of allocations of variable records.

The arguments to CDF\_set\_zvar\_allocblockrecs are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
first_rec	The first record number to allocate.
last_rec	The last record number to allocate.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.41.1. Example(s)

The following example allocates 100 records, from record number 21 to 120, for zVariable "MY\_VAR" in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 first_rec    ! Starting record number to allocate.
INTEGER*4 last_rec     ! Ending record number to allocate.
INTEGER*4 status       ! Returned status code.

.
.
first_rec = 21
last_rec = 120
CALL CDF_set_zvar_allocblockrecs (id, CDF_get_var_num(id, 'MY_VAR'),
1                                first_rec, last_rec, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.3.42 CDF\_set\_zvar\_allocrecs

SUBROUTINE CDF\_set\_zvar\_allocrecs (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_recs,   ! in -- Number of allocated records.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_set\_zvar\_allocrecs specifies the number of records allocated for the specified zVariable in a CDF. The records are allocated beginning at record number one (1). This operation is only applicable to uncompressed variables in single-file CDFs. Refer to the CDF User's Guide for the description of allocating variable records in a single-file CDF.

The arguments to CDF\_set\_zvar\_allocrecs are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
num_recs	The number of records allocated for the variable.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.42.1. Example(s)

The following example allocates 100 records (record number 1 to 100) for zVariable "MY\_VAR" in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 num_recs    ! Number of allocated records.
INTEGER*4 status      ! Returned status code.

.
.
num_recs = 100
CALL CDF_set_zvar_allocrecs (id, CDF_get_var_num(id, 'MY_VAR'),
1 num_recs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.3.43 CDF\_set\_zvar\_blockingfactor

SUBROUTINE CDF\_set\_zvar\_blockingfactor (



```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 bf,          ! in -- Variable blocking factor.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_blockingfactor respecifies the blocking factor for the specified zVariable in a CDF. Refer to the CDF User's Guide for the description of a variable's blocking factor.

The arguments to CDF\_set\_zvar\_blockingfactor are defined as follows:

```

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

var_num      The zVariable number.

bf           The blocking factor of the variable.

status       The completion status code. Chapter 8 explains how to interpret status codes.

```

### 6.3.43.1. Example(s)

The following example sets the blocking factor to 100 records for zVariable "MY\_VAR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 bf           ! Blocking factor.
INTEGER*4 status       ! Returned status code.
.
.
bf = 100
CALL CDF_set_zvar_blockingfactor (id, CDF_get_var_num(id, 'MY_VAR'),
1                                bf, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.44 CDF\_set\_zvar\_cachesize

SUBROUTINE CDF\_set\_zvar\_cachesize (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 num_buffers, ! in -- Number of cache buffers.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_cachesize specifies the number of cache buffers being for the specified zVariable in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User's Guide for the description about caching scheme used by the CDF library.

The arguments to CDF\_set\_zvar\_cachesize are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
num_buffers	The number of cache buffers.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.44.1. Example(s)

The following example sets the number of cache buffers to 10 to be used for zVariable "MY\_VAR" in a multi-file CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 num_buffers! Number of cache buffers.
INTEGER*4 status      ! Returned status code.
.
.
num_buffers = 10
CALL CDF_set_zvar_cachesize (id, CDF_get_var_num(id, 'MY_VAR'),
1                             num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.45 CDF\_set\_zvar\_compression

SUBROUTINE CDF\_set\_zvar\_compression (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 var_num,    ! in -- zVariable number.
INTEGER*4 compress_type, ! in -- Compression type.
INTEGER*4 compress_parms, ! in -- Compression parameters.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_set\_zvar\_compression respecifies the compression type/parameters of the specified zVariable in a CDF. Refer to Section 4.10 for the description of the CDF supported compression types/parameters.

The arguments to CDF\_set\_zvar\_compression are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
compress_type	The compression type.
compress_parms	The compression parameters.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.45.1. Example(s)

The following example uses GZIP.9 compression for zVariable “MY\_VAR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 ctype             ! Compression type.
INTEGER*4 cparms(CDF_MAX_DIMS) ! Compression parameters.
INTEGER*4 status           ! Returned status code.

.
.
ctype = GZIP_COMPRESSION
cparms(1) = 9
CALL CDF_set_zvar_compression (id, CDF_get_var_num(id, 'MY_VAR'),
1                               ctype, cparms, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.46 CDF\_set\_zvar\_dataspec

```

SUBROUTINE CDF_set_zvar_dataspec (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 data_type,        ! in -- Data type.
INTEGER*4 status)           ! out -- Completion status

```

CDF\_set\_zvar\_dataspec is used to respecify the data specification (data type and number of elements) of the specified zVariable in a CDF. A zVariable's data specification may not be changed if the new data specification is not equivalent to the old one and any values, including pad value, have been written. Data specifications are considered equivalent if the data types are equivalent and the number of elements are the same. Refer to Section 4.5 for the description of the CDF data types.

The arguments to CDF\_set\_zvar\_dataspec are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to set. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
data_type	The data type of the variable data.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.46.1. Example(s)

The following example respecifies the data type of zVariable "Temperature" to CDF\_UINT2, from its original CDF\_INT2, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 data_type    ! Data type.
.
.
data_type = CDF_UINT2
CALL CDF_set_zvar_dataspec (id, CDF_get_var_num (id, 'Temperature'),
1                          data_type, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.47 CDF\_set\_zvar\_dimvariances

SUBROUTINE CDF\_set\_zvar\_dimvariances (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 dim_varys(*), ! in -- Dimension variances.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_dimvariances respecifies the dimension variances of the specified zVariable in a CDF. For 0-dimensional zVariable, this operation is not applicable. Refer to Section 4.9 for the description of the CDF variable's dimension variances.

The arguments to CDF\_set\_zvar\_dimvariances are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to set. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
dim_varys	The dimension variances.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.47.1. Example(s)

The following example sets the dimension variances to VARY and VARY for zVariable "Temperature", a 2-dimensional variable, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 dim_varys(CDF_MAX_DIMS) ! Dimension variances.
.
.
dim_varys(1) = VARY
dim_varys(2) = VARY
CALL CDF_set_zvar_dimvariances (id, CDF_get_var_num (id, 'Temperature'),
1                                dim_varys, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.48 CDF\_set\_zvar\_initialrecs

SUBROUTINE CDF\_set\_zvar\_initialrecs (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 num_recs,        ! in -- Number of written records.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_set\_zvar\_initialrecs specifies the number of records initially written for the specified zVariable in a CDF. The records are written beginning at record number one (1). This may be specified only once per variable and before any other records have been written to that variable. If a pad value has not yet been specified, the default value is used. If a

pad value has been explicitly specified, that value is written to the records. Refer to the CDF User's Guide for the description of initial variable records.

The arguments to `CDF_set_zvar_initialrecs` are defined as follows:

<code>id</code>	The identifier of the CDF. This identifier must have been initialized by a call to <code>CDF_create_cdf</code> or <code>CDF_open_cdf</code> .
<code>var_num</code>	The zVariable number.
<code>num_recs</code>	The number of records to be written for the variable.
<code>status</code>	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.48.1. Example(s)

The following example writes initially 100 records (record number 1 to 100) for zVariable "MY\_VAR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 num_recs    ! Number of initially written records.
INTEGER*4 status      ! Returned status code.

.
.
num_recs = 100
CALL CDF_set_zvar_initialrecs (id, CDF_get_var_num(id, 'MY_VAR'),
1                               num_recs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.49 CDF\_set\_zvar\_padvalue

SUBROUTINE `CDF_set_zvar_padvalue` (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
<type> pad_value,      ! in -- Pad value.
INTEGER*4 status)      ! out -- Completion status

```

`CDF_set_zvar_padvalue` respecifies the pad value for the specified zVariable in a CDF. A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values.

The arguments to `CDF_set_zvar_padvalue` are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to set. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
pad_value	The pad value.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.49.1. Example(s)

The following example sets the pad value to -999 for zVariable “MY\_VAR”, a CDF\_INT4 type variable, in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 pad_value    ! Pad value.
.
.
pad_value = -999
CALL CDF_set_zvar_padvalue (id, CDF_get_var_num (id, 'MY_VAR'),
1  pad_value, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.50 CDF\_set\_zvar\_recvariance

SUBROUTINE CDF\_set\_zvar\_recvariance (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 rec_vary,    ! in -- Record variance.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_recvariance respecifies the record variance for the specified zVariable in a CDF. Refer to Section 4.9 for the description of the CDF variable’s record variance.

The arguments to CDF\_set\_zvar\_recvariance are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable to which to set. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).

rec_vary	The record variance.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.50.1. Example(s)

The following example sets the record variance to VARY for zVariable “Temperature” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 rec_vary     ! Record variance.
.
.
rec_vary = VARY
CALL CDF_set_zvar_recvariance (id, CDF_get_var_num (id, 'Temperature'),
1 rec_vary, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.3.51 CDF\_set\_zvar\_reservepercent

SUBROUTINE CDF\_set\_zvar\_reservepercent (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 res_percent, ! in -- Reserve percentage.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_reservepercent respecifies the reserve percentage being used for the specified zVariable in a CDF. This operation only applies to compressed zVariables. Refer to the CDF User’s Guide for the description of the reserve scheme used by the CDF library.

The arguments to CDF\_set\_zvar\_reservepercent are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The number of the zVariable from which to read. This number may be determined with a call to CDF_get_var_num (see Section 6.3.8).
res_percent	The reserve percentage.
status	The completion status code. Chapter 8 explains how to interpret status codes.



### 6.3.51.1. Example(s)

The following example sets the reserve percentage to 15 for the compressed zVariable “Temperature” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 status       ! Returned status code.
INTEGER*4 res_percent  ! Reserve percentage.
.
.
res_percent = 15
CALL CDF_set_zvar_reservepercent (id, CDF_get_var_num (id, 'Temperature'),
1 res_percent, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.3.52 CDF\_set\_zvars\_cachesize

SUBROUTINE CDF\_set\_zvars\_cachesize (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 num_buffers, ! in -- zVariables’s number of cache buffers.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_zvars\_cachesize respecifies the number of cache buffers being used for all zVariables in a CDF. This operation is not applicable to a single-file CDF. Refer to the CDF User’s Guide for the description about caching scheme used by the CDF library.

The arguments to CDF\_set\_zvars\_cachesize are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

num\_buffers        The number of cache buffers.

status        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.3.52.1. Example(s)

The following example sets the number of cache buffers to 10 for all zVariables in a CDF.

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

```

.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 num_buffers ! Number of cache buffers.
INTEGER*4 status      ! Returned status code.

.
.
num_buffers = 10
CALL CDF_set_zvars_cachesize (id, num_buffers, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.53 CDF\_set\_zvar\_seqpos

SUBROUTINE CDF\_set\_zvar\_seqpos (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 var_num,     ! in -- zVariable number.
INTEGER*4 rec_num,     ! in -- Record number.
INTEGER*4 indices(*), ! in -- Indices in a record.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_set\_zvar\_seqpos specifies the current sequential value (position) for sequential access for the specified zVariable in a CDF. Note that a current sequential value is maintained for each zVariable individually. Use CDF\_get\_zvar\_seqdata subroutine to get the data value.

The arguments to CDF\_set\_zvar\_seqpos are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	The zVariable number.
rec_num	The record number.
indices	The dimension indices. Each element of indices receives the corresponding dimension index. For 0-dimensional zVariable, this argument is ignored, but must be presented.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.53.1. Example(s)

The following example sets the current sequential value to the first value element in record number 2 for zVariable “MY\_VAR”, a 2-dimensional variable, in a CDF.

```

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

```

```

.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.

INTEGER*4 rec_num          ! Record number.
INTEGER*4 indices(2)      ! Dimension indices.
.
.
rec_num = 2
indices(1) = 0
indices(2) = 0
CALL CDF_set_zvar_seqpos (id, CDF_get_var_num(id, 'MY_VAR'), rec_num,
1                          indices, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.3.54 CDF\_set\_zvar\_sparserecords

SUBROUTINE CDF\_set\_zvar\_sparserecords (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 var_num,          ! in -- zVariable number.
INTEGER*4 srecords_type,    ! in -- Sparse records type.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_set\_zvar\_sparserecords respecifies the sparse records type for the specified zVariable in a CDF. Refer to Section 4.11 for the description of the sparse records.

The arguments to CDF\_set\_zvar\_sparserecords are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
var_num	zVariable number.
srecords_type	Sparse records type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.3.54.1. Example(s)

The following example sets the sparse records type to PAD\_SPARSERECORDS from its original type for zVariable “MY\_VAR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.

```

```

INTEGER*4 srecords_type           ! Sparse records type.
INTEGER*4 num_dims                ! Dimension sizes.
.
.
srecords_type = PAD_SPARSERECORDS
CALL CDF_set_zvar_sparserecords (id, CDF_get_var_num(id, "MY_VAR"),
1                               srecords_type, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4 Attributes/Entries

This section provides the functions related to attributes or entries in an attribute. An attribute is identified by its name or an number in the CDF. To operate an attribute or entry, the CDF it resides in must be open.

### 6.4.1 CDF\_confirm\_attr\_existence

```

INTEGER*4 FUNCTION CDF_confirm_attr_existence (

```

```

INTEGER*4 id,           ! in -- CDF identifier.
CHARACTER attr_name*(*) ! in -- Attribute name.

```

CDF\_confirm\_attr\_existence confirms whether the specified name is an existing attribute in a CDF. It returns CDF\_OK if the attribute exists.

The arguments to CDF\_confirm\_attr\_existence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_name	Checks if an attribute with the given name exists in the CDF.

#### 6.4.1.1. Example(s)

The following example checks whether the attribute by the name of "ATTR\_NAME1" is in a CDF.

```

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

```

```

.
status = CDF_confirm_attr_existence (id, "ATTR_NAME1", status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.2 CDF\_confirm\_gentry\_existence

```

INTEGER*4 FUNCTION CDF_confirm_gentry_existence (

```

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Global attribute identifier.
INTEGER*4 entry_num)   ! in -- gEntry number.

```

CDF\_confirm\_gentry\_existence confirms the existence of the specified gEntry in an (global) attribute of a CDF. If the gEntry does not exist, NO\_SUCH\_ENTRY will be returned.

The arguments to CDF\_confirm\_gentry\_existence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The (global) attribute number.
entry_num	The gEntry number.

### 6.4.2.1. Example(s)

The following example will check the existence of gEntry numbered 1 for attribute "MY\_ATTR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 attr_num     ! Attribute number.
INTEGER*4 status       ! Returned status code.
.
.
attr_num = CDF_get_attr_num(id, 'MY_ATTR')
IF (attr_num .LT. 1) CALL UserQuit(....)
status = CDF_confirm_gentry_existence (id, attr_num, 1)
IF (status .EQ. NO_SUCH_ENTRY) CALL UserStatusHandler (status)
.
.

```

### 6.4.3 CDF\_confirm\_reentry\_existence

```
INTEGER*4 FUNCTION CDF_confirm_reentry_existence (  
  
INTEGER*4 id,           ! in -- CDF identifier.  
INTEGER*4 attr_num,    ! in -- Variable attribute identifier.  
INTEGER*4 entry_num)   ! in -- rEntry number.
```

CDF\_confirm\_reentry\_existence confirms the existence of the specified rEntry, corresponding to an rVariable, in an (variable) attribute of a CDF. If the rEntry does not exist, NO\_SUCH\_ENTRY will be returned.

The arguments to CDF\_confirm\_reentry\_existence are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The (variable) attribute number.
entry_num	The rEntry number.

#### 6.4.3.1. Example(s)

The following example will check the existence of the rEntry corresponding to rVariable “MY\_VAR” for attribute “MY\_ATTR” in a CDF.

```
.  
.   
INCLUDE '<path>cdf.inc'  
.   
.   
INTEGER*4 id           ! CDF identifier.  
INTEGER*4 attr_num     ! Attribute number.  
INTEGER*4 entry_num    ! rEntry number.  
INTEGER*4 status       ! Returned status code.  
.   
.   
attr_num = CDF_get_attr_num(id, 'MY_ATTR')  
IF (attr_num .LT. 1) CALL UserQuit(...)  
entry_num = CDF_get_var_num(id, 'MY_VAR')  
IF (entry_num .LT. 1) CALL UserQuit(...)  
status = CDF_confirm_reentry_existence (id, attr_num, entry_num, status)  
IF (status .EQ. NO_SUCH_ENTRY) CALL UserStatusHandler (status)  
.   
.
```

### 6.4.4 CDF\_confirm\_zentry\_existence

```
INTEGER*4 FUNCTION CDF_confirm_zentry_existence (  
  

```

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Variable attribute identifier.
INTEGER*4 entry_num)   ! in -- zEntry number.

```

CDF\_confirm\_zentry\_existence confirms the existence of the specified zEntry, corresponding to a zVariable, in an (variable) attribute of a CDF. If the zEntry does not exist, NO\_SUCH\_ENTRY will be returned.

The arguments to CDF\_confirm\_zentry\_existence are defined as follows:

```

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

attr_num     The (variable) attribute number.

entry_num    The zEntry number.

```

#### 6.4.4.1. Example(s)

The following example will check the existence of the zEntry corresponding to zVariable “MY\_VAR” for attribute “MY\_ATTR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id           ! CDF identifier.
INTEGER*4 attr_num     ! Attribute number.
INTEGER*4 entry_num    ! zEntry number.
INTEGER*4 status       ! Returned status code.
.
.
attr_num = CDF_get_attr_num(id, 'MY_ATTR')
IF (attr_num .LT. 1) CALL UserQuit(...)
entry_num = CDF_get_var_num(id, 'MY_VAR')
IF (entry_num .LT. 1) CALL UserQuit(...)
Status = CDF_confirm_zentry_existence (id, attr_num, entry_num, status)
IF (status .EQ. NO_SUCH_ENTRY) CALL UserStatusHandler (status)
.
.

```

#### 6.4.5 CDF\_create\_attr

SUBROUTINE CDF\_create\_attr (

```

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

The arguments to CDF\_create\_attr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_name	The name of the attribute to create. This may be at most CDF_ATTR_NAME_LEN256 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 subroutine calls when referring to this attribute. An existing attribute's number may be determined with the CDF_get_attr_num function.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.5.1. Example(s)

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

```

.
.
INCLUDE '<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_create_attr (id, 'TITLE', TITLE_attr_scope, TITLE_attr_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
CALL CDF_create_attr (id, UNITS_attr_name, VARIABLE_SCOPE, UNITS_attr_num,
1                      status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```



## 6.4.6 CDF\_delete\_attr

SUBROUTINE CDF\_delete\_attr (

```
INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_delete\_attr deletes the specified attribute from a CDF.

The arguments to CDF\_delete\_attr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number to be deleted.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.6.1. Example(s)

The following example will delete attribute “MY\_ATTR” in a CDF.

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

## 6.4.7 CDF\_delete\_attr\_gentry

SUBROUTINE CDF\_delete\_attr\_gentry (

```
INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Global attribute number.
INTEGER*4 entry_num, ! in -- gEntry number.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_delete\_attr\_gentry deletes the specified gEntry in an (global) attribute from a CDF

The arguments to `CDF_delete_attr_gentry` are defined as follows:

- `id`            The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create_cdf` or `CDF_open_cdf`.
- `attr_num`    The global attribute number.
- `entry_num`   The gEntry number to be deleted.
- `status`       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.7.1. Example(s)

The following example will delete gEntry numbered 2 from the global attribute “MY\_ATTR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 status                        ! Returned status code.
.
.
CALL CDF_delete_attr_gentry (id, CDF_get_attr_num(id, 'MY_ATTR'), 2, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.8 CDF\_delete\_attr\_rentry

SUBROUTINE `CDF_delete_attr_rentry` (

```
INTEGER*4 id,            ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Variable attribute number.
INTEGER*4 entry_num, ! in -- rEntry number.
INTEGER*4 status)      ! out -- Completion status
```

`CDF_delete_attr_rentry` deletes the specified rEntry, corresponding to an rVariable, in an (variable) attribute from a CDF

The arguments to `CDF_delete_attr_rentry` are defined as follows:

- `id`            The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create_cdf` or `CDF_open_cdf`.
- `attr_num`    The variable attribute number.
- `entry_num`   The rEntry number to be deleted.



### 6.4.9.1. Example(s)

The following example will delete the entry for zVariable “MY\_VAR” from the variable attribute “MY\_ATTR” in a CDF.

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

.
.
entry_num = CDF_get_var_num(id, "MY_VAR")
IF (entry_num .LT. 1) CALL UserQuit(.....)
CALL CDF_delete_attr_zentry (id, CDF_get_attr_num(id, 'MY_ATTR'), entry_num,
1                             status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.10 CDF\_get\_attr\_gentry

SUBROUTINE CDF\_get\_attr\_gentry (

```
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,          ! in -- Global 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\_get\_attr\_gentry is used to read a global attribute’s entry from a CDF. In most cases it will be necessary to call CDF\_inquire\_attr\_gentry before calling CDF\_get\_attr\_gentry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDF\_get\_attr\_gentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The global attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. This is the gEntry number and has meaning only to the application.
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 memory at address 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 8 explains how to interpret status codes.

### 6.4.10.1. Example(s)

The following example displays the value of the global attribute UNITS for the gEntry numbered 2 (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 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_get_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.

CALL CDF_inquire_attr_gentry (id, attr_n, 2, data_type, num_elems,
1                                                            status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

IF (data_type .EQ. CDF_CHAR) THEN
    CALL CDF_get_attr_gentry (id, attr_n, 2, buffer, status)
    IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
    WRITE (6,10) buffer(1:num_elems)
10     FORMAT (' ',A)
END IF
.
.

```

## 6.4.11 CDF\_get\_attr\_gentry\_datatype

SUBROUTINE CDF\_get\_attr\_gentry\_datatype (

```

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 of the entry.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_attr\_gentry\_datatype acquires the data type of the specified gEntry from an (global) attribute in a CDF

The arguments to CDF\_get\_attr\_gentry\_datatype are defined as follows:

```

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

attr_num    The attribute number.

entry_num   The gEntry number.

data_type   The data type of the entry.

status      The completion status code. Chapter 8 explains how to interpret status codes.

```

### 6.4.11.1. Example(s)

The following example acquires the data type for gEntry numbered 5 in the global attribute “MY\_ATTR” in a CDF.

```

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

.
.
CALL CDF_get_attr_gentry_datatype (id, CDF_get_attr_num(id, 'MY_ATTR'), 5,
1 data_type, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.12 CDF\_get\_attr\_gentry\_numelems

SUBROUTINE CDF\_get\_attr\_gentry\_numelems (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 entry_num,  ! in -- Entry number.
INTEGER*4 num_elems, ! out -- Number of elements of the entry.

```

```
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_attr\_gentry\_numelems acquires the number of elements of the specified gEntry from an (global) attribute in a CDF

The arguments to CDF\_get\_attr\_gentry\_numelems are defined as follows:

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

attr_num    The attribute number.

entry_num   The gEntry number.

num_elems  The number of elements of the gEntry.

status      The completion status code. Chapter 8 explains how to interpret status codes.
```

### 6.4.12.1. Example(s)

The following example acquires the number of elements for gEntry numbered 5 in the global attribute “MY\_ATTR” in a CDF.

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

.
.
CALL CDF_get_attr_gentry_numelems (id, CDF_get_attr_num(id, 'MY_ATTR'), 5,
1                                num_elems, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.13 CDF\_get\_attr\_max\_gentry

```
SUBROUTINE CDF_get_attr_max_gentry (

INTEGER*4 id,              ! in -- CDF identifier.
INTEGER*4 attr_num,       ! in -- Attribute number.
INTEGER*4 entry_num,      ! out -- Entry number.
INTEGER*4 status)         ! out -- Completion status
```

CDF\_get\_attr\_max\_gentry acquires the last gEntry number from an (global) attribute in a CDF.

The arguments to `CDF_get_attr_max_gentry` are defined as follows:

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

`attr_num`    The attribute number.

`entry_num`   The last `gEntry` number.

`status`       The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.13.1. Example(s)

The following example acquires the last `gEntry` number from the global attribute “MY\_ATTR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                            ! CDF identifier.
INTEGER*4 entry_num                    ! The last gEntry number.
INTEGER*4 status                       ! Returned status code.

.
.
CALL CDF_get_attr_max_gentry (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                    entry_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.14        `CDF_get_attr_max_rentry`

SUBROUTINE `CDF_get_attr_max_rentry` (

`INTEGER*4 id,                    ! in -- CDF identifier.`  
`INTEGER*4 attr_num,            ! in -- Attribute number.`  
`INTEGER*4 entry_num,          ! out -- Entry number.`  
`INTEGER*4 status)              ! out -- Completion status`

`CDF_get_attr_max_rentry` acquires the last `rEntry` number from an (variable) attribute in a CDF.

The arguments to `CDF_get_attr_max_rentry` are defined as follows:

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

`attr_num`    The attribute number.



entry\_num The last rEntry number.

status The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.14.1. Example(s)

The following example acquires the last rEntry number from the variable attribute “MY\_ATTR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 entry_num        ! The last rEntry number.
INTEGER*4 status           ! Returned status code.
.
.
CALL CDF_get_attr_max_zentry (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                               entry_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.15 CDF\_get\_attr\_max\_zentry

SUBROUTINE CDF\_get\_attr\_max\_zentry (

```
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Attribute number.
INTEGER*4 entry_num,  ! out -- Entry number.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_get\_attr\_max\_zentry acquires the last zEntry number from an (variable) attribute in a CDF.

The arguments to CDF\_get\_attr\_max\_zentry are defined as follows:

id The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

attr\_num The attribute number.

entry\_num The last zEntry number.

status The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.15.1. Example(s)

The following example acquires the last zEntry number from the variable attribute “MY\_ATTR” in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 entry_num         ! The last zEntry number.
INTEGER*4 status            ! Returned status code.
.
.
CALL CDF_get_attr_max_gentry (id, CDF_get_attr_num(id, 'MY_ATTR'),
1 entry_num, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.16 CDF\_get\_attr\_name

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

CDF\_get\_attr\_name acquires the name of the specified attribute (by its number) in a CDF.

The arguments to CDF\_get\_attr\_name are defined as follows:

id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

attr\_num     The attribute number.

attr\_name    The attribute name.

status       The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.4.16.1. Example(s)

The following example acquires the name of the attribute number 2 in a CDF.

```
.
.
```

```

INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                               ! CDF identifier.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN256 ) ! The last rEntry number.
INTEGER*4 status                           ! Returned status code.

.
.
CALL CDF_get_attr_name (id, 2, attr_name, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.17 CDF\_get\_attr\_num

```

INTEGER*4 FUNCTION CDF_get_attr_num (
INTEGER*4 id,           ! in -- CDF identifier.
CHARACTER attr_name*(*) ! in -- Attribute name.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_attr\_num is used to determine the attribute number associated with a given attribute name. If the attribute is found, CDF\_get\_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 (of type INTEGER\*4) is returned. Error codes are less than zero (0).

The arguments to CDF\_get\_attr\_num are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_name	The name of the attribute for which to search. This may be at most CDF_ATTR_NAME_LEN256 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.)

### 6.4.17.1. 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. Note that if the attribute pressure did not exist in the CDF, the call to CDF\_get\_attr\_num would have returned an error code. Passing that error code to CDF\_rename\_attr as an attribute number would have resulted in CDF\_rename\_attr also returning an error code. CDF\_rename\_attr is described in Section 6.4.38.

```

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

```

```

INTEGER*4 id          ! CDF identifier.
INTEGER*4 status      ! Returned status code.
.
.
CALL CDF_rename_attr (id, CDF_get_attr_num(id, 'pressure'), 'PRESSURE',
1                      status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.18 CDF\_get\_attr\_num\_gentries

SUBROUTINE CDF\_get\_attr\_num\_gentries (

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 entries,    ! out -- Total entries.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_attr\_num\_gentries acquires the total number of entries (gEntries) in the specified (global) attribute of a CDF.

The arguments to CDF\_get\_attr\_num\_gentries are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
entries	Total gEntries.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.18.1. Example(s)

The following example acquires the total number of entries (gEntries) in the global attribute "MY\_ATTR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 entries     ! Total entries.
INTEGER*4 status      ! Returned status code.
.
.
CALL CDF_get_attr_num_gentries (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                      entries, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)

```

.  
.

## 6.4.19 CDF\_get\_attr\_num\_retries

SUBROUTINE CDF\_get\_attr\_num\_retries (

```
INTEGER*4 id,          ! in -- CDF identifier.  
INTEGER*4 attr_num,   ! in -- Attribute number.  
INTEGER*4 entries,    ! out -- Total entries.  
INTEGER*4 status)     ! out -- Completion status
```

CDF\_get\_attr\_num\_retries acquires the total number of entries for the rVariables (rEntries) in the specified (variable) attribute of a CDF.

The arguments to CDF\_get\_attr\_num\_retries are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
entries	Total rEntries.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.19.1. Example(s)

The following example acquires the total number of entries (rEntries) in the variable attribute “MY\_ATTR” in a CDF.

```
.  
.   
INCLUDE '<path>cdf.inc'  
.   
.   
INTEGER*4 id                ! CDF identifier.  
INTEGER*4 entries           ! Total entries.  
INTEGER*4 status            ! Returned status code.  
  
.   
.   
CALL CDF_get_attr_num_retries (id, CDF_get_attr_num(id, 'MY_ATTR'),  
1  entries, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.   
.
```

## 6.4.20 CDF\_get\_attr\_num\_zentries

SUBROUTINE CDF\_get\_attr\_num\_zentries (

```
INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 entries,    ! out -- Total entries.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_get\_attr\_num\_zentries acquires the total number of entries for the zVariable (zEntries) in the specified (variable) attribute of a CDF.

The arguments to CDF\_get\_attr\_num\_zentries are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
entries	Total zEntries.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.20.1. Example(s)

The following example acquires the total number of entries (zEntries) in the variable attribute “MY\_ATTR” in a CDF.

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

.
.
CALL CDF_get_attr_num_zentries (id, CDF_get_attr_num(id, 'MY_ATTR'),
1  entries, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.21 CDF\_get\_attr\_rentry

SUBROUTINE CDF\_get\_attr\_rentry (

```
INTEGER*4 id,          ! in -- CDF identifier.
```

```

INTEGER*4 attr_num,      ! in -- Variable 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\_get\_attr\_rentry is used to read a variable attribute's entry corresponding to an rVariable (rEntry) from a CDF. In most cases it will be necessary to call CDF\_inquire\_attr\_rentry before calling CDF\_get\_attr\_rentry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDF\_get\_attr\_rentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The variable attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. 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 memory at address value.
	<b>WARNING:</b> 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 8 explains how to interpret status codes.

### 6.4.21.1. Example(s)

The following example displays the value of the variable attribute UNITS 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_get_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_get_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_inquire_attr_reentry (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_get_attr_reentry (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
.
.

```

## 6.4.22 CDF\_get\_attr\_reentry\_datatype

SUBROUTINE CDF\_get\_attr\_reentry\_datatype (

```

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 of the entry.
INTEGER*4 status) ! out -- Completion status

```

CDF\_get\_attr\_reentry\_datatype acquires the data type of the specified rEntry, corresponding to an rVariable, from an (variable) attribute in a CDF.

The arguments to CDF\_get\_attr\_reentry\_datatype are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
entry_num	The rEntry number.
data_type	The data type of the entry.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.22.1. Example(s)



The following example acquires the data type for rEntry, corresponding to rVariable “MY\_VAR” in the variable attribute “MY\_ATTR” in a CDF.

```

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

.
.
CALL CDF_get_attr_rentry_datatype (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                CDF_get_var_num(id, "MY_VAR"), data_type,
2                                status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.4.23 CDF\_get\_attr\_rentry\_numelems

SUBROUTINE CDF\_get\_attr\_rentry\_numelems (

```

INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Attribute number.
INTEGER*4 entry_num,   ! in -- Entry number.
INTEGER*4 num_elems,  ! out -- Number of elements of the entry.
INTEGER*4 status)      ! out -- Completion status

```

CDF\_get\_attr\_rentry\_numelems acquires the number of elements of the specified rEntry, corresponding to an rVariable, from an (variable) attribute in a CDF.

The arguments to CDF\_get\_attr\_rentry\_numelems are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- attr\_num     The attribute number.
- entry\_num    The rEntry number.
- num\_elems    The number of elements of the rEntry.
- status       The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.4.23.1. Example(s)

The following example acquires the number of elements for rEntry, corresponding to rVariable “MY\_VAR”, in the variable attribute “MY\_ATTR” in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 num_elements      ! Number of elements.
INTEGER*4 status            ! Returned status code.

.
.
CALL CDF_get_attr_reentry_numelems (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                CDF_get_var_num(id, "MY_VAR"), num_elems,
2                                status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.24 CDF\_get\_attr\_scope

```
SUBROUTINE CDF_get_attr_scope (
```

```

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 scope,      ! out -- Attribute scope.
INTEGER*4 status)     ! out -- Completion status

```

CDF\_get\_attr\_scope acquires the scope, either GLOBAL\_SCOPE or VARIABLE\_SCOPE, of the specified attribute in a CDF.

The arguments to CDF\_get\_attr\_scope are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
scope	The attribute scope.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.24.1. Example(s)

The following example acquires the scope for the attribute “MY\_ATTR” in a CDF.

```

.
.

```

```

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

.
.
CALL CDF_get_attr_scope (id, CDF_get_attr_num(id, 'MY_ATTR'), scope,
1  status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.25 CDF\_get\_attr\_zentry

SUBROUTINE CDF\_get\_attr\_zentry (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,          ! in -- variable 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\_get\_attr\_zentry is used to read a variable attribute's entry, corresponding to a zVariable, (zEntry) in a CDF. In most cases it will be necessary to call CDF\_inquire\_attr\_zentry before calling CDF\_get\_attr\_zentry in order to determine the data type and number of elements (of that data type) for the entry.

The arguments to CDF\_get\_attr\_zentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The variable attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. This is the number of the associated zVariable (the zVariable being described in some way by the zEntry).
value	The value read. This buffer must be large enough to hold the value. The subroutine CDF_inquire_attr_zentry would be used to determine the entry data type and number of elements (of that data type). The value is read from the CDF and placed into memory at address value.
	<b>WARNING:</b> 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 8 explains how to interpret status codes.

### 6.4.25.1. Example(s)

The following example displays the value of the UNITS attribute for the zEntry corresponding to the PRES\_LVL zVariable (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_get_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_get_var_num (id, 'PRES_LVL')         ! The zEntry number is
                                                    ! the zVariable number.

IF (entryN .LT. 0) CALL UserStatusHandler (entryN) ! If less than one (1),
                                                    ! then it must be a
                                                    ! warning/error code.

CALL CDF_inquire_attr_zentry (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_get_attr_zentry (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
.
.
```

### 6.4.26 CDF\_get\_attr\_zentry\_datatype

```
SUBROUTINE CDF_get_attr_zentry_datatype (
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 of the entry.
INTEGER*4 status)    ! out -- Completion status

```

CDF\_get\_attr\_zentry\_datatype acquires the data type of the specified zEntry, corresponding to a zVariable, from an (variable) attribute in a CDF.

The arguments to CDF\_get\_attr\_zentry\_datatype are defined as follows:

```

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

attr_num    The attribute number.

entry_num   The zEntry number.

data_type   The data type of the entry.

status      The completion status code. Chapter 8 explains how to interpret status codes.

```

### 6.4.26.1. Example(s)

The following example acquires the data type for zEntry, corresponding to zVariable “MY\_VAR” in the variable attribute “MY\_ATTR” in a CDF.

```

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

.
.
CALL CDF_get_attr_zentry_datatype (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                CDF_get_var_num(id, 'MY_VAR'), data_type,
2                                Status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.4.27 CDF\_get\_attr\_zentry\_numelems

```

SUBROUTINE CDF_get_attr_rentry_numelems (

INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attr_num,   ! in -- Attribute number.
INTEGER*4 entry_num, ! in -- Entry number.
INTEGER*4 num_elems, ! out -- Number of elements of the entry.

```

```
INTEGER*4 status)      ! out -- Completion status
```

CDF\_get\_attr\_zentry\_numelems acquires the number of elements of the specified zEntry, corresponding to a zVariable, from an (variable) attribute in a CDF.

The arguments to CDF\_get\_attr\_zentry\_numelems are defined as follows:

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

attr_num    The attribute number.

entry_num   The zEntry number.

num_elems   The number of elements of the zEntry.

status      The completion status code. Chapter 8 explains how to interpret status codes.
```

### 6.4.27.1. Example(s)

The following example acquires the number of elements for zEntry corresponding to zVariable “MY\_VAR” in the variable attribute “MY\_ATTR” in a CDF.

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

.
.
CALL CDF_get_attr_zentry_numelems (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                CDF_get_var_num(id, 'MY_VAR'), num_elems,
2                                status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.28 CDF\_get\_num\_attrs

```
SUBROUTINE CDF_get_num_attrs (

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 num_attrs,        ! out -- Number of attributes.
INTEGER*4 status)           ! out -- Completion status
```

CDF\_get\_num\_attrs acquires the total number of (global and variable) attributes in a CDF.

The arguments to `CDF_get_num_attrs` are defined as follows:

- `id`            The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create_cdf` or `CDF_open_cdf`.
- `num_attrs`    The number of attributes.
- `status`        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.28.1. Example(s)

The following example acquires the total number of attributes in a CDF.

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

.
.
CALL CDF_get_num_attrs (id, attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.29        `CDF_get_num_gattrs`

SUBROUTINE `CDF_get_num_gattrs` (

```
INTEGER*4 id,            ! in -- CDF identifier.
INTEGER*4 attrs,        ! out -- Number of attributes.
INTEGER*4 status)        ! out -- Completion status
```

`CDF_get_num_gattrs` acquires the total number of global attributes in a CDF.

The arguments to `CDF_get_num_gattrs` are defined as follows:

- `id`            The identifier of the CDF. This identifier must have been initialized by a call to `CDF_create_cdf` or `CDF_open_cdf`.
- `attrs`        The number of global attributes.
- `status`        The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.29.1. Example(s)

The following example acquires the total number of global attributes in a CDF.

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

.
.
CALL CDF_get_num_gattrs (id, attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.30 CDF\_get\_num\_vattrs

```
SUBROUTINE CDF_get_num_vattrs (
INTEGER*4 id,          ! in -- CDF identifier.
INTEGER*4 attrs,      ! out -- Number of attributes.
INTEGER*4 status)     ! out -- Completion status
```

CDF\_get\_num\_vattrs acquires the total number of variable attributes in a CDF.

The arguments to CDF\_get\_num\_vattrs are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- attrs        The number of variable attributes.
- status       The completion status code. Chapter 8 explains how to interpret status codes.

#### 6.4.30.1. Example(s)

The following example acquires the total number of variable attributes in a CDF.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id          ! CDF identifier.
INTEGER*4 attrs       ! Attributes.
```



```

INTEGER*4 status      ! Returned status code.

.
.
CALL CDF_get_num_vattrs (id, attrs, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.4.31 CDF\_inquire\_attr

SUBROUTINE CDF\_inquire\_attr (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Attribute number.
CHARACTER attr_name*(CDF_ATTR_NAME_LEN256), ! out -- Attribute name.
INTEGER*4 attr_scope,      ! out -- Attribute scope.
INTEGER*4 max_gentry,      ! out -- Maximum gEntry number if global attribute.
INTEGER*4 max_rentry,      ! out -- Maximum rEntry number if variable attribute.
INTEGER*4 max_zentry,      ! out -- Maximum zEntry number if variable attribute.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_inquire\_attr is used to inquire about the specified attribute. This subroutine expands the original Standard Interface subroutine CDF\_attr\_inquire (Section 5.4) by including an extra information about zEntry if variable attribute is involved. To inquire about a specific attribute entry, use CDF\_inquire\_attr\_gentry (Section 6.4.32), CDF\_inquire\_attr\_rentry (Section 6.4.33) or CDF\_inquire\_attr\_zentry (Section 6.4.34).

The arguments to CDF\_inquire\_attr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The number of the attribute to inquire. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
attr_name	The attribute's name. This character string must be large enough to hold CDF_ATTR_NAME_LEN256 characters and will be blank padded if necessary.
attr_scope	The scope of the attribute. Attribute scopes are defined in Section 4.12.
max_gentry	For gAttributes this is the maximum gEntry number used. 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 CDF_get_attr_num_gentries (see Section 6.4.18). If no entries exist for the attribute, then a value of zero (0) will be passed back.
max_rentry	For vAttributes this is the maximum rEntry number used. 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 CDF_get_attr_num_rentries (see Section 6.4.19). If no entries exist for the attribute, then a value of zero (0) will be passed back.
max_zentry	For vAttributes, this is the maximum zEntry number used. This may not correspond with the number of entries (if some entry numbers were not used). The number of entries



## 6.4.32 CDF\_inquire\_attr\_gentry

SUBROUTINE CDF\_inquire\_attr\_gentry (

```
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Global 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\_inquire\_attr\_gentry is used to inquire about a specific global attribute's entry. To inquire about the attribute in general, use CDF\_inquire\_attr (see Section 6.4.31). CDF\_inquire\_attr\_gentry would normally be called before calling CDF\_get\_attr\_gentry 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_cdf or CDF_open_cdf.
attr_num	The global attribute number for which to inquire an entry. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number to inquire. This is simply the gEntry number and has meaning only to the application.
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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.32.1. Example(s)

The following example inquires each entry for a global 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.

```
.
.
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_LEN256) ! Attribute name.
INTEGER*4 attr_scope       ! Attribute scope.
INTEGER*4 max_gentry       ! Maximum gEntry number used.
INTEGER*4 max_reentry      ! Maximum rEntry number used.
```

```

INTEGER*4 max_zentry           ! Maximum zEntry number used.
INTEGER*4 data_type           ! Data type.
INTEGER*4 num_elems           ! Number of elements (of the
                              ! data type).
.
.
attr_n = CDF_get_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_inquire_attr (id, attr_n, attr_name, attr_scope, max_gentry,
1      max_rentry, max_zentry, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
DO entryN = 1, max_gentry
  CALL CDF_inquire_attr_gentry (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

```

### 6.4.33 CDF\_inquire\_attr\_rentry

SUBROUTINE CDF\_inquire\_attr\_rentry (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,          ! in -- Variable 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\_inquire\_attr\_rentry is used to inquire about a specific entry, corresponding to an rVariable, in a variable attribute, (rEntry). To inquire about the attribute in general, use CDF\_inquire\_attr (see Section 6.4.31). CDF\_inquire\_attr\_rentry would normally be called before calling CDF\_get\_attr\_rentry 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\_get\_attr\_zentry.

The arguments to CDF\_inquire\_attr\_rentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number for which to inquire an entry. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number to inquire. The attribute must be variable in scope. This is the number of the associated rVariable (the rVariable being described in some way by the zEntry).

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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.33.1. Example(s)

The following example inquires each rEntry for variable attribute “TMP” in a CDF. 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.

```

.
.
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_LEN256) ! Attribute name.
INTEGER*4 attr_scope        ! Attribute scope.
INTEGER*4 max_gentry        ! Maximum gEntry number used.
INTEGER*4 max_rentry        ! Maximum rEntry number used.
INTEGER*4 max_zentry        ! Maximum zEntry number used.
INTEGER*4 data_type         ! Data type.
INTEGER*4 num_elems         ! Number of elements (of the
                             ! data type).
.
.
attr_n = CDF_get_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_inquire_attr (id, attr_n, attr_name, attr_scope, max_gentry,
1   max_rentry, max_zentry, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
DO entryN = 1, max_rentry
  CALL CDF_inquire_attr_rentry (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

```

## 6.4.34 CDF\_inquire\_attr\_zentry

SUBROUTINE CDF\_inquire\_attr\_zentry (

```
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Variable 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\_inquire\_attr\_zentry is used to inquire about a specific entry, corresponding to a zVariable, in a variable attribute, (zEntry). To inquire about the attribute in general, use CDF\_inquire\_attr (see Section 6.4.31). CDF\_inquire\_attr\_zentry would normally be called before calling CDF\_get\_attr\_zentry 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\_get\_attr\_zentry.

The arguments to CDF\_inquire\_attr\_zentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number for which to inquire an entry. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number to inquire. The attribute must be variable in scope. This is the number of the associated zVariable (the zVariable being described in some way by the zEntry).
data_type	The data type of the specified entry. The data types are defined in Section .
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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.34.1. Example(s)

The following example inquires each zEntry for variable attribute “TMP” in a CDF. 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.

```
.
.
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_LEN256) ! Attribute name.
INTEGER*4 attr_scope       ! Attribute scope.
INTEGER*4 max_gentry       ! Maximum gEntry number used.
```

```

INTEGER*4 max_rentry           ! Maximum rEntry number used.
INTEGER*4 max_zentry         ! Maximum zEntry number used.
INTEGER*4 data_type          ! Data type.
INTEGER*4 num_elems          ! Number of elements (of the
                             ! data type).
.
.
attr_n = CDF_get_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_inquire_attr (id, attr_n, attr_name, attr_scope, max_gentry,
1                                     max_rentry, max_zentry, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
DO entryN = 1, max_zentry
  CALL CDF_inquire_attr_zentry (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

```

### 6.4.35 CDF\_put\_attr\_gentry

SUBROUTINE CDF\_put\_attr\_gentry (

```

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

```

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

The arguments to CDF\_put\_attr\_gentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The global attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. The attribute must be global in scope.

data_type	The data type of the specified 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
value	The value(s) to write. The entry value is written to the CDF from memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 6.4.35.1. Example(s)

The following example writes one global attribute's gEntry. It is to the global scope attribute VALIDs for gEntry numbered 2. This entry is of CDF\_INT2 type.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 num_elements     ! Number of elements (of data type).
INTEGER*2 TMPvalid         ! Value of VALIDs attribute.

DATA TMPvalids/15/
.
.
num_elements = 1
CALL CDF_put_attr_gentry (id, CDF_get_attr_num(id,'VALIDs'), 2,
1 CDF_INT2, num_elements, TMPvalid, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.36 CDF\_put\_attr\_rentry

SUBROUTINE CDF\_put\_attr\_rentry (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Variable attribute number.
INTEGER*4 entry_num,       ! in -- Entry number.
INTEGER*4 data_type,       ! in -- Data type of this entry.

```



```

INTEGER*4 num_elements,      ! in -- Number of elements (of the data type).
<type>    value,           ! in -- Value (<type> is dependent on the data type of the entry).
INTEGER*4 status)          ! out -- Completion status

```

CDF\_put\_attr\_reentry is used to write an entry, corresponding to an rVariable, (rEntry) to a variable attribute in a CDF. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry.

The arguments to CDF\_put\_attr\_reentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. The attribute must be variable in scope. This is the number of the associated rVariable (the rVariable being described in some way by the zEntry).
data_type	The data type of the specified 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
value	The value(s) to write. The entry value is written to the CDF from memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 6.4.36.1. Example(s)

The following example writes one variable attribute's rEntry. It is to the variable scope attribute VALIDs for the rEntry that corresponds to the zVariable TMP. This entry has two (2) elements, each one is of CDF\_INT2 type.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 num_elements     ! Number of elements (of data type).
INTEGER*2 TMPvalids(2)     ! Value(s) of VALIDs attribute,

DATA TMPvalids/15,30/
.

```

```

.
num_elements = 2
CALL CDF_put_attr_reentry (id, CDF_get_attr_num(id,'VALIDs'),
1      CDF_get_var_num(id,'TMP'),
2      CDF_INT2, num_elements, TMPvalids, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

### 6.4.37 CDF\_put\_attr\_zentry

SUBROUTINE CDF\_put\_attr\_zentry (

```

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

```

CDF\_put\_attr\_zentry is used to write an entry, corresponding to a zVariable, (zEntry) to a variable attribute in a CDF. The entry may or may not already exist. If it does exist, it is overwritten. The data type and number of elements (of that data type) may be changed when overwriting an existing entry.

The arguments to CDF\_put\_attr\_zentry are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
entry_num	The entry number. The attribute must be variable in scope. This is the number of the associated zVariable (the zVariable being described in some way by the zEntry).
data_type	The data type of the specified 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.
value	The value(s) to write. The entry value is written to the CDF from memory address value.  <b>WARNING:</b> 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 8 explains how to interpret status codes.

### 6.4.37.1. Example(s)

The following example writes one variable attribute's zEntry. It is to the variable scope attribute VALIDs for the zEntry that corresponds to the zVariable TMP. This entry has two (2) elements, each one is of CDF\_INT2 type.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
INTEGER*4 num_elements     ! Number of elements (of data type).
INTEGER*2 TMPvalids(2)     ! Value(s) of VALIDs attribute,

DATA TMPvalids/15,30/
.
.
num_elements = 2
CALL CDF_put_attr_zentry (id, CDF_get_attr_num(id,'VALIDs'),
1                          CDF_get_var_num(id,'TMP'),
2                          CDF_INT2, num_elements, TMPvalids, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.38 CDF\_rename\_attr

SUBROUTINE CDF\_rename\_attr (

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

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

The arguments to CDF\_rename\_attr are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The number of the attribute to rename. This number may be determined with a call to CDF_get_attr_num (see Section 6.4.17).
attr_name	The new attribute name. This may be at most CDF_ATTR_NAME_LEN256 characters. Attribute names are case-sensitive.

status                    The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.38.1. 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_rename_attr (id, CDF_get_attr_num(id, 'LAT'), 'LATITUDE', status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.39            CDF\_set\_attr\_gentry\_dataspec

```
SUBROUTINE CDF_set_attr_gentry_dataspec (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Global attribute number.
INTEGER*4 entry_num,        ! in -- gEntry number.
INTEGER*4 data_type,        ! in -- Data type.
INTEGER*4 status)            ! out -- Completion status
```

CDF\_set\_attr\_gentry\_dataspec respecifies the data specification (data type and number of elements) of a gEntry of a global attribute in a CDF. The only part of the data specification that can be changed is the data type. However, the new and old data type must be equivalent. Refer to the CDF User's Guide for the descriptions of equivalent data types.

The arguments to CDF\_set\_attr\_gentry\_dataspec are defined as follows:

id                    The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.

attr\_num             The (global) attribute number.

entry\_num            The gEntry number.

data\_type            The data type.

status                The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.39.1. Example(s)

The following example modifies a gEntry's (numbered 2) data specification in the global attribute "MY\_ATTR" in a CDF. It will change its original data type from CDF\_INT2 to CDF\_UINT2.

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

.
.
entry_num = 2
CALL CDF_set_attr_gentry_dataspec (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                entry_num, CDF_UINT2, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

### 6.4.40 CDF\_set\_attr\_reentry\_ddataspec

```
SUBROUTINE CDF_set_attr_reentry_ddataspec (
INTEGER*4 id,           ! in -- CDF identifier.
INTEGER*4 attr_num,    ! in -- Variable attribute number.
INTEGER*4 entry_num,   ! in -- rEntry number.
INTEGER*4 data_type,   ! in -- Data type.
INTEGER*4 status)      ! out -- Completion status
```

CDF\_set\_attr\_reentry\_ddataspec respecifies the data specification (data type and number of elements) of an rEntry, corresponding to an rVariable, of a variable attribute in a CDF. The only part of the data specification that can be changed is the data type. However, the new and old data type must be equivalent. Refer to the CDF User's Guide for the descriptions of equivalent data types.

The arguments to CDF\_set\_attr\_reentry\_ddataspec are defined as follows:

- id            The identifier of the CDF. This identifier must have been initialized by a call to CDF\_create\_cdf or CDF\_open\_cdf.
- attr\_num    The (variable) attribute number.
- entry\_num   The rEntry number.
- data\_type   The data type.
- status      The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.40.1. Example(s)

The following example modifies an rEntry's (corresponding to rVariable "MY\_VAR") data specification in the variable attribute "MY\_ATTR" in a CDF. It will change its original data type from CDF\_INT2 to CDF\_UINT2.

```
.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
.
.
CALL CDF_set_attr_reentry_dataspec (id, CDF_get_attr_num(id, 'MY_ATTR'),
1                                CDF_get_var_num(id, 'MY_VAR'),
2                                CDF_UINT2, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.
```

## 6.4.41 CDF\_set\_attr\_scope

```
SUBROUTINE CDF_set_attr_scope (
INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Attribute number.
INTEGER*4 scope,           ! in -- Attribute scope.
INTEGER*4 status)          ! out -- Completion status
```

CDF\_set\_attr\_scope respecifies the scope of an attribute in a CDF. Specify one of the scopes described in Section 4.12. Global-scoped attributes will contain only gEntries, while variable-scoped attributes can hold rEntries and zEntries.

The arguments to CDF\_set\_attr\_scope are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The attribute number.
scope	The attribute scope.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.41.1. Example(s)

The following example respecifies the scope to VARIABLE\_SCOPE (from its original GLOBAL\_SCOPE) for attribute "MY\_ATTR" in a CDF.

```

.
.
INCLUDE '<path>cdf.inc'
.
.
INTEGER*4 id                ! CDF identifier.
INTEGER*4 status            ! Returned status code.
.
.
CALL CDF_set_attr_scope (id, CDF_get_attr_num(id, 'MY_ATTR'), VARIABLE_SCOPE,
1                          status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

## 6.4.42 CDF\_set\_attr\_zentry\_dataspec

SUBROUTINE CDF\_set\_attr\_zentry\_dataspec (

```

INTEGER*4 id,                ! in -- CDF identifier.
INTEGER*4 attr_num,         ! in -- Variable attribute number.
INTEGER*4 entry_num,       ! in -- zEntry number.
INTEGER*4 data_type,       ! in -- Data type.
INTEGER*4 status)          ! out -- Completion status

```

CDF\_set\_attr\_zentry\_dataspec respecifies the data specification (data type and number of elements) of a zEntry, corresponding to a zVariable, of a variable attribute in a CDF. The only part of the data specification that can be changed is the data type. However, the new and old data type must be equivalent. Refer to the CDF User's Guide for the descriptions of equivalent data types.

The arguments to CDF\_set\_attr\_zentry\_dataspec are defined as follows:

id	The identifier of the CDF. This identifier must have been initialized by a call to CDF_create_cdf or CDF_open_cdf.
attr_num	The (variable) attribute number.
entry_num	The zEntry number.
data_type	The data type.
num_elems	The number of elements.
status	The completion status code. Chapter 8 explains how to interpret status codes.

### 6.4.42.1 Example(s)

The following example modifies a zEntry's (corresponding to zVariable "MY\_VAR") data specification in the variable attribute "MY\_ATTR" in a CDF. It will change its original data type from CDF\_INT2 to CDF\_UINT2.

```
.  
.   
INCLUDE '<path>cdf.inc'  
.   
.   
INTEGER*4 id           ! CDF identifier.  
INTEGER*4 status       ! Returned status code.  
  
.   
.   
CALL CDF_set_attr_zentry_dataspec (id, CDF_get_attr_num(id, 'MY_ATTR'),  
1                                CDF_get_var_num(id, 'MY_VAR'),  
2                                CDF_UINT2, status)  
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)  
.   
.
```



# Chapter 7

## 7 Internal Interface – CDF\_lib

The Internal interface consists of only one routine, CDF\_lib.<sup>1</sup> 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 7.6.

### 7.1 Example(s)

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

```
.
.
INCLUDE '<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.
INTEGER*4 dim_sizes(1)     ! Dimension sizes.
INTEGER*4 format            ! Format of CDF.

DATA CDF_name/'test1'/, num_dims/0/, dim_sizes/0/,
```

---

<sup>1</sup> See section 6.5.1 for an ugly exception to this.

```

1   format/SINGLE_FILE/
.
.
CALL CDF_create_cdf (CDF_name, 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. <sup>2</sup> This is the case since all of the Standard Interface functions actually call the Internal Interface to perform their operations.
CDF_FORMAT	The item to be put. In this case it is the CDF's format.
format	The actual format for the CDF. Depending on the item being put, one or more arguments would have been necessary. In this case only one argument is necessary.
NULL_	This argument could have been one of two things. It could have been another item to put (followed by the arguments required for that item) or it could have been a new function to perform. In this case it is a new function to perform - the NULL_function. NULL_ indicates the end of the call to CDF_lib. Specifying NULL_ at the end of the argument list is required because not all compilers/operating systems provide the ability for a called function to determine how many arguments were passed in by the calling function.
status	The completion status code. Note that CDF_lib also returns the completion status code. <sup>3</sup> Chapter 8 explains how to interpret status codes.

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_FORMAT_, format,
2             NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

<sup>2</sup> 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.

<sup>3</sup> Section 6.5 explains why it does both.

The purpose of each argument is as follows:

CREATE_	The first function to be performed. In this case something will be created.
CDF_	The item to be created - a CDF in this case. There are four required arguments that must follow. When a CDF is created (with 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 to the CDF.
CDF_FORMAT_	Once again this argument could have been either another item to put or a new function to perform. It is another item to put - the CDF's format.
format	The format to be put to the CDF.
NULL_	This argument could have been either another item to put or a new function to perform. Here it is another function to perform - the NULL_function that ends the call to CDF_lib.
status	The completion status code. Note that CDF_lib also returns the completion status code. Chapter 8 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).

## 7.2 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\_><sup>4</sup> operation. There is no current CDF until one is opened or created (which implicitly selects it) or until one is explicitly selected.<sup>5</sup>

---

<sup>4</sup> This notation is used to specify a function to be performed on an item. The syntax is <function\_,item\_>.

<sup>5</sup> 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.

#### rVariable (object)

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

#### zVariable (object)

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

#### attribute (object)

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

#### gEntry number (state)

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

#### rEntry number (state)

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

#### zEntry number (state)

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

#### record number, rVariables (state)

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

#### record count, rVariables (state)

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

#### record interval, rVariables (state)

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

#### dimension indices, rVariables (state)

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

#### dimension counts, rVariables (state)

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

#### dimension intervals, rVariables (state)

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

#### sequential value, rVariable (state)

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

#### record number, zVariable (state)

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

#### record count, zVariable (state)

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

#### record interval, zVariable (state)

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

dimension indices, zVariable (state)

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

dimension counts, zVariable (state)

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

dimension intervals, zVariable (state)

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

sequential value, zVariable (state)

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

status code (state)

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

## 7.3 Returned Status

CDF\_lib returns a status code of type INTEGER\*4 in the last argument given.<sup>7</sup> Since more than one operation may be performed with a single call to CDF\_lib, the following rules apply:

---

<sup>6</sup> The CDF library now maintains the current status code from one call to the next of CDF\_lib.

<sup>7</sup> 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 8 explains how to interpret status codes. Appendix A lists the possible status codes and the type of each: error, warning, or informational.

## 7.4 Indentation/Style

Indentation should be used to make calls to 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.

## 7.5 Syntax

CDF\_lib takes a variable number of arguments. There must always be at least one argument. The maximum number of arguments is not limited by CDF but rather the Fortran compiler and operating system being used. Under normal circumstances that limit would never be reached (or even approached). Note also 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 fncx is a function to perform, itemx is the item on which to perform the function, and argx is a required argument for the operation. The NULL\_function must be used to end the call to CDF\_lib. The completion status, status, is 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.<sup>8</sup> 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.

## 7.5.1 Macintosh, MPW Fortran

The MPW Fortran compiler does not allow variable length argument lists such as those used by CDF\_lib.<sup>9</sup> 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 CDF support. 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,
1                 PUT_, CDF_ENCODING_, encoding,
2                 CDF_MAJORITY_, majority,
3                 CDF_FORMAT_, format,
4                 NULL_, status)
IF (status .NE. CDF_OK) CALL UserStatusHandler (status)
.
.

```

<sup>8</sup> Current applications do not have to be changed because the completion status code is still stored in the last argument.

<sup>9</sup> 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.



Note that CDF\_lib may still be used but with the same number of arguments for each occurrence.

## 7.6 Operations . . .

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

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

For each function the supported items, required arguments, and required preselected objects/states are listed below. The required preselected objects/states are those objects/states that must be selected (typically with the SELECT\_ function) before a particular operation may be performed. Note that some of the required preselected objects/states have default values as described at Section 7.2.

### <CLOSE\_CDF\_>

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

There are no required arguments.

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

### <CLOSE\_rVAR\_>

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

There are no required arguments.

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

### <CLOSE\_zVAR\_>

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

There are no required arguments.

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

### <CONFIRM\_ATTR\_>

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

out: 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\_LEN256 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 accessibility of the current CDF. If a fatal error occurred while accessing the CDF the error code NO\_MORE\_ACCESS will be returned. If this is the case, the CDF should still be closed.

There are no required arguments.

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

<CONFIRM\_CDF\_CACHESIZE\_>

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

out: 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\_PATHNAME\_LEN)

File name of the CDF.

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

<CONFIRM\_CDF\_NEGtoPOSfp0\_MODE\_>

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

out: 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\_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 (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

The gEntry number.

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

<CONFIRM\_gENTRY\_EXISTENCE\_>

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

in: 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

The rEntry number.

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

<CONFIRM\_rENTRY\_EXISTENCE\_>

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

in: 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\_LEN256 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

The number of cache buffers being used.

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

<CONFIRM\_zENTRY\_>

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

out: INTEGER\*4 entry\_num

The zEntry number.

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

<CONFIRM\_zENTRY\_EXISTENCE\_>

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

in: 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\_LEN256 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\_LEN256 characters. Attribute names are case-sensitive.

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 also be determined with the <GET\_,ATTR\_NUMBER\_> operation.

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

#### <CREATE\_,CDF\_>

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

in: 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.

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 dimension size must be greater than zero (0). For 0-dimensional rVariables this argument is ignored (but must be present). Note that this must be specified even if the CDF will contain only zVariables.

out: 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.

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

There are no required preselected objects/states.

#### `<CREATE_,rVAR_>`

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

in: CHARACTER `var_name`\*(\*)

Name of the rVariable to be created. This can be at most `CDF_VAR_NAME_LEN256` characters (excluding the NUL). Variable names are case-sensitive.

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

in: 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`

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

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

#### <CREATE\_zVAR\_>

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

in: CHARACTER var\_name\*(\*)

Name of the zVariable to be created. This can be at most CDF\_VAR\_NAME\_LEN256 characters. Variable names are case-sensitive.

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

in: 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 also be determined with the <GET\_zVAR\_NUMBER\_> operation.

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

#### <DELETE\_ATTR\_>

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

There are no required arguments.

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

<DELETE\_,CDF\_>

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

There are no required arguments.

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

<DELETE\_,gENTRY\_>

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

There are no required arguments.

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

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

<DELETE\_,rENTRY\_>

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

There are no required arguments.

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

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

<DELETE\_,rVAR\_>

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

There are no required arguments.

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

<DELETE\_,rVAR\_RECORDS\_>

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

Required arguments are as follows:

in: 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 rEntries which numerically follow the rEntries being deleted are also immediately renumbered. When the zVariable is deleted, there is no longer a current zVariable. **NOTE:** This operation is only allowed on single-file CDFs.

There are no required arguments.

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

<DELETE\_zVAR\_RECORDS\_>

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

in: 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 -1 will be passed back.

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

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

<GET\_,ATTR\_MAXrENTRY\_>

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

out: INTEGER\*4 max\_entry

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

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

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

<GET\_,ATTR\_MAXzENTRY\_>

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

out: INTEGER\*4 max\_entry

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

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

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

<GET\_,ATTR\_NAME\_>

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

out: CHARACTER attr\_name\*(CDF\_ATTR\_NAME\_LEN256)

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\_LEN256 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\_CHECKSUM\_>

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

out: INTEGER\*4 checksum

Checksum. The checksum is described in Section 4.19.

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

<GET\_CDF\_COMPRESSION\_>



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

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. The character string will be 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\*8<sup>10</sup> c\_size

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

out: INTEGER\*8<sup>10</sup> 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.

---

<sup>10</sup> You need to have a Fortran compiler supporting 8-byte integer.

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

Value. This buffer must be large to hold the value. (<type> is dependent on the data type of the gEntry). The value is read from the CDF and placed into memory at address 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

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

<GET\_LIB\_COPYRIGHT\_>

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

out: CHARACTER copy\_right\*(CDF\_COPYRIGHT\_LEN)

CDF library copyright text.

**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

Value. This buffer must be large to hold the value. <type> is dependent on the data type of the rEntry. The value is read from the CDF and placed into memory at address 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

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

<GET\_rVAR\_ALLOCATEDFROM\_>

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

in: 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\_><sup>11</sup>

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

Value. This buffer must be large to hold the value. <type> is dependent on the data type of the rVariable. The value is read from the CDF and placed into memory at address 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.

---

<sup>11</sup> The item rVAR\_BLOCKINGFACTOR was previously named rVAR\_EXTENDRECS.

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.

<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. This buffer must be large to hold the value. <type> is dependent on the data type of the rVariable. The value is read from the CDF and placed into memory at address 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, 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 zero (0). A value of negative one (-1) 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\_LEN256

Name of the rVariable. This character string will be 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\_LEN256 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. This buffer must be large to hold the value. <type> is dependent on the data type of the pad value. The value is read from the CDF and placed into memory at address 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. This buffer must be large to hold the value. <type> is dependent on the data type of the rVariable. The value is read from the CDF and placed into memory at address 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 a\_arrays\_parms(CDF\_MAX\_PARMS)

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

out: INTEGER\*4 a\_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(CDF\_MAX\_DIMS)

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

out: 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\_RECADATA\_>

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.

out: <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.<sup>12</sup>

<GET\_STATUS\_TEXT\_>

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

out: CHARACTER text\*(CDF\_STATUSTEXT\_LEN)

Text explaining the status code.

**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

Value. This buffer must be large to hold the value. <type> is dependent on the data type of the zEntry. The value is read from the CDF and placed into memory at address 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

---

<sup>12</sup> A Standard Interface at Section 5.13 provides the same functionality.

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 (an array of characters). For all other data types this is the number of elements in an array of that data type.

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

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

<GET\_zVAR\_ALLOCATEDFROM\_>

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

in: 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\_><sup>13</sup>

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

---

<sup>13</sup> The item zVAR\_BLOCKINGFACTOR was previously named zVAR\_EXTENDRECS .

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

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

<GET\_,zVAR\_COMPRESSION\_>

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

out: 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

Value. This buffer must be large to hold the value. <type> is dependent on the data type of the zVariable. The value is read from the CDF and placed into memory at address 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`

Value. This buffer must be large to hold the value. <type> is dependent on the data type of the zVariable. The value is read from the CDF and placed into memory at address 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, 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 zero (0). A value of negative one (-1) 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\_LEN256)

Name of the zVariable.

**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 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\_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 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\_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\_LEN256 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 zVariable number.

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

<GET\_zVAR\_NUMDIMS\_>

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

out: 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

Pad value. This buffer must be large to hold the value. <type> is dependent on the data type of the zVariable. The value is read from the CDF and placed into memory at address 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

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

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

<GET\_zVAR\_SPARSEARRAYS\_>

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

out: INTEGER\*4 s\_arrays\_type

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

out: INTEGER\*4 a\_arrays\_parms(CDF\_MAX\_PARMS)

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

out: INTEGER\*4 a\_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. Note that this is not the number of records but rather the maximum record number (which is one less than the number of records). A value of negative one (-1) indicates that the zVariables contain no records. The maximum record number for an individual zVariable may be inquired using the <GET\_zVAR\_MAXREC\_> operation. Required arguments are as follows:

out: 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.

out: <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 <SELECT\_zVAR\_RECNUMBER\_>).<sup>14</sup>

---

<sup>14</sup> A Standard Interface at Section 5.14 provides the same functionality.

<NULL\_>

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

<OPEN\_CDF\_>

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

in: 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 on the CDF.

There are no required preselected objects/states.

<PUT\_ATTR\_NAME\_>

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

in: CHARACTER attr\_name\*(\*)

New attribute name. This may be at most CDF\_ATTR\_NAME\_LEN256 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\_CHECKSUM\_>

Respecifies the checksum mode for the current CDF. Required arguments are as follows:

in: INTEGER\*4 checksum

New checksum. The checksum is described in Section 4.19.

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

<PUT\_CDF\_COMPRESSION\_>

Specifies the compression type/parameters for the current CDF. This refers to the compression of the CDF - not of any variables. Required arguments are as follows:

in: INTEGER\*4 cType

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 (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: <type> value

Value. <type> is dependent 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 (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: <type> value

Value. <type> is dependent 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 0 (zero). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: 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\_><sup>15</sup>

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

in: INTEGER\*4 blockingFactor

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

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

<PUT\_rVAR\_COMPRESSION\_>

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

in: INTEGER\*4 cType

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> is dependent 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

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

in: INTEGER\*4 num\_elements

---

<sup>15</sup> The item rVAR\_BLOCKINGFACTOR was previously named rVAR\_EXTENDRECS .

Number of elements of the data type at each value. For character data types (CDF\_CHAR and CDF\_UCHAR), this is the number of characters in each string (an array of characters). A string exists at each value. For the non-character data types this must be one (1) - arrays of values are not allowed for non-character data types.

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

Value. <type> is dependent 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 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, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

#### <PUT\_rVAR\_INITIALRECS\_>

Specifies the number of records to initially write to the current rVariable (in the current CDF). The records are written beginning at record number 0 (zero). This may be specified only once per rVariable and before any other records have been written to that rVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: 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\_LEN256 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

Pad value. <type> is dependent 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\_RECVAR\_>

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> is dependent 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 a\_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\_SPASERECORDS\_>

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\_RECADATA\_>

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.<sup>16</sup>

#### <PUT\_zENTRY\_DATA\_>

---

<sup>16</sup> A Standard Interface at Section 5.17 provides the same functionality.

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 (an array of characters). 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 0 (zero). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: 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\_><sup>17</sup>

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

in: INTEGER\*4 blockingFactor

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

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

<PUT\_zVAR\_COMPRESSION\_>

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

in: INTEGER\*4 cType

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.

<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

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

---

<sup>17</sup> The item zVAR\_BLOCKINGFACTOR was previously named zVAR\_EXTENDRECS .

**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 (an array of characters). A string exists at each value. For the non-character data types this must be one (1) - arrays of values are not allowed for non-character data types.

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

<PUT\_zVAR\_DIMVARYS\_>

Respecifies the dimension variances of the current zVariable (in the current CDF). A zVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value - it may have been written). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: 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 0 (zero). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: 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

Value. <type> is dependent 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, 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\_LEN256 characters.

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

Pad value. <type> is dependent 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 and its current zVariable.

<PUT\_zVAR\_RECVAR\_>

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

Value. <type> is dependent 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 a\_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\_RECADATA\_>

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 <SELECT\_,zVAR\_RECNUMBER\_>).<sup>18</sup>

<SELECT\_,ATTR\_>

Explicitly selects the current attribute (in the current CDF) by number. 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\_LEN256 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\_>

---

<sup>18</sup> A Standard Interface at Section 5.18 provides the same functionality.

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 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\_LEN256 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\_LEN256 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 a 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\_LEN256 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\_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\_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\_LEN256 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\_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 a 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.

## 7.7 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 <SELECT\_,CDF\_>).

### 7.7.1 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.inc'   
.   
.   
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)
.
.

```

## 7.7.2 zVariable Creation (Character Data Type)

In this example a zVariable with a character data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```

.
.
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)
.
.

```

## 7.7.3 Hyper Read with Subsampling

In this example an rVariable will be subsampled in a CDF whose rVariables are 2-dimensional and have dimension sizes [100,200]. The CDF is column major, and the data type of the rVariable is CDF\_UINT2. 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
.
.

```

## 7.7.4 Attribute Renaming

In this example the attribute named Tmp will be renamed to TMP. It is assumed that the current CDF has already been selected.

```
.
.
INCLUDE '<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)
.
.
```

## 7.7.5 Sequential Access

In this example the values for a zVariable will be averaged. The values will be read using the sequential access method (see the Concepts chapter in the CDF User's Guide). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be CDF\_REAL4. It is assumed that the current CDF has already been selected.

```
.
.
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

.
.

```

## 7.7.6 Attribute rEntry Writes

In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```

.
.
INCLUDE '<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)
.
.

```

## 7.7.7 Multiple zVariable Write

In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see the Concepts chapter in the CDF User's Guide). It is assumed that the current CDF has already been selected.

```

.
.
INCLUDE '<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 8

## 8 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.

<code>status &gt; CDF_OK</code>	Indicates successful completion but some additional information is provided. These are informational codes.
<code>status = CDF_OK</code>	Indicates successful completion.
<code>CDF_WARN &lt; status &lt; CDF_OK</code>	Indicates that the function completed but probably not as expected. These are warning codes.
<code>status &lt; CDF_WARN</code>	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 encodes in a text string an explanation of a given status code.

# Chapter 9

## 9 EPOCH Utility Routines

Several subroutines exist that compute, decompose, parse, and encode CDF\_EPOCH and CDF\_EPOCH16 values. These functions may be called by applications using the CDF\_EPOCH and CDF\_EPOCH16 data types and are included in the CDF library. Function prototypes for these functions may be found in the include file cdf.h. The Concepts chapter in the CDF User's Guide describes EPOCH values.

The CDF\_EPOCH and CDF\_EPOCH16 data types are used to store time values referenced from a particular epoch. For CDF that epoch values for CDF\_EPOCH and CDF\_EPOCH16 are 01-Jan-0000 00:00:00.000 and 01-Jan-0000 00:00:00.000.000.000.000, respectively.

### 9.1 compute\_EPOCH

compute\_EPOCH calculates a CDF\_EPOCH value given the individual components. If an illegal component is detected, the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE compute_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.

## 9.2 EPOCH\_breakdown

EPOCH\_breakdown decomposes a CDF\_EPOCH value into the individual components.

```
SUBROUTINE EPOCH_breakdown (  
  
  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).
```

## 9.3 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 (  
  
  REAL*8    epoch;          ! in -- The CDF_EPOCH value.  
  CHARACTER epString*(EPOCH_STRING_LEN) ! out -- The standard date/time character string.
```

EPOCH\_STRING\_LEN is defined in cdf.inc.

## 9.4 encode\_EPOCH1

encode\_EPOCH1 encodes a CDF\_EPOCH value into an alternate date/time character string. The format of the string is yyyyymmdd.tttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and tttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
SUBROUTINE encode_EPOCH1(  
  
  REAL*8    epoch;          ! in -- The CDF_EPOCH value.  
  CHARACTER epString*(EPOCH1_STRING_LEN) ! out -- The alternate date/time character string.
```

EPOCH1\_STRING\_LEN is defined in cdf.inc.

## 9.5 encode\_EPOCH2

encode\_EPOCH2 encodes a CDF\_EPOCH value into an alternate date/time character string. The format of the string is `yyyymmddhhmmss` 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 (  
  
    REAL*8      epoch;                ! in  -- The CDF_EPOCH value.  
    CHARACTER  epString*(EPOCH2_STRING_LEN) ! out -- The alternate date/time character string.
```

EPOCH2\_STRING\_LEN is defined in `cdf.inc`.

## 9.6 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 (  
  
    REAL*8      epoch;                ! in  -- The CDF_EPOCH value.  
    CHARACTER  epString*(EPOCH3_STRING_LEN) ! out -- The alternate date/time character string.
```

EPOCH3\_STRING\_LEN is defined in `cdf.inc`.

## 9.7 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 (  
  
    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. . .

<u>Token</u>	<u>Meaning</u>	<u>Default</u>
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 9.3) would be. . .

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

EPOCHx\_FORMAT\_LEN and EPOCHx\_STRING\_MAX are defined in cdf.inc.

## 9.8 parse\_EPOCH

parse\_EPOCH parses a standard date/time character string and returns a CDF\_EPOCH value. The format of the string is that produced by the encode\_EPOCH function described in Section 9.3. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH (
CHARACTER epString*(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.

## 9.9 parse\_EPOCH1

parse\_EPOCH1 parses An alternate date/time character string and returns a CDF\_EPOCH value. The format of the string is that produced by the encode\_EPOCH1 function described in Section 9.4. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH1 (
CHARACTER epString*(EPOCH1_STRING_LEN),      ! in -- The standard date/time character string.
REAL*8     epoch)                             ! out -- CDF_EPOCH value
```

EPOCH1\_STRING\_LEN is defined in cdf.inc.

## 9.10 parse\_EPOCH2

parse\_EPOCH2 parses An alternate date/time character string and returns a CDF\_EPOCH value. The format of the string is that produced by the encode\_EPOCH2 function described in Section 9.5. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH2 (  
  
    CHARACTER epString*(EPOCH2_STRING_LEN),      ! in -- The standard date/time character string.  
    REAL*8     epoch)                             ! out -- CDF_EPOCH value
```

EPOCH2\_STRING\_LEN is defined in cdf.inc.

## 9.11 parse\_EPOCH3

parse\_EPOCH3 parses An alternate date/time character string and returns a CDF\_EPOCH value. The format of the string is that produced by the encode\_EPOCH3 function described in Section 9.6. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH3 (  
  
    CHARACTER epString*(EPOCH3_STRING_LEN),      ! in -- The standard date/time character string.  
    REAL*8     epoch)                             ! out -- CDF_EPOCH value
```

EPOCH3\_STRING\_LEN is defined in cdf.inc.

## 9.12 compute\_EPOCH16

compute\_EPOCH16 calculates a CDF\_EPOCH16 value given the individual components. If An illegal component is detected, the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE compute_EPOCH16 (  
  
    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).  
    INTEGER*4 usec,          ! in -- Microsecond (0-999).  
    INTEGER*4 nsec,          ! in -- Nanosecond (0-999).
```

```

INTEGER*4 psec,           ! in -- Picosecond (0-999).
REAL*8   epoch(2))      ! out-- CDF_EPOCH16 value

```

## 9.13 EPOCH16\_breakdown

EPOCH16\_breakdown decomposes a CDF\_EPOCH16 value into the individual components.

```

SUBROUTINE EPOCH_breakdown (

REAL*8   epoch(2),           ! in -- The CDF_EPOCH16 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).
INTEGER*4 usec,            ! out -- Microsecond (0-999).
INTEGER*4 nsec,            ! out -- Nanosecond (0-999).
INTEGER*4 psec)            ! out -- Picosecond (0-999).

```

## 9.14 encode\_EPOCH16

encode\_EPOCH16 encodes a CDF\_EPOCH16 value into the standard date/time character string. The format of the string is dd-mmm-yyyy hh:mm:ss.ccc.uuu.nnn.ppp where dd is the day of the month (1-31), mmm is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), yyyy is the year, hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```

SUBROUTINE encode_EPOCH16 (

REAL*8   epoch(2),           ! in -- The CDF_EPOCH16 value.
CHARACTER epString*(EPOCH16_STRING_LEN) ! out -- The standard date/time string.

```

EPOCH16\_STRING\_LEN is defined in cdf.inc.

## 9.15 encode\_EPOCH16\_1

encode\_EPOCH16\_1 encodes a CDF\_EPOCH16 value into an alternate date/time character string. The format of the string is yyymmdd.tttttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and tttttttttt is the fraction of the day (e.g., 5000000000000000 is 12 o'clock noon).

```

SUBROUTINE encode_EPOCH16_1(

```

```

REAL*8      epoch(2),
CHARACTER  epString*(EPOCH16_1_STRING_LEN))
! in  -- The CDF_EPOCH16 value.
! out -- The date/time string.

```

EPOCH16\_1\_STRING\_LEN is defined in cdf.inc.

## 9.16 encode\_EPOCH16\_2

encode\_EPOCH16\_2 encodes a CDF\_EPOCH16 value into an alternate date/time character string. The format of the string is yyymoddhmmss 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_EPOCH16_2 (
REAL*8      epoch(2),
CHARACTER  epString*(EPOCH16_2_STRING_LEN))
! in  -- The CDF_EPOCH16 value.
! out -- The date/time string.

```

EPOCH16\_2\_STRING\_LEN is defined in cdf.inc.

## 9.17 encode\_EPOCH16\_3

encode\_EPOCH16\_3 encodes a CDF\_EPOCH16 value into an alternate date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.ccc.uuu.nnn.pppZ where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), ccc is the millisecond (0-999), uuu is the microsecond (0-999), nnn is the nanosecond (0-999), and ppp is the picosecond (0-999).

```

SUBROUTINE encode_EPOCH16_3 (
REAL*8      epoch(2),
CHARACTER  epString*(EPOCH16_3_STRING_LEN))
! in  -- The CDF_EPOCH16 value.
! out -- The date/time string.

```

EPOCH16\_3\_STRING\_LEN is defined in cdf.inc.

## 9.18 encode\_EPOCH16\_x

encode\_EPOCH16\_x encodes a CDF\_EPOCH16 value into a custom date/time character string. The format of the encoded string is specified by a format string.

```

SUBROUTINE encode_EPOCH16_x (
REAL*8 epoch(2);
CHARACTER format*(EPOCHx_FORMAT_MAX)
CHARACTER encoded*(EPOCHx_STRING_MAX))
! in  -- The CDF_EPOCH16 value.
! in  -- The format string.
! 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>
msec	Millisecond (000-999)	<msec.3>
usec	Microsecond (000-999)	<usec.3>
nsec	Nanosecond (000-999)	<nsec.3>
psec	Picosecond (000-999)	<psec.3>
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 EPOCH16 date/time character string (see Section 9.14) would be. . .

```
<dom.02>-<month>-<year> <hour>:<min>:<sec>.<msec>.<usec>.<nsec>.<psec>.<fos>
```

EPOCHx\_FORMAT\_LEN and EPOCHx\_STRING\_MAX are defined in cdf.inc.

## 9.19 parse\_EPOCH16

parse\_EPOCH16 parses a standard date/time character string and returns a CDF\_EPOCH16 value. The format of the string is that produced by the encode\_EPOCH16 function. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH16 (
CHARACTER epString*(EPOCH16_STRING_LEN),      ! in -- The date/time string.
REAL*8    epoch(2)                             ! out -- CDF_EPOCH16 value
```

EPOCH16\_STRING\_LEN is defined in cdf.inc.

## 9.20 parse\_EPOCH16\_1

parse\_EPOCH16\_1 parses an alternate date/time character string and returns a CDF\_EPOCH16 value. The format of the string is that produced by the encode\_EPOCH16\_1 function. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH16_1 (  
  
    CHARACTER epString*(EPOCH16_1_STRING_LEN),      ! in -- The date/time string.  
    REAL*8    epoch(2)                               ! out -- CDF_EPOCH16 value
```

EPOCH16\_1\_STRING\_LEN is defined in cdf.inc.

## 9.21 parse\_EPOCH16\_2

parse\_EPOCH16\_2 parses an alternate date/time character string and returns a CDF\_EPOCH16 value. The format of the string is that produced by the encode\_EPOCH16\_2 function. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH16_2 (  
  
    CHARACTER epString*(EPOCH16_2_STRING_LEN),      ! in -- The date/time string.  
    REAL*8    epoch(2)                               ! out -- CDF_EPOCH16 value
```

EPOCH16\_2\_STRING\_LEN is defined in cdf.inc.

## 9.22 parse\_EPOCH16\_3

parse\_EPOCH16\_3 parses an alternate date/time character string and returns a CDF\_EPOCH16 value. The format of the string is that produced by the encode\_EPOCH16\_3 function. If an illegal field is detected in the string the value returned will be ILLEGAL\_EPOCH\_VALUE.

```
SUBROUTINE parse_EPOCH16_3 (  
  
    CHARACTER epString*(EPOCH16_3_STRING_LEN      ), ! in -- The date/time string.  
    REAL*8    epoch(2)                               ! out -- CDF_EPOCH16 value
```

EPOCH16\_3\_STRING\_LEN is defined in cdf.inc.

# Appendix A

## A.1 Introduction

A status code is returned from most CDF functions. The `cdf.inc` (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.

<code>ATTR_EXISTS</code>	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]
<code>ATTR_NAME_TRUNC</code>	Attribute name truncated to <code>CDF_ATTR_NAME_LEN256</code> 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 <sup>1</sup>	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_CHECKSUM	An illegal checksum mode received. It is invalid or currently not supported. [Error]
BAD_COMPRESSION_PARM	An illegal compression parameter was specified. [Error]
BAD_DATA_TYPE	An unknown data type was specified or encountered. The CDF data types are defined in cdf.inc 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.inc for C applications and in cdf.inc for Fortran applications. [Error]
BAD_DIM_COUNT	Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension. [Error]
BAD_DIM_INDEX	One or more dimension index is out of range. A valid value must be specified regardless of the dimension variance. Note also that

---

<sup>1</sup> The status code BAD\_BLOCKING\_FACTOR was previously named BAD\_EXTEND\_RECS.

	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.inc 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.inc 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.inc 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.inc 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.inc 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.inc for C applications and in cdf.inc for Fortran applications. [Error]
BAD_SCRATCH_DIR	An illegal scratch directory was specified. The scratch directory must be writeable and accessible (if a relative path was specified) from the directory in which the application has been executed. [Error]
BAD_SPARSEARRAYS_PARM	An illegal sparse arrays parameter was specified. [Error]
BAD_VAR_NAME	Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]
BAD_VAR_NUM	Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]
BAD_zMODE	Illegal zMode specified. The CDF zModes are defined in cdf.inc 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	<p>Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow:</p> <ol style="list-style-type: none"> <li>1. Changing a CDF's data encoding after a variable value (including a pad value) or an attribute entry has been written.</li> <li>2. Changing a CDF's format after a variable has been created or if a compressed single-file CDF.</li> <li>3. Changing a CDF's variable majority after a variable value (excluding a pad value) has been written.</li> <li>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.</li> <li>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.</li> </ol>

6. Changing a variable's dimension variances after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
7. Writing "initial" records to a variable after a value (excluding the pad value) has already been written to that variable.
8. Changing a variable's blocking factor when a compressed variable and a value (excluding the pad value) has been written or when a variable with sparse records and a value has been accessed.
9. Changing an attribute entry's data specification where the new specification is not equivalent to the old specification.

CANNOT\_COMPRESS

The CDF or variable cannot be compressed. For CDFs, this occurs if the CDF has the multi-file format. For variables, this occurs if the variable is in a multi-file CDF, values have been written to the variable, or if sparse arrays have already been specified for the variable. [Error]

CANNOT\_SPARSEARRAYS

Sparse arrays cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, records have been allocated for the variable, or if compression has already been specified for the variable. [Error]

CANNOT\_SPARSERECORDS

Sparse records cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, or records have been allocated for the variable. [Error]

CDF\_CLOSE\_ERROR

Error detected while trying to close CDF. Check that sufficient disk space exists for the dotCDF file and that it has not been corrupted. [Error]

CDF\_CREATE\_ERROR

Cannot create the CDF specified - error from file system. Make sure that sufficient privilege exists to create the dotCDF file in the disk/directory location specified and that an open file quota has not already been reached. [Error]

CDF\_DELETE\_ERROR

Cannot delete the CDF specified - error from file system. Insufficient privileges exist the delete the CDF file(s). [Error]

CDF\_EXISTS

The CDF named already exists - cannot create it. The CDF library will not overwrite an existing CDF. [Error]

CDF\_INTERNAL\_ERROR

An unexpected condition has occurred in the CDF library. Report this error to CDFsupport. [Error]

CDF\_NAME\_TRUNC

CDF file name truncated to CDF\_PATHNAME\_LEN characters. The CDF was created but with a truncated name. [Warning]

CDF\_OK

Function completed successfully.

CDF_OPEN_ERROR	Cannot open the CDF specified - error from file system. Check that the dotCDF file is not corrupted and that sufficient privilege exists to open it. Also check that an open file quota has not already been reached. [Error]
CDF_READ_ERROR	Failed to read the CDF file - error from file system. Check that the dotCDF file is not corrupted. [Error]
CDF_WRITE_ERROR	Failed to write the CDF file - error from file system. Check that the dotCDF file is not corrupted. [Error]
CHECKSUM_ERROR	The data integrity verification through the checksum failed. [Error]
CHECKSUM_NOT_ALLOWED	The checksum is not allowed for old versioned files. [Error]
COMPRESSION_ERROR	An error occurred while compressing a CDF or block of variable records. This is an internal error in the CDF library. Contact CDF User Support. [Error]
CORRUPTED_V2_CDF	This Version 2 CDF is corrupted. An error has been detected in the CDF's control information. If the CDF file(s) are known to be valid, please contact CDF User Support. [Error]
DECOMPRESSION_ERROR	An error occurred while decompressing a CDF or block of variable records. The most likely cause is a corrupted dotCDF file. [Error]
DID_NOT_COMPRESS	For a compressed variable, a block of records did not compress to smaller than their uncompressed size. They have been stored uncompressed. This can result if the blocking factor is set too low or if the characteristics of the data are such that the compression algorithm chosen is unsuitable. [Informational]
EMPTY_COMPRESSED_CDF	The compressed CDF being opened is empty. This will result if a program which was creating/modifying the CDF abnormally terminated. [Error]
END_OF_VAR	The sequential access current value is at the end of the variable. Reading beyond the end of the last physical value for a variable is not allowed (when performing sequential access). [Error]
FORCED_PARAMETER	A specified parameter was forced to an acceptable value (rather than an error being returned). [Warning]
IBM_PC_OVERFLOW	An operation involving a buffer greater than 64k bytes in size has been specified for PCs running 16-bit DOS/Windows 3.*. [Error]
ILLEGAL_FOR_SCOPE	The operation is illegal for the attribute's scope. For example, only gEntries may be written for gAttributes - not rEntries or zEntries. [Error]
ILLEGAL_IN_zMODE	The attempted operation is illegal while in zMode. Most operations involving rVariables or rEntries will be illegal. [Error]



ILLEGAL_ON_V1_CDF	The specified operation (i.e., opening) is not allowed on Version 1 CDFs. [Error]
MULTI_FILE_FORMAT	The specified operation is not applicable to CDFs with the multi-file format. For example, it does not make sense to inquire indexing statistics for a variable in a multi-file CDF (indexing is only used in single-file CDFs). [Informational]
NA_FOR_VARIABLE	The attempted operation is not applicable to the given variable. [Warning]
NEGATIVE_FP_ZERO	One or more of the values read/written are -0.0 (An illegal value on VAXes and DEC Alphas running OpenVMS). [Warning]
NO_ATTR_SELECTED	An attribute has not yet been selected. First select the attribute on which to perform the operation. [Error]
NO_CDF_SELECTED	A CDF has not yet been selected. First select the CDF on which to perform the operation. [Error]
NO_DELETE_ACCESS	Deleting is not allowed (read-only access). Make sure that delete access is allowed on the CDF file(s). [Error]
NO_ENTRY_SELECTED	An attribute entry has not yet been selected. First select the entry number on which to perform the operation. [Error]
NO_MORE_ACCESS	Further access to the CDF is not allowed because of a severe error. If the CDF was being modified, an attempt was made to save the changes made prior to the severe error. In any event, the CDF should still be closed. [Error]
NO_PADVALUE_SPECIFIED	A pad value has not yet been specified. The default pad value is currently being used for the variable. The default pad value was returned. [Informational]
NO_STATUS_SELECTED	A CDF status code has not yet been selected. First select the status code on which to perform the operation. [Error]
NO_SUCH_ATTR	The named attribute was not found. Note that attribute names are case-sensitive. [Error]
NO_SUCH_CDF	The specified CDF does not exist. Check that the file name specified is correct. [Error]
NO_SUCH_ENTRY	No such entry for specified attribute. [Error]
NO_SUCH_RECORD	The specified record does not exist for the given variable. [Error]
NO_SUCH_VAR	The named variable was not found. Note that variable names are case-sensitive. [Error]
NO_VAR_SELECTED	A variable has not yet been selected. First select the variable on which to perform the operation. [Error]

NO_VARS_IN_CDF	This CDF contains no rVariables. The operation performed is not applicable to a CDF with no rVariables. [Informational]
NO_WRITE_ACCESS	Write access is not allowed on the CDF file(s). Make sure that the CDF file(s) have the proper file system privileges and ownership. [Error]
NOT_A_CDF	Named CDF is corrupted or not actually a CDF. 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 preceding the range of records being allocated were automatically allocated as well. [Informational]
READ_ONLY_DISTRIBUTION	Your CDF distribution has been built to allow only read access to CDFs. Check with your system manager if you require write access. [Error]
READ_ONLY_MODE	The CDF is in read-only mode - modifications are not allowed. [Error]
SCRATCH_CREATE_ERROR	Cannot create a scratch file - error from file system. If a scratch directory has been specified, ensure that it is 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 caused by a corrupted CDF or if the CDF was created/modified by a CDF distribution more recent than the one being used. [Error]
TOO_MANY_VARS	A multi-file CDF on a PC may contain only a limited number of variables because of the 8.3 file naming convention of MS-DOS. This consists of 100 rVariables and 100 zVariables. [Error]
UNKNOWN_COMPRESSION	An unknown type of compression was specified or encountered. [Error]
UNKNOWN_SPARSENESS	An unknown type of sparseness was specified or encountered. [Error]

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



# Appendix B

## B.1 Standard Interface (original)

```
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

SUBROUTINE CDF_attr_entry_inquire (id, attr_num, entry_num, data_type, num_elements,
1                                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_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_attr_inquire (id, attr_num, attr_name, attr_scope, max_entry, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                         ! in
CHARACTER    attr_name*(*)                    ! out
INTEGER*4    attr_scope                       ! out
INTEGER*4    max_entry                       ! out
INTEGER*4    status                           ! out

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

SUBROUTINE CDF_attr_put (id, attr_num, entry_num, data_type, num_elements, value,
1                                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_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_close (id, status)
INTEGER*4    id                       ! in
INTEGER*4    status                   ! out

SUBROUTINE CDF_create (CDF_name, num_dims, dim_sizes, encoding, majority, 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_delete (id, status)
INTEGER*4    id                       ! in
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_error (status, message, status)
INTEGER*4    status                   ! in
CHARACTER    message*(CDF_STATUSTEXT_LEN) ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_getrvarsrecorddata (id, num_var, var_nums, rec_num,
1                                buffer, status)
INTEGER*4    id                       ! in
INTEGER*4    num_var                  ! in
INTEGER*4    var_nums(*)              ! in
INTEGER*4    rec_num                  ! in
<type>      buffer                   ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_getzvarsrecorddata (id, num_var, var_nums, rec_num,
1                                buffer, status)
INTEGER*4    id                       ! in
INTEGER*4    num_var                  ! in
INTEGER*4    var_nums(*)              ! in
INTEGER*4    rec_num                  ! in

```

```

<type>      buffer                ! out
INTEGER*4   status                ! out

SUBROUTINE CDF_inquire (id, num_dims, dim_sizes, encoding, majority, 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_open (CDF_name, id, status)
CHARACTER   CDF_name*(*)         ! in
INTEGER*4   id                   ! out
INTEGER*4   status               ! out

SUBROUTINE CDF_putrvsrecorndata (id, num_var, var_nums, rec_num,
1                                buffer, status)
INTEGER*4   id                    ! in
INTEGER*4   num_var               ! in
INTEGER*4   var_nums(*)          ! in
INTEGER*4   rec_num              ! in
<type>     buffer                ! in
INTEGER*4   status               ! out

SUBROUTINE CDF_putzvsrecorndata (id, num_var, var_nums, rec_num,
1                                buffer, status)
INTEGER*4   id                    ! in
INTEGER*4   num_var               ! in
INTEGER*4   var_nums(*)          ! in
INTEGER*4   rec_num              ! in
<type>     buffer                ! in
INTEGER*4   status               ! out

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

SUBROUTINE CDF_var_create (id, var_name, data_type, num_elements, rec_variances,
1                           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

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_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_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_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_LEN256) ! 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

```

```

INTEGER*4 FUNCTION CDF_var_num (id, var_name)

```

```

INTEGER*4    id                    ! in
CHARACTER    var_name*(*)          ! in

```

```

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_rename (id, var_num, var_name, status)

```



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



## B.2 Standard Interface (new)

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

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

INTEGER*4 FUNCTION CDF_confirm_attr_existence (id, attr_name)
INTEGER*4    id                ! in
CHARACTER    attr_name*(*)    ! in

INTEGER*4 FUNCTION CDF_confirm_gentry_existence (id, attr_num, entry_num)
INTEGER*4    id                ! in
INTEGER*4    attr_num         ! in
INTEGER*4    entry_num       ! in

INTEGER*4 FUNCTION CDF_confirm_reentry_existence (id, attr_num, entry_num)
INTEGER*4    id                ! in
INTEGER*4    attr_num         ! in
INTEGER*4    entry_num       ! in

INTEGER*4 FUNCTION CDF_confirm_zentry_existence (id, attr_num, entry_num)
INTEGER*4    id                ! in
INTEGER*4    attr_num         ! in
INTEGER*4    entry_num       ! in

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

INTEGER*4 FUNCTION CDF_confirm_zvar_padvalue_exist (id, var_num)
INTEGER*4    id                ! in
INTEGER*4    var_num         ! in

SUBROUTINE CDF_create_attr (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

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

```

SUBROUTINE CDF_create_zvar (id, var_name, data_type, num_elements, num_dims,
1                          dim_sizes, rec_variances, 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    num_dims                          ! in
INTEGER*4    dim_sizes(*)                      ! in
INTEGER*4    rec_variance                      ! in
INTEGER*4    dim_variances(*)                 ! in
INTEGER*4    var_num                           ! out
INTEGER*4    status                            ! out

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

SUBROUTINE CDF_delete_attr_gentry (id, attr_num, entry_num, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                         ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_delete_attr_rentry (id, attr_num, entry_num, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                         ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_delete_attr_zentry (id, attr_num, entry_num, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                         ! in
INTEGER*4    status                            ! out

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

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

SUBROUTINE CDF_delete_zvar_recs (id, var_num, start_rec, end_rec, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    start_rec                         ! in
INTEGER*4    end_rec                           ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_attr_gentry_datatype (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                         ! in

```

```

INTEGER*4    data_type                ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_gentry_numelems (id, attr_num, entry_num, num_elems, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entry_num               ! in
INTEGER*4    num_elems               ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_gentry (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_get_attr_max_gentry (id, attr_num, entry_num, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entry_num               ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_max_rentry (id, attr_num, entry_num, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entry_num               ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_max_zentry (id, attr_num, entry_num, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entry_num               ! out
INTEGER*4    status                   ! out

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

INTEGER*4 FUNCTION CDF_get_attr_num (id, attr_name, status)
INTEGER*4    id                       ! in
CHARACTER    attr_name*(*)           ! in
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_num_gentries (id, attr_num, entries, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entries                 ! out
INTEGER*4    status                   ! out

SUBROUTINE CDF_get_attr_num_rentries (id, attr_num, entries, status)
INTEGER*4    id                       ! in
INTEGER*4    attr_num                 ! in
INTEGER*4    entries                 ! out

```

```

INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_num_zentries (id, attr_num, entries, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    entries                               ! out
INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_rentry (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_get_attr_rentry_datatype (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    entry_num                             ! in
INTEGER*4    data_type                             ! out
INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_rentry_numelems (id, attr_num, entry_num, num_elems, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    entry_num                             ! in
INTEGER*4    num_elems                             ! out
INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_scope (id, attr_num, scope, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    scope                                 ! out
INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_zrentry (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_get_attr_zrentry_datatype (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    entry_num                             ! in
INTEGER*4    data_type                             ! out
INTEGER*4    status                                ! out

SUBROUTINE CDF_get_attr_zrentry_numelems (id, attr_num, entry_num, num_elems, status)
INTEGER*4    id                                    ! in
INTEGER*4    attr_num                              ! in
INTEGER*4    entry_num                             ! in
INTEGER*4    num_elems                             ! out
INTEGER*4    status                                ! out

```

```

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

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

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

SUBROUTINE CDF_get_compression (id, ctype, cparms, cpercent, status)
INTEGER*4    id                                ! in
INTEGER*4    ctype                            ! out
INTEGER*4    cparms(*)                        ! out
INTEGER*4    cpercent                         ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_compression_info (cdf_name, compress_type, compress_parms,
1                                     compress_size, decompress_size, status)
CHARACTER    cdf_name*(*)                     ! in
INTEGER*4    compress_type                     ! out
INTEGER*4    compress_parms(*)                ! out
INTEGER*8    compress_size                     ! out
INTEGER*8    decompress_size                  ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_copyright (id, copyright, status)
INTEGER*4    id                                ! in
CHARACTER    copyright*(*)                    ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_datatype_size (data_type, size, status)
INTEGER*4    data_type                         ! in
INTEGER*4    size                              ! out
INTEGER*4    status                            ! out

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

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

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

```

```

SUBROUTINE CDF_get_lib_copyright (copyright, status)
CHARACTER  copyright*(*)                ! out
INTEGER*4  status                        ! out

SUBROUTINE CDF_get_lib_version (version, release, increment, sub_increment, status)
INTEGER*4  version                       ! out
INTEGER*4  release                       ! out
INTEGER*4  increment                     ! out
CHARACTER  sub_increment*(*)            ! out
INTEGER*4  status                        ! out

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

SUBROUTINE CDF_get_name (id, name, status)
INTEGER*4  id                            ! in
CHARACTER  name*(*)                      ! out
INTEGER*4  status                        ! out

SUBROUTINE CDF_get_negtoposfp0_mode (id, negtoposfp0, status)
INTEGER*4  id                            ! in
INTEGER*4  negtoposfp0                  ! out
INTEGER*4  status                        ! out

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

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

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

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

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

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

```



```

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

SUBROUTINE CDF_get_status_text (statusid, text, status)
INTEGER*4    statusid                          ! in
CHARACTER    text*(*)                          ! out
INTEGER*4    status                            ! out

INTEGER*4 FUNCTION CDF_get_var_num (id, var_name)
INTEGER*4    id                                ! in
INTEGER*4    var_name*(*)                      ! in

SUBROUTINE CDF_get_vars_maxwrittenrecnums (id, max_rvars_recnum,
1                                          max_zvars_recnum, status)
INTEGER*4    id                                ! in
INTEGER*4    max_rvars_recnum                  ! out
INTEGER*4    max_zvars_recnum                  ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_version (id, version, release, increment, status)
INTEGER*4    id                                ! in
INTEGER*4    version                          ! out
INTEGER*4    release                          ! out
INTEGER*4    increment                        ! out
INTEGER*4    status                            ! out

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

SUBROUTINE CDF_get_zvar_allocrecs (id, var_num, num_recs, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    num_recs                         ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_zvar_blockingfactor (id, var_num, bf, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    bf                                ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_zvar_cachesize (id, var_num, num_buffers, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    num_buffers                       ! out
INTEGER*4    status                            ! out

SUBROUTINE CDF_get_zvar_compression (id, var_num, compress_type, compress_parms,
1                                          compress_percent, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    compress_type                     ! out

```

```

INTEGER*4    compress_parms(*)           ! out
INTEGER*4    compress_percent            ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_data (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_get_zvar_datatype (id, var_num, data_type, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    data_type                     ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_dimsizes (id, var_num, dim_sizes, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    dim_sizes(*)                 ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_dimvariances (id, var_num, dim_varys, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    dim_varys(*)                 ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_maxallocrecnum (id, var_num, rec_num, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    rec_num                       ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_maxwrittenrecnum (id, var_num, rec_num, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    rec_num                       ! out
INTEGER*4    status                       ! out

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

SUBROUTINE CDF_get_zvar_numdims (id, var_num, num_dims, status)
INTEGER*4    id                           ! in
INTEGER*4    var_num                       ! in
INTEGER*4    num_dims                     ! out
INTEGER*4    status                       ! out

SUBROUTINE CDF_get_zvar_numelems (id, var_num, num_elems, status)
INTEGER*4    id                           ! in

```

```

INTEGER*4    var_num                ! in
INTEGER*4    num_elems              ! out
INTEGER*4    status                 ! out

SUBROUTINE CDF_get_zvar_numrecs (id, var_num, num_recs, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    num_recs              ! out
INTEGER*4    status                ! out

SUBROUTINE CDF_get_zvar_padvalue (id, var_num, pad_value, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
<type>      pad_value              ! out
INTEGER*4    status                ! out

SUBROUTINE CDF_get_zvar_recorddata (id, var_num, rec_num, record_data, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    rec_num               ! in
<type>      record_data            ! out
INTEGER*4    status                ! out

SUBROUTINE CDF_get_zvar_recvariance (id, var_num, rec_vary, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    rec_vary              ! out
INTEGER*4    status                ! out

SUBROUTINE CDF_get_zvar_reservepercent (id, var_num, reserve_percent, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    reserve_percent        ! out
INTEGER*4    status                ! out

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

SUBROUTINE CDF_get_zvar_seqpos (id, var_num, rec_num, indices, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    rec_num               ! out
INTEGER*4    indices(*)            ! out
INTEGER*4    status                ! out

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

SUBROUTINE CDF_get_zvar_sparserecords (id, var_num, srecords, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in

```

```

INTEGER*4   srecords           ! out
INTEGER*4   status            ! out

```

```

SUBROUTINE CDF_get_zvars_reorddata (id, num_var, var_nums, rec_num,
1                                     buffer, status)

```

```

INTEGER*4   id                ! in
INTEGER*4   num_var           ! in
INTEGER*4   var_nums(*)       ! in
INTEGER*4   rec_num           ! in
<type>     buffer            ! out
INTEGER*4   status            ! out

```

```

SUBROUTINE CDF_hyper_get_zvar_data (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_hyper_put_zvar_data (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_inquire_attr (id, attr_num, attr_name, attr_scope, max_gentry,
1                             max_reentry, max_zentry, status)

```

```

INTEGER*4   id                ! in
INTEGER*4   attr_num          ! in
CHARACTER   attr_name*(*)     ! out
INTEGER*4   attr_scope        ! out
INTEGER*4   max_gentry        ! out
INTEGER*4   max_reentry       ! out
INTEGER*4   max_zentry        ! out
INTEGER*4   status            ! out

```

```

SUBROUTINE CDF_inquire_attr_gentry (id, attr_num, entry_num, data_type, num_elements,
1                                     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\_inquire\_attr\_rentry (id, attr\_num, entry\_num, data\_type, num\_elements,  
1 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\_inquire\_attr\_zentry (id, attr\_num, entry\_num, data\_type, num\_elements,  
1 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\_inquire\_cdf (id, num\_dims, dim\_sizes, encoding, majority, max\_rrec,  
1 num\_rvars, max\_zrec, num\_zvars, 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\_rrec ! out  
INTEGER\*4 num\_rvars ! out  
INTEGER\*4 max\_zrec ! out  
INTEGER\*4 num\_zvars ! out  
INTEGER\*4 num\_attrs ! out  
INTEGER\*4 status ! out

SUBROUTINE CDF\_inquire\_zvar (id, var\_num, var\_name, data\_type, num\_elements, num\_dims,  
1 dim\_sizes, rec\_variance, dim\_variances, status)

INTEGER\*4 id ! in  
INTEGER\*4 var\_num ! in  
CHARACTER var\_name\*(CDF\_VAR\_NAME\_LEN256) ! out  
INTEGER\*4 data\_type ! out  
INTEGER\*4 num\_elements ! out  
INTEGER\*4 num\_dims ! out  
INTEGER\*4 dim\_sizes(CDF\_MAX\_DIMS) ! out  
INTEGER\*4 rec\_variance ! out  
INTEGER\*4 dim\_variances(CDF\_MAX\_DIMS) ! out  
INTEGER\*4 status ! out

SUBROUTINE CDF\_open\_cdf (CDF\_name, id, status)

CHARACTER CDF\_name\*(\*) ! in  
INTEGER\*4 id ! out  
INTEGER\*4 status ! out

SUBROUTINE CDF\_put\_attr\_gentry (id, attr\_num, entry\_num, data\_type, num\_elements, value,  
1 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_put_attr_rentry (id, attr_num, entry_num, data_type, num_elements, value,
1                               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_put_attr_zentry (id, attr_num, entry_num, data_type, num_elements, value,
1                               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_put_zvar_data (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_put_zvar_recorddata (id, var_num, rec_num, values, status)
INTEGER*4    id                        ! in
INTEGER*4    var_num                   ! in
INTEGER*4    rec_num                   ! in
<type>      values                     ! in
INTEGER*4    status                    ! out

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

SUBROUTINE CDF_put_zvars_recorddata (id, num_var, var_nums, rec_num,
1                                   buffer, status)
INTEGER*4    id                        ! in
INTEGER*4    num_var                   ! in
INTEGER*4    var_nums(*)               ! in
INTEGER*4    rec_num                   ! in
<type>      buffer                     ! in
INTEGER*4    status                    ! out

```

```

SUBROUTINE CDF_rename_attr (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_rename_zvar (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_set_attr_gentry_dataspec (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                        ! in
INTEGER*4    data_type                        ! in
INTEGER*4    num_elems                        ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_set_attr_reentry_dataspec (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                        ! in
INTEGER*4    data_type                        ! in
INTEGER*4    num_elems                        ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_set_attr_scope (id, attr_num, scope, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    scope                            ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_set_attr_zentry_dataspec (id, attr_num, entry_num, data_type, status)
INTEGER*4    id                                ! in
INTEGER*4    attr_num                          ! in
INTEGER*4    entry_num                        ! in
INTEGER*4    data_type                        ! in
INTEGER*4    num_elems                        ! in
INTEGER*4    status                            ! out

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

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

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

```

```

SUBROUTINE CDF_set_compression (id, compress_type, compress_parms, status)
INTEGER*4    id                                ! in
INTEGER*4    compress_type                    ! in
INTEGER*4    compress_parms(*)                ! in
INTEGER*4    status                            ! out

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

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

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

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

SUBROUTINE CDF_set_negtoposfp0_mode (id, negtoposfp0, status)
INTEGER*4    id                                ! in
INTEGER*4    negtoposfp0                      ! in
INTEGER*4    status                            ! out

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

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

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

SUBROUTINE CDF_set_zvar_allocblockrecs (id, var_num, start_rec, end_rec, status)
INTEGER*4    id                                ! in
INTEGER*4    var_num                          ! in
INTEGER*4    start_rec                        ! in
INTEGER*4    end_rec                          ! in
INTEGER*4    status                            ! out

SUBROUTINE CDF_set_zvar_allocrecs (id, var_num, num_recs, status)
INTEGER*4    id                                ! in

```



```

INTEGER*4    var_num                ! in
INTEGER*4    num_recs               ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_blockingfactor (id, var_num, bf, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    bf                    ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_cachesize (id, var_num, num_buffers, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    num_buffers           ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_compression (id, var_num, compress_type, compress_parms, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    compress_type         ! in
INTEGER*4    compress_parms(*)     ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_dataspec (id, var_num, data_type, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    data_type             ! in
INTEGER*4    num_elems             ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_dimvariances (id, var_num, dimvarys, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    dimvarys(*)          ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_initialrecs (id, var_num, num_recs, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    num_recs             ! in
INTEGER*4    status                 ! out

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

SUBROUTINE CDF_set_zvar_recvariance (id, var_num, rec_vary, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num               ! in
INTEGER*4    rec_vary             ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_reservepercent (id, var_num, reserve_percent, status)
INTEGER*4    id                    ! in

```

```

INTEGER*4    var_num                ! in
INTEGER*4    reserve_percent        ! in
INTEGER*4    status                 ! out

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

SUBROUTINE CDF_set_zvar_seqpos (id, var_num, rec_num, indices, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num                ! in
INTEGER*4    rec_num                ! in
INTEGER*4    indices(*)             ! in
INTEGER*4    status                 ! out

SUBROUTINE CDF_set_zvar_sparserecords (id, var_num, sparse_records, status)
INTEGER*4    id                    ! in
INTEGER*4    var_num                ! in
INTEGER*4    sparse_records         ! in
INTEGER*4    status                 ! out

```

## B.3 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_LEN256)	! 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_CHECKSUM_	INTEGER*4	checksum	! 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*8	c_size	! out
	INTEGER*8	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_LEN256)	! 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_RECVAR_	INTEGER*4	rec_vary	! out
rVAR_SEQDATA_	<type>	value	! out
rVAR_SPARSEARRAYS_	INTEGER*4	s_arrays_type	! out
	INTEGER*4	a_arrays_parms(CDF_MAX_PARMS)	! out
	INTEGER*4	a_arrays_pct	! out
rVAR_SPARSERECORDS_	INTEGER*4	s_records_type	! out
rVARs_DIMSIZES_	INTEGER*4	dim_sizes(CDF_MAX_DIMS)	! 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_DATA_	<type>	value	! out
zENTRY_DATATYPE_	INTEGER*4	data_type	! 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_LEN256)	! 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	a_arrays_parms(CDF_MAX_PARMS)	! out
	INTEGER*4	a_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_			
OPEN_			
CDF_	CHARACTER	CDF_name*(*)	! in
	INTEGER*4	id	! out
PUT_			
ATTR_NAME_	CHARACTER	attr_name*(*)	! in
ATTR_SCOPE_	INTEGER*4	scope	! in
CDF_CHECKSUM_	INTEGER*4	checksum	! in
CDF_COMPRESSION_	INTEGER*4	cType	! 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	numRecords	! in
rVAR_BLOCKINGFACTOR_	INTEGER*4	blockingFactor	! in
rVAR_COMPRESSION_	INTEGER*4	cType	! 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_RECVALY_	INTEGER*4	rec_vary	! in
rVAR_SEQDATA_	<type>	value	! in
rVAR_SPARSEARRAYS_	INTEGER*4	s_arrays_type	! in
	INTEGER*4	a_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	numRecords	! in
zVAR_BLOCKINGFACTOR_	INTEGER*4	blockingFactor	! in
zVAR_COMPRESSION_	INTEGER*4	cType	! 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_RECVALY_	INTEGER*4	rec_vary	! in
zVAR_SEQDATA_	<type>	value	! in
zVAR_SPARSEARRAYS_	INTEGER*4	s_arrays_type	! in
	INTEGER*4	a_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_DIMCOUNTS_	INTEGER*4	counts(*)	! in
zVAR_DIMINDICES_	INTEGER*4	indices(*)	! in
zVAR_DIMINTERVALS_	INTEGER*4	intervals(*)	! in
zVAR_NAME_	CHARACTER	var_name*(*)	! 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.4 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*4 epoch;           ! out

SUBROUTINE EPOCH_breakdown (epoch, year, month, day, hour, minute, second, msec)
REAL*4 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, epString)
REAL*8 epoch;           ! in
CHARACTER epString*(EPOCH_STRING_LEN); ! out

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

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

SUBROUTINE encode_EPOCH3 (epoch, epString)
REAL*8 epoch;           ! in
CHARACTER epString*(EPOCH3_STRING_LEN); ! out

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

SUBROUTINE parse_EPOCH (epString)
CHARACTER epString*(EPOCH_STRING_LEN); ! in
REAL*8 epoch;           ! out

SUBROUTINE parse_EPOCH1 (epString)
CHARACTER epString*(EPOCH1_STRING_LEN); ! in
REAL*8 epoch;           ! out
```

```

SUBROUTINE parse_EPOCH2 (epString)
CHARACTER epString*(EPOCH2_STRING_LEN);           ! in
REAL*8    epoch;                                   ! out

SUBROUTINE parse_EPOCH3 (epString)
CHARACTER epString*(EPOCH3_STRING_LEN);           ! in
REAL*8    epoch;                                   ! out

SUBROUTINE compute_EPOCH16 (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*4    epoch(2);                                ! out

SUBROUTINE EPOCH16_breakdown (epoch, year, month, day, hour, minute, second, msec)
REAL*4    epoch(2);                                ! 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_EPOCH16 (epoch, epString)
REAL*8    epoch(2);                                ! in
CHARACTER epString*(EPOCH16_STRING_LEN);          ! out

SUBROUTINE encode_EPOCH16_1 (epoch, epString)
REAL*8    epoch(2);                                ! in
CHARACTER epString*(EPOCH16_1_STRING_LEN);        ! out

SUBROUTINE encode_EPOCH16_2 (epoch, epString)
REAL*8    epoch(2);                                ! in
CHARACTER epString*(EPOCH16_2_STRING_LEN);        ! out

SUBROUTINE encode_EPOCH16_3 (epoch, epString)
REAL*8    epoch(2);                                ! in
CHARACTER epString*(EPOCH16_3_STRING_LEN);        ! out

SUBROUTINE encode_EPOCH16_x (epoch, format, epString)
REAL*8    epoch(2);                                ! in
CHARACTER format*(EPOCHx_FORMAT_MAX);            ! in
CHARACTER epString*(EPOCHx_STRING_MAX);          ! out

SUBROUTINE parse_EPOCH16 (epString)
CHARACTER epString*(EPOCH16_STRING_LEN);          ! in
REAL*8    epoch(2);                                ! out

SUBROUTINE parse_EPOCH16_1 (epString)
CHARACTER epString*(EPOCH16_1_STRING_LEN);        ! in

```

```
REAL*8      epoch(2);                                ! out

SUBROUTINE parse_EPOCH16_2 (epString)
CHARACTER   epString*(EPOCH16_2_STRING_LEN);        ! in
REAL*8      epoch;                                   ! out

SUBROUTINE parse_EPOCH16_3 (epString)
CHARACTER   epString*(EPOCH16_3_STRING_LEN);        ! in
REAL*8      epoch(2);                                ! out
```



# Index

ALPHAOSF1_DECODING .....	18
ALPHAOSF1_ENCODING .....	16
ALPHAVMSd_DECODING .....	18
ALPHAVMSd_ENCODING .....	16
ALPHAVMSg_DECODING .....	18
ALPHAVMSg_ENCODING .....	16
ALPHAVMSi_DECODING .....	18
ALPHAVMSi_ENCODING .....	16
Attribute	
gEntry	
Number of Elements	
accessing .....	174
Attribute	
gEntry	
Data Type	
accessing .....	173
name	
inquiring .....	178
attributes	
numbering	
inquiring .....	178
attributes	
creating .....	29, 167, 229
current .....	214
confirming .....	220
selecting	
by name .....	272
by number .....	272
deleting .....	232
entries	
current .....	214, 215
confirming .....	222, 223, 226
selecting	
by name .....	274, 277
by number .....	274, 277
data specification	
changing .....	36, 259, 267
data type	
inquiring .....	31, 239, 241, 249
number of elements	
inquiring .....	31, 240, 241, 249
deleting .....	232, 233
existence, determining .....	223, 226
maximum	
inquiring .....	34, 193, 234
number of	
inquiring .....	235
reading .....	32, 239, 241, 249
writing .....	36, 258, 259, 264
existence, determining .....	220
gEntries	
data specification	
changing .....	200, 205
writing .....	200
naming .....	23, 30, 167
inquiring .....	34, 194, 234
renaming .....	38, 257
number of	
inquiring .....	238
numbering .....	14
inquiring .....	35, 234
numberof	
inquiring .....	48
rEntries	
data specification	
changing .....	201
writing .....	201
rEntry	
reading .....	182
scopes	
changing .....	257
constants .....	21
GLOBAL_SCOPE .....	21
VARIABLE_SCOPE .....	21
inquiring .....	34, 193, 235
zEntries	
data specification	
changing .....	203
writing .....	203
zEntry	
reading .....	187
Attributes	
deleting .....	168
gEntries	
data specification	
data type	
inquiring .....	195
number of elements	
inquiring .....	195
number of	
inquiring .....	179
reading .....	171
gEntry	
deleting .....	169
Maximum entry .....	175
name	
renaming .....	204
number of	
global attributes	
inquiring .....	191
inquiring .....	90, 191
variable attributes	
inquiring .....	192
rEntries	
data specification	
data type	

inquiring .....	197	ALPHAVMSi_DECODING.....	18
number of elements		DECSTATION_DECODING.....	18
inquiring .....	197	HOST_DECODING.....	17
number of		HP_DECODING .....	18
inquiring.....	180	IBMRS_DECODING.....	18
rEntry		MAC_DECODING.....	18
data specification		NETWORK_DECODING.....	17
changing.....	206	NeXT_DECODING.....	18
data type		PC_DECODING .....	18
inquiring.....	184	SGi_DECODING.....	18
deleting .....	169	SUN_DECODING.....	18
Maximum entry.....	176	VAX_DECODING .....	17
number of elements		selecting.....	272
inquiring.....	185	read-only	
scope		confirming.....	222
changing.....	207	constants	
inquiring.....	186	READONLYoff.....	22
zEntries		READONLYon.....	22
data specification		selecting.....	22, 273
data type		zMode	
inquiring.....	198	confirming.....	222
number of elements		constants	
inquiring.....	198	zMODEoff.....	22
number of		zMODEon1.....	22
inquiring.....	181	zMODEon2.....	22
zEntry		selecting.....	22, 273
data specification		shared CDF library.....	9
changing.....	208	standard interface (New).....	69
data type		standard interface (Original).....	29
inquiring.....	188	version	
deleting .....	170	inquiring.....	241
Maximum entry.....	177	CDF\$LIB.....	5
number of elements		cdf.inc.....	13
inquiring.....	189	CDF_get_stage_cachesize.....	88
CDF		CDF_attr_create.....	29, 167
backward file.....	23	CDF_attr_entry_inquire.....	31
backward file flag		CDF_attr_get.....	32
getting.....	25	CDF_attr_inquire.....	34
setting.....	24	CDF_ATTR_NAME_LEN256.....	23
Checksum.....	25	CDF_attr_num.....	35
Checksum mode		CDF_attr_put.....	36
setting.....	26, 27	CDF_attr_rename.....	38
copyright		CDF_BYTE.....	15
inquiring.....	81	CDF_CHAR.....	15
CDF library		CDF_close.....	39
copy right notice		CDF_close_cdf.....	73
max length.....	23	CDF_close_zvar.....	104
reading.....	240	CDF_confirm_attr_existence.....	163
internal interface.....	211	CDF_confirm_gentry_existence.....	164
modes		CDF_confirm_reentry_existence.....	165
-0.0 to 0.0		CDF_confirm_zentry_existence.....	166
confirming.....	221	CDF_confirm_zvar_existence.....	105
constants		CDF_confirm_zvar_padvalue_existence.....	105
NEGtoPOSfp0off.....	22	CDF_COPYRIGHT_LEN.....	23
NEGtoPOSfp0on.....	22	CDF_create.....	40
selecting.....	273	CDF_create_cdf.....	74
decoding		CDF_create_zvar.....	106
confirming.....	221	CDF_delete.....	41
constants		CDF_delete_attr.....	168
ALPHAOSF1_DECODING.....	18	CDF_delete_attr_gentry.....	169
ALPHAVMSd_DECODING.....	18	CDF_delete_attr_reentry.....	169
ALPHAVMSg_DECODING.....	18	CDF_delete_attr_zentry.....	170

CDF_delete_cdf.....	76	CDF_get_zvar_maxwrittenrecnum.....	123
CDF_delete_zvar.....	109	CDF_get_zvar_name.....	123
CDF_delete_zvar_recs.....	109	CDF_get_zvar_numdims.....	124
CDF_doc.....	42	CDF_get_zvar_numelems.....	125
CDF_DOUBLE.....	15	CDF_get_zvar_numrecs.....	126
CDF_EPOCH.....	15	CDF_get_zvar_padvalue.....	127
CDF_EPOCH16.....	15	CDF_get_zvar_recorddata.....	128
CDF_error.....	44	CDF_get_zvar_recvariance.....	129
CDF_error or CDF_error.....	299	CDF_get_zvar_reservepercent.....	130
CDF_FLOAT.....	15	CDF_get_zvar_seq.....	131
CDF_get_attr_gentry.....	171	CDF_get_zvar_seqpos.....	132
CDF_get_attr_gentry_datatype.....	173	CDF_get_zvar_sparserecords.....	134
CDF_get_attr_max_gentry.....	174	CDF_get_zvars_maxwrittenrecnum.....	133
CDF_get_attr_max_rentry.....	176	CDF_get_zvars_recorddata.....	135
CDF_get_attr_max_zentry.....	177	CDF_getrvarsrecorddata.....	45
CDF_get_attr_name.....	178	CDF_getzvarsrecorddata.....	46
CDF_get_attr_num.....	178	CDF_hyper_get_zvar_data.....	137
CDF_get_attr_num_gentries.....	179	CDF_hyper_put_zvar_data.....	139
CDF_get_attr_num_rentries.....	180	CDF_inquire.....	48
CDF_get_attr_num_zentries.....	181	CDF_inquire_attr.....	193
CDF_get_attr_rentry.....	182	CDF_inquire_attr_gentry.....	195
CDF_get_attr_rentry_datatype.....	184	CDF_inquire_attr_rentry.....	197
CDF_get_attr_rentry_numelems.....	185	CDF_inquire_attr_zentry.....	198
CDF_get_attr_scope.....	186	CDF_inquire_cdf.....	90
CDF_get_attr_zentry.....	187	CDF_inquire_zvar.....	141
CDF_get_attr_zentry_datatype.....	188	CDF_INT1.....	15
CDF_get_attr_zentry_numelems.....	189	CDF_INT2.....	15
CDF_get_cachesize.....	76	CDF_INT4.....	15
CDF_get_checksum.....	77	CDF_lib.....	211
CDF_get_compress_cachesize.....	78	CDF_LIB.....	6
CDF_get_compression.....	79	CDF_MAX_DIMS.....	23
CDF_get_compression_info.....	80	CDF_MAX_PARAMS.....	23
CDF_get_copyright.....	81	CDF_OK.....	14
CDF_get_datatype_size.....	70	CDF_open.....	50
CDF_get_decoding.....	82	CDF_open_cdf.....	92
CDF_get_encoding.....	83	CDF_PATHNAME_LEN.....	23
CDF_get_format.....	84	CDF_put_attr_gentry.....	200
CDF_get_lib_copyright.....	71	CDF_put_attr_rentry.....	201
CDF_get_lib_version.....	71	CDF_put_attr_zentry.....	203
CDF_get_majority.....	84	CDF_put_zvar_data.....	142
CDF_get_name.....	85	CDF_put_zvar_recorddata.....	144
CDF_get_negtoposfp0_mode.....	86	CDF_put_zvar_seqdata.....	145
CDF_get_num_attrs.....	191	CDF_put_zvars_recorddata.....	146
CDF_get_num_gattrs.....	191	CDF_putrvarsrecorddata.....	51
CDF_get_num_vattrs.....	192	CDF_putzvarsrecorddata.....	53
CDF_get_num_zvars.....	110	CDF_REAL4.....	15
CDF_get_readonly_mode.....	87	CDF_REAL8.....	15
CDF_get_status_text.....	72	CDF_rename_attr.....	204
CDF_get_var_num.....	111	CDF_rename_zvar.....	148
CDF_get_vars_maxwrittenrecnums.....	113	CDF_set_attr_gentry_dataspec.....	205
CDF_get_version.....	89	CDF_set_attr_rentry_dataspec.....	206
CDF_get_zmode.....	90	CDF_set_attr_scope.....	207
CDF_get_zvar_allocrecs.....	113	CDF_set_attr_zentry_dataspec.....	208
CDF_get_zvar_blockingfactor.....	114	CDF_set_blockingfactor.....	151
CDF_get_zvar_cachesize.....	115	CDF_set_cachesize.....	93
CDF_get_zvar_compression.....	116	CDF_set_checksum.....	94
CDF_get_zvar_data.....	117	CDF_set_compression.....	96
CDF_get_zvar_datatype.....	119	CDF_set_compression_cachesize.....	95
CDF_get_zvar_dimsizes.....	120	CDF_set_decoding.....	97
CDF_get_zvar_dimvariances.....	121	CDF_set_encoding.....	98
CDF_get_zvar_maxallocrecnum.....	122	CDF_set_format.....	98
		CDF_set_majority.....	99

CDF_set_negtoposfp0_mode.....	100	reading.....	42, 236
CDF_set_readonly_mode.....	101	corrupted.....	40, 74
CDF_set_stage_cachesize.....	102	creating.....	40, 74, 229
CDF_set_zmode.....	103	current.....	214
CDF_set_zvar_allocblockrecs.....	149	confirming.....	221
CDF_set_zvar_allocrecs.....	150	selecting.....	272
CDF_set_zvar_cachesize.....	152	decoding	
CDF_set_zvar_compression.....	153	inquiring.....	82
CDF_set_zvar_dataspec.....	154	resetting.....	97
CDF_set_zvar_dimvariances.....	155	deleting.....	41, 76, 232
CDF_set_zvar_initialrecs.....	156	encoding	
CDF_set_zvar_padvalue.....	157	changing.....	258
CDF_set_zvar_recvariance.....	158	constants.....	16
CDF_set_zvar_reservepercent.....	159	ALPHAOSF1_ENCODING.....	16
CDF_set_zvar_seqpos.....	161	ALPHAVMSd_ENCODING.....	16
CDF_set_zvar_sparserecords.....	162	ALPHAVMSg_ENCODING.....	16
CDF_set_zvars_cachesize.....	160	ALPHAVMSi_ENCODING.....	16
CDF_STATUSTEXT_LEN.....	23	DECSTATION_ENCODING.....	17
CDF_UCHAR.....	15	HOST_ENCODING.....	16
CDF_UINT1.....	15	HP_ENCODING.....	17
CDF_UINT2.....	15	IBMRS_ENCODING.....	17
CDF_UINT4.....	15	MAC_ENCODING.....	17
CDF_var_close.....	55	NETWORK_ENCODING.....	16
CDF_var_create.....	56	NeXT_ENCODING.....	17
CDF_var_get.....	58	PC_ENCODING.....	17
CDF_var_hyper_get.....	59	SGi_ENCODING.....	16
CDF_var_hyper_put.....	61	SUN_ENCODING.....	16
CDF_var_inquire.....	63	VAX_ENCODING.....	16
CDF_VAR_NAME_LEN256.....	23	default.....	16
CDF_var_num.....	65	inquiring.....	48, 83, 90, 236
CDF_var_put.....	66	resetting.....	98
CDF_var_rename.....	67	format	
CDF_WARN.....	14	changing.....	258
CDFs		constants	
accessing.....	50, 92, 221	MULTI_FILE.....	14
browsing.....	22	SINGLE_FILE.....	14
cache buffers		default.....	14
confirming.....	221, 222, 224, 226, 227	inquiring.....	84
selecting.....	272, 274, 275, 277, 280	inquiring.....	236
cache size		resetting.....	98
inquiring.....	76	majority	
resetting.....	93	inquiring.....	84
stage		resetting.....	99
resetting.....	102	mode	
staging		postoposfp0	
inquiring.....	88	resetting.....	100
checksum		read only	
inquiring.....	77	resetting.....	101
checksum		name	
reading.....	236, 257	inquiring.....	85
resetting.....	94	naming.....	23, 40, 50, 75, 92
closing.....	39, 73, 220	negtoposfp0 mode	
compression		inquiring.....	86
cache size		nulling.....	256
inquiring.....	78	opening.....	50, 92, 256
resetting.....	95	overwriting.....	40, 74
inquiring.....	79, 80, 236, 242, 250	readonly mode	
resetting.....	96	inquiring.....	87
specifying.....	257	scratch directory	
compression types/parameters.....	20	specifying.....	273
copy right notice		status	
max length.....	23	test	



inquiring.....	72
version	
inquiring.....	42, 89, 237, 239
zmode	
resetting.....	103
zMode	
inquiring.....	90
zVariables	
records	
maximum written.....	133
Cchecksum.....	77, 94
COLUMN_MAJOR.....	19
Compiling.....	1
compression	
CDF	
inquiring.....	236, 237
specifying.....	257
types/parameters.....	20
variables	
inquiring.....	242, 250
reserve percentage	
confirming.....	224, 228
selecting.....	275, 279
specifying.....	260, 268
compute_EPOCH.....	288
compute_EPOCH16.....	293
confirm	
existence	
attribute.....	163
gEntry.....	164
rEntry.....	165
zEntry.....	166
zVariable.....	105
padValue.....	105
data type	
size	
inquiring.....	70
data types	
constants.....	14
CDF_BYTE.....	15
CDF_CHAR.....	15
CDF_DOUBLE.....	15
CDF_EPOCH.....	15
CDF_EPOCH16.....	15
CDF_FLOAT.....	15
CDF_INT1.....	15
CDF_INT2.....	15
CDF_INT4.....	15
CDF_REAL4.....	15
CDF_REAL8.....	15
CDF_UCHAR.....	15
CDF_UINT1.....	15
CDF_UINT2.....	15
CDF_UINT4.....	15
inquiring size.....	239
DECSTATION_DECODING.....	18
DECSTATION_ENCODING.....	17
definitions file.....	5
DEFINITIONS.COM.....	5
dimensions	
limit.....	23
numbering.....	14

encode_EPOCH.....	289
encode_EPOCH1.....	289
encode_EPOCH16.....	294
encode_EPOCH16_1.....	294
encode_EPOCH16_2.....	295
encode_EPOCH16_3.....	295
encode_EPOCH16_x.....	295
encode_EPOCH2.....	290
encode_EPOCH3.....	290
encode_EPOCHx.....	291
EPOCH	
computing.....	288
decomposing.....	289
encoding.....	289, 290, 291
parsing.....	292, 293, 297
utility routines.....	288
compute_EPOCH.....	288
compute_EPOCH16.....	293
encode_EPOCH.....	289
encode_EPOCH1.....	289
encode_EPOCH16.....	294
encode_EPOCH16_1.....	294
encode_EPOCH16_2.....	295
encode_EPOCH16_3.....	295
encode_EPOCH16_x.....	295
encode_EPOCH2.....	290
encode_EPOCH3.....	290
encode_EPOCHx.....	291
EPOCH_breakdown.....	289
EPOCH16_breakdown.....	293
parse_EPOCH.....	292
parse_EPOCH1.....	292
parse_EPOCH16.....	296
parse_EPOCH16_1.....	297
parse_EPOCH16_2.....	297
parse_EPOCH16_3.....	297
parse_EPOCH2.....	292
parse_EPOCH3.....	293
EPOCH_breakdown.....	289
EPOCH16	
computing.....	293
decomposing.....	293
encoding.....	294, 295
parsing.....	296, 297
EPOCH16_breakdown.....	293
examples	
accessing	
Attribute	
rEntry	
Maximum entry.....	176
zEntry	
Maximum entry.....	177
accessing	
Attribute	
gEntry	
Data Type.....	173
Maximum entry.....	175
Number of Elements.....	174
allocating	
zVariable	
records.....	150, 151
changing	

attribute		attribute.....	34, 194
rEntry		entry.....	31
data specification .....	207	gEntry .....	196
scope.....	207	number.....	36, 179
zEntry		rEntry.....	197
data specification .....	209	zEntry.....	199
CDF		Attribute	
cache size .....	94	rEntry	
stage.....	102	number of elements .....	184, 185
checksum.....	94	scope .....	186
compression.....	96	zEntry	
cache size.....	95	data type.....	189
decoding.....	97	number of elements .....	190
encoding.....	98	Attributes	
format.....	99	gEntries.....	180
majority.....	100	number of attributes .....	191
mode		number of gAttributes.....	192
negtoposfp0.....	101	number of vAttributes.....	193
read only.....	101	rEntries.....	181
zmode.....	103	zEntries.....	182
zVariable.....	154	CDF.....	43, 49, 91
attribute		cache size.....	77
data specification.....	205	checksum.....	78
zVariable		compression.....	80, 81
blocking factor.....	152	cache size.....	79
cache size.....	153	copyright.....	81
data specification.....	155	decoding.....	82
dimension variances .....	156	encoding.....	83
record variance.....	158	format.....	84
reserve percentage.....	159	majority.....	85
sparse records.....	162	name.....	86
closing		negtoposfp0 mode.....	86
CDF.....	39, 74	number of zVariables.....	111
rVariable.....	55, 56	readonly mode.....	87
zVariable.....	104	staging cache size.....	88
confirm		version.....	89
existence		zMode.....	90
gEntry.....	164	zVariables	
rEntry.....	165	records	
zEntry.....	166	maximum written.....	133
zVariable.....	105	data type	
padValue.....	106	size.....	70
confirm		error code explanation text.....	44, 73
existence		library	
attribute.....	163	copyright.....	71
creating		Library	
attribute.....	30, 167	version.....	72
CDF.....	41, 75, 211	rVariable.....	64
rVariable.....	57, 280	variable	
zVariable.....	107, 281	number.....	65
deleting		Variable	
Attribute.....	168	number.....	112
gEntry.....	169	Variables	
rEntry.....	170	records	
zEntry.....	171	maximum written.....	113
CDF.....	42, 76	zVariable.....	142
zVariable.....	109	allocated records.....	114
records.....	110	blocking factor.....	115
get		cache size.....	116
Attribute		compression.....	117
name.....	178	data type.....	119
inquiring		dimension sizes .....	120

dimension variances .....	121	rVariables full record.....	52
name .....	124	zVariable values	
number of dimensions .....	125	full record .....	144
number of elements .....	126	hyper .....	140
record variance.....	130	multiple variable.....	284
records		sequential .....	145
maximum allocated.....	122	single.....	143
maximum written .....	123	zVariables.....	53, 146
written.....	127	zVariables full record .....	54, 147
reserve percentage .....	130	GLOBAL_SCOPE.....	21
sequential position .....	133	HOST_DECODING .....	17
sparse records type .....	134	HOST_ENCODING .....	16
Internal Interface.....	211, 280	HP_DECODING .....	18
interpreting		HP_ENCODING.....	17
status codes.....	286	IBMRS_DECODING .....	18
opening		IBMRS_ENCODING .....	17
CDF.....	51, 93	interfaces	
reading		Internal .....	211
attribute		Standard (New).....	69
gEntry.....	172	Standard (Original).....	29
rEntry .....	183	Internal Interface .....	211
zEntry.....	188	currnt objects/states.....	214
attribute entry .....	33	attribute.....	214
rVariable values		attribute entries.....	214, 215
hyper.....	60, 281	CDF .....	214
single.....	59	records/dimensions .....	215, 216, 217
rVariables.....	45	sequential value.....	216, 217
rVariables full record .....	45	status code .....	217
zVariable		variables .....	214
pad value .....	128	examples .....	211, 280
zVariable values		Indentation/Style .....	218
full record.....	129	Operations.....	220
hyper.....	138	status codes, returned .....	217
sequential .....	131, 283	syntax .....	218
single.....	118	argument list.....	218
zVariables .....	46, 135	limitations.....	218
zVariables full record .....	47, 135	item referencing .....	14
renaming		libcdf.a.....	6
attribute .....	204	LIBCDF.OLB.....	5, 6
attributes .....	38, 283	Library	
rVariable .....	68	copyright	
zVariable.....	149	inquiring .....	71
resetting		version	
zVariable		inquiring .....	71
pad value .....	158	limits	
setting		attribute name .....	23
zVariable		copyright text .....	23
sequential position .....	161	dimensions .....	23
setting		explanation/status text.....	23
zVariables		file name .....	23
cache size .....	160	parameters.....	23
status handler.....	286	variable name.....	23
writing		linking.....	5
attribute		shareable CDF library .....	9
gEntry.....	37, 201	MAC_DECODING.....	18
rEntry .....	37, 202, 284	MAC_ENCODING.....	17
zEntry.....	203	MULTI_FILE.....	14
zVariable.....	157	NEGtoPOSfp0off.....	22
rVariable values		NEGtoPOSfp0on.....	22
hyper.....	62	NETWORK_DECODING .....	17
single .....	67	NETWORK_ENCODING .....	16
rVariables.....	51	NeXT_DECODING.....	18

NeXT_ENCODING	17	types	21
NO_COMPRESSION	20	Standard Interface (New)	69
NO_SPARSEARRAYS	21	Standard Interface (Original)	29
NO_SPARSERECORDS	21	status codes	
NOVARY	20	constants	14, 286
PAD_SPARSERECORDS	21	CDF_OK	14
parse_EPOCH	292	CDF_WARN	14
parse_EPOCH1	292	current	217
parse_EPOCH16	296	confirming	222
parse_EPOCH16_1	297	selecting	273
parse_EPOCH16_2	297	error	299
parse_EPOCH16_3	297	explanation text	
parse_EPOCH2	292	inquiring	44, 248
parse_EPOCH3	293	max length	23
PC_DECODING	18	explanation text	299
PC_ENCODING	17	informational	299
PREV_SPARSERECORDS	21	interpreting	286, 299
programming interface	13	status handler, example	284
compiling	1	warning	299
linking	5	SUN_DECODING	18
READONLYoff	22	SUN_ENCODING	16
READONLYon	22	VARIABLE_SCOPE	21
ROW_MAJOR	19	variables	
rVariables		aparse arrays	
closing	55	inquiring	247, 255, 263, 271
creating	56	closing	104, 220
data specification		compression	
data type		confirming	224, 228
inquiring	63	inquiring	236, 242, 250
number of elements		selecting	275, 279
inquiring	63	specifying	260, 268
dimensionality		types/parameters	20
inquiring	48, 90	creating	230, 231
full record		current	214
reading	45	confirming	224, 227
writing	51	selecting	
multiple values		by name	275, 278
accessing	59	by number	274, 277
writing	61	data specification	
naming		changing	261, 268
inquiring	63	data type	
renaming	67	inquiring	243, 251
number of		number of elements	
inquiring	49	inquiring	245, 254
records		deleting	232, 233
maximum		dimension counts	
inquiring	49	current	215, 217
single value		confirming	225, 227
accessing	58	selecting	276, 278
writing	66	dimension indices, starting	
scratch directory		current	215, 216
specifying	273	confirming	225, 227
SGi_DECODING	18	selecting	276, 278
SGi_ENCODING	16	dimension intervals	
SINGLE_FILE	14	current	216, 217
sparse arrays		confirming	225, 227
inquiring	247, 255	selecting	276, 278
specifying	263, 271	dimensionality	
types	21	inquiring	248, 253
sparse records		existence, determining	224, 228
inquiring	247, 255	indices	
specifying	264, 271	numbering	14

majority	
changing	258
considering	19
constants	19
COLUMN_MAJOR	19
ROW_MAJOR	19
default	230
inquiring	237
naming	56, 107
inquiring	244, 252
max length	23
renaming	262, 269
number of, inquiring	238
numbering	14
inquiring	245, 253
pad value	
confirming	224, 228
inquiring	246, 254
specifying	262, 270
reading	243, 251
record count	
current	215, 216
confirming	225, 228
selecting	276, 278
record interval	
current	215, 216
confirming	226, 228
selecting	276, 279
record number, starting	
current	215, 216
confirming	226, 228
selecting	277, 279
records	
allocated	
inquiring	242, 245, 249, 250, 253
specifying	260, 267
blocking factor	
inquiring	242, 250
specifying	260, 268
deleting	233
indexing	
inquiring	244, 252
initial	
writing	262, 269
maximum	
inquiring	244, 247, 252, 255
number of	
inquiring	246, 254
numbering	14
sparse	21
inquiring	247, 255
specifying	264, 271
sparse arrays	
types	21
variances	
constants	19
NOVARY	20
VARY	19
dimensional	
inquiring	243, 251
specifying	261, 269
record	
changing	263, 270
inquiring	246, 254
writing	262, 269
Variables	
number of	
inquiring	90
numbering	
inquiring	65, 111
records	
maximum	
inquiring	90
maximum written	
inquiring	113
VARY	19
VAX_DECODING	17
VAX_ENCODING	16
zMODEoff	22
zMODEon1	22
zMODEon2	22
zVariables	
records	
allocating	149
zVariables	
accessing	
full record	128
hyper values	137
sequential value	131
single value	117
blocking factor	
inquiring	114
resetting	151
cache size	
inquiring	115
resetting	152, 160
compression	
inquiring	116
resetting	153
creating	106
data specification	
data type	
inquiring	141
number of elements	
inquiring	141
resetting	154
data type	
inquiring	119
deleting	109
dimension sizes	
inquiring	120
dimension variances	
inquiring	121
resetting	155
full record	
reading	46
writing	53
name	
inquiring	123
renaming	148
naming	
inquiring	141
number of	
inquiring	110

number of dimensions		maximum written	
inquiring.....	124	inquiring .....	123
number of elements		written	
inquiring.....	125	inquiring .....	126
pad value		written initially.....	156
accessing.....	127	reserve percentage	
resetting.....	157	inquiring .....	130
reading		resetting.....	159
full record .....	135	sequential position	
record variance		inquiring .....	132
inquiring.....	129	setting .....	161
resetting.....	158	sparse records type	
records		inquiring .....	134
allocated		resetting .....	162
inquiring.....	113	writing.....	146
allocation .....	150	full record .....	144
deleting .....	109	hyper values .....	139
maximum allocated		sequential value.....	145
inquiring.....	122	single value .....	142