```
  "Latitude"        CDF_REAL4       1          F         F T

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "VALIDMIN"      CDF_REAL4    { -90.0 }
    "VALIDMAX"      CDF_REAL4    { 90.0 } .

  ! NRV values follow...

  [ 1, 1 ] = 40.0
  [ 1, 2 ] = 30.0


! Variable            Data       Number     Record    Dimension
! Name                Type      Elements   Variance   Variances
! --------            ----      --------   --------   ---------

  "Temperature"     CDF_REAL4       1          T         T T

  ! Attribute         Data
  ! Name              Type       Value
  ! --------          ----       -----

    "VALIDMIN"      CDF_REAL4    { -40.0 }
    "VALIDMAX"      CDF_REAL4    { 50.0 } .

#end
```

Assuming that **CDFskeleton** was used to create a CDF containing the metadata and data in the above skeleton table, the following Fortran program would be used to complete the creation of the CDF.

```
        PROGRAM exampleSKT

C-----------------------------------------------------------------------
C
C  NSSDC/CDF                   Create an example CDF (using skeleton table).
C
C  Version 1.0, 5-Jan-94, CDF, Inc.
C
C  Modification history:
C
C   V1.0   5-Jan-94, Joe Programmer      Original version.
C
C-----------------------------------------------------------------------

        INCLUDE '../../include/cdf.inc'
```

```
        INTEGER*4 id            ! CDF identifier.
        INTEGER*4 status        ! CDF completion status.

        INTEGER*4 lun           ! Logical unit number for input data file.

        INTEGER*4 indices(2)    ! Dimension indices.
        INTEGER*4 rec_num       ! Record number.

        INTEGER*4 time_var_num  ! 'Time' rVariable number.
        INTEGER*4 tmp_var_num   ! 'Temperature' rVariable number.

        INTEGER*4 time          ! 'Time' rVariable value.
        REAL*4    lat           ! 'Latitude' rVariable value.
        REAL*4    lon           ! 'Longitude' rVariable value.
        REAL*4    tmp           ! 'Temperature' rVariable value.

        DATA lun/1/

C------------------------------------------------------------------------
C Open the CDF.
C------------------------------------------------------------------------

        CALL CDF_open ('example2', id, status)
        IF (status .NE. CDF_OK) CALL StatusHandler (status)

C------------------------------------------------------------------------
C Determine rVariable numbers.
C------------------------------------------------------------------------

        time_var_num = CDF_var_num (id, 'Time')
        IF (time_var_num .LT. CDF_OK) CALL StatusHandler (status)

        tmp_var_num = CDF_var_num (id, 'Temperature')
        IF (tmp_var_num .LT. CDF_OK) CALL StatusHandler (status)

C------------------------------------------------------------------------
C Read input values for rVariables and write them to the CDF.  Not
C every value must be written to the CDF - many of the values are redundant.
C The 'Time' value only has to be written once per CDF record (every 4 input
C records).  The 'Longitude' and 'Latitude' values are not written at all
C because they had been specified in the skeleton table.  Each 'Temperature'
C value read is written to the CDF.
C------------------------------------------------------------------------

        OPEN (lun, FILE='example.dat', ERR=99)

        DO rec_num = 1, 24
          DO x1 = 1, 2
            DO x2 = 1, 2
              indices(1) = x1
              indices(2) = x2
```

```
              READ (lun, *, ERR=99) time, lon, lat, tmp

              IF (indices(1) .EQ. 1 .AND. indices(2) .EQ. 1) THEN
                CALL CDF_var_put (id, time_var_num, rec_num, indices,
     .                            time, status)
                IF (status .NE. CDF_OK) CALL StatusHandler (status)
              END IF

              CALL CDF_var_put (id, tmp_var_num, rec_num, indices,
     .                          tmp, status)
              IF (status .NE. CDF_OK) CALL StatusHandler (status)
            END DO
          END DO
        END DO

        CLOSE (lun, ERR=99)

C-------------------------------------------------------------------------------
C Close CDF.
C-------------------------------------------------------------------------------

        CALL CDF_close (id, status)
        IF (status .NE. CDF_OK) CALL StatusHandler (status)

        STOP

C-------------------------------------------------------------------------------
C Input file error handler.
C-------------------------------------------------------------------------------

 99     WRITE (6,101)
 101    FORMAT (' ','Error reading input file')
        STOP

        END



C-------------------------------------------------------------------------------
C Status handler.
C-------------------------------------------------------------------------------

        SUBROUTINE StatusHandler (status)
        INTEGER*4 status

        INCLUDE '../../include/cdf.inc'

        CHARACTER message*(CDF_ERRTEXT_LEN)

        IF (status .LT. CDF_WARN) THEN
```

```
          WRITE (6,10)
10        FORMAT (' ','Error (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...')
            CALL CDF_error (status, message)
            WRITE (6,13) message
13          FORMAT (' ',A)
          ELSE
            IF (status .GT. CDF_OK) THEN
              WRITE (6,14)
14            FORMAT (' ','Be advised that...')
              CALL CDF_error (status, message)
              WRITE (6,15) message
15            FORMAT (' ',A)
            END IF
          END IF
        END IF

        RETURN
        END
```

The CDF was opened (since it already existed) and the values for only the `Time` and `Temperature` rVariables were written to the CDF. All of the other functions performed by the program in Section 1.9.1 were done by the `CDFskeleton` program when it read the skeleton table.

Chapter 2

General CDF Concepts

2.1 CDF Library

The CDF library is the only way to access a CDF. Various properties of the CDF library are described in the following sections.

2.1.1 Interfaces

Several interfaces to the CDF library exist. They are described in the following sections.

Standard Interface

The Standard Interface[1] provides a standard set of routines with which to access a CDF. Not all CDF features are available with the Standard Interface. The Internal Interface must be used to perform operations not available with the Standard Interface routines (e.g., access to zVariables). The Standard Interface is callable from both C and Fortran applications. Table 2.1 lists the routines available when using the Standard Interface. Each routine is described in detail in the corresponding programming reference chapter.

Internal Interface

The Internal Interface may be used to perform all supported CDF operations. The Internal Interface must be used to perform those operations not available with the Standard Interface. Table 2.2 lists the routines available when using the Internal Interface. Each is described in detail in the corresponding programming reference chapter.

[1] The Standard Interface was previously known as the C/Fortran Interface(s). That was an unfortunate choice of terms since the Internal Interface is also callable from C and Fortran.

Callable from C	Callable from Fortran	Purpose
CDFcreate()	CDF_create()	Creates a new CDF.
CDFopen()	CDF_open()	Opens an existing CDF.
CDFdoc()	CDF_doc()	Inquires version/release and copyright notice.
CDFinquire()	CDF_inquire()	Inquires rVariable dimensionality, encoding, etc.
CDFclose()	CDF_close()	Closes a CDF.
CDFdelete()	CDF_delete()	Deletes a CDF.
CDFerror()	CDF_error()	Inquires error (status) code meaning.
CDFvarCreate()	CDF_var_create()	Creates an rVariable.
CDFvarNum()	CDF_var_num()	Determines an rVariable number.
CDFvarRename()	CDF_var_rename()	Renames an rVariable.
CDFvarInquire()	CDF_var_inquire()	Inquires about an rVariable.
CDFvarPut()	CDF_var_put()	Writes an rVariable value.
CDFvarGet()	CDF_var_get()	Reads an rVariable value.
CDFvarHyperPut()	CDF_var_hyper_put()	Writes one or more rVariable values.
CDFvarHyperGet()	CDF_var_hyper_get()	Reads one or more rVariable values.
CDFattrCreate()	CDF_attr_create()	Creates an attribute.
CDFattrNum()	CDF_attr_num()	Determines an attribute number.
CDFattrRename()	CDF_attr_rename()	Renames an attribute.
CDFattrInquire()	CDF_attr_inquire()	Inquires about an attribute.
CDFattrEntryInquire()	CDF_attr_entry_inquire()	Inquires about an attribute entry.
CDFattrPut()	CDF_attr_put()	Writes an attribute entry.
CDFattrGet()	CDF_attr_get()	Reads an attribute entry.

Table 2.1: Standard Interface Routines

Callable from C	Callable from Fortran	Purpose
CDFlib ()	CDF_lib ()	Performs all available operations.

Table 2.2: Internal Interface Routines

Obsolete Interface, VAX/VMS Systems Only

Fortran applications written for CDF Version 1 on VAX/VMS systems may still be used, without modification, via the Obsolete Interface. While we don't encourage the use of the Obsolete Interface, it is provided for use while existing applications are being updated for the new Standard Interface routines. To use the Obsolete Interface, simply relink the application with the Obsolete CDF library (LIBCDF_OBSOLETE.OLB). An example of this would be as follows:

```
$ LINK file1, file2, ...fileN, CDF$LIB:LIBCDF_OBSOLETE/LIBRARY
```

where file1, file2, ...fileN are the application's object modules. CDF$LIB is defined by executing the definitions file, DEFINITIONS.COM, provided with the CDF distribution.

There are several conditions that apply to the use of the Obsolete Interface.

1. The functions (and their associated arguments) available for use are exactly the same as in CDF Version

1 (refer to the *CDF Version 1 Implementer's Guide* for details). New CDF features are not available.

2. CDFs created will be Version 2 CDFs. To create a Version 1 CDF you must use the CDF Version 1 library.

3. Version 2 CDFs may be accessed except in those cases where the Obsolete Interface does not support a new Version 2 feature. In those cases an error code indicating the incompatibility will be returned. These incompatibilities deal mainly with attribute entries. In Version 1 the data type was specified for the attribute — in Version 2 the data type is specified per entry.

4. Version 1 CDFs may be accessed but the only modification allowed is the updating of existing variable values. No other modifications are allowed to Version 1 CDFs (with the Version 2 CDF library).

CDF's IDL Interface

The CDF distribution contains an interface that allows full access to the CDF library (and hence CDFs) from within IDL. CDF's IDL interface consists of a set of functions that mirror the functions in the Standard and Internal interfaces for C and Fortran applications. CDF's IDL interface is described in Appendix B.

IDL also provides its own interface to the CDF library (as well as other data formats) that differs from CDF's IDL interface. The differences are mainly syntactic with the functionality of the two interfaces being essentially the same. IDL's documentation describes their built-in CDF interface. Another difference between the two interfaces is that CDF's IDL interface is only available on those computers that support dynamic linking. Appendix B lists the computers on which this is the case.

2.1.2 CDF Modes

Once a CDF has been opened (or created and not yet closed), the CDF library may be configured to act on that CDF in one or more modes. These modes are specified independently for each open CDF.

Read-Only Mode

A CDF may be placed in read-only mode via the Internal Interface using the `<SELECT_,CDF_READONLY_MODE_>` operation. Only read access will be allowed on the CDF — all attempts to modify the CDF will fail. A CDF may be toggled in and out of read-only mode any number of times. (Note that attempts to modify a CDF may also fail if insufficient access privileges exist for the CDF — the file system enforces this access.)

zMode

A CDF may be placed into zMode[2] via the Internal Interface using the `<SELECT_,CDF_zMODE_>` operation. When in zMode a CDF's rVariables essentially disappear and are replaced by corresponding zVariables.[3] Likewise, the rEntries for a vAttribute become zEntries (because they are now associated with zVariables). While in zMode most operations involving rVariables/rEntries will fail. (Some inquiry operations will be

[2] There are actually two types of zMode — read on.

[3] In a future release of CDF, support for rVariables will be eliminated. zMode is provided to ease the transition from rVariables to the more flexible zVariables. rVariables are essentially a subset of zVariables.

allowed. For example, inquiring the number of rVariables [always zero] is allowed.) When zMode is used, the number of variables remains the same — rVariables simply change into zVariables. Note that the existing contents of the CDF are not changed — the CDF simply appears different.

Each new zVariable has the same exact properties as the corresponding (hidden) rVariable except for dimensionality and variances. The data specification (data type and number of elements), pad value, etc. stay the same. The dimensionality/variances of each zVariable are dependent on which zMode is currently being used: zMode/1 or zMode/2. In zMode/1 the dimensionality/variances stay exactly the same. In zMode/2, however, those dimensions with a false variance (`NOVARY`) are eliminated. Consider a CDF with an rVariable dimensionality of `2:[180,360]`[4] containing the following rVariables.

rVariable Name	Variances
EPOCH	T/FF[5]
LATITUDE	T/TF
LONGITUDE	T/FT
HUMIDITY	T/TT

If this CDF were to be placed into zMode/1, the following zVariables would replace the existing rVariables.

zVariable Name	Dimensionality	Variances
EPOCH	2:[180,360]	T/FF
LATITUDE	2:[180,360]	T/TF
LONGITUDE	2:[180,360]	T/FT
HUMIDITY	2:[180,360]	T/TT

Note that the dimensionality of each zVariable is the same as it was for the rVariables in the CDF. However, if zMode/2 were used, the following zVariables would replace the existing rVariables.

zVariable Name	Dimensionality	Variances
EPOCH	0:[]	T/
LATITUDE	1:[180]	T/T
LONGITUDE	1:[360]	T/T
HUMIDITY	2:[180,360]	T/TT

In this case the false dimensional variances were removed (which decreased the dimensionality in several of the variables).

A CDF can be placed into or taken out of zMode any number of times while it is open. Each time the zMode is changed for a CDF, it would be best to think of the CDF as being closed and reopened in that zMode. The numbering of variable/entries may or may not be as you would expect (and the scheme used could change in a future release of CDF). Most applications will simply select a zMode immediately after opening a CDF. (zMode being off is the default if a zMode is not selected.)

NOTE: Using zMode does not change the contents of a CDF. A CDF containing rVariables will appear to contain only zVariables when in zMode. If the same CDF is then opened without using zMode, the rVariables will still exist.

[4] This notation is used throughout this document. In this case there are two dimensions whose sizes are `180` and `360`. A dimensionality of zero is represented as `0:[]`.

[5] This notation is also used throughout this document. The record variance is before the slash and the dimension variances are after the slash. In this case the record variance is true (`VARY`) and the dimension variances are both false (`NOVARY`). The variances for a 0-dimensional variable would be represented as `T/` (if the record variance were true).

-0.0 to 0.0 Mode

The floating-point value -0.0 is legal on those computers which use the IEEE 754 floating-point representation (e.g., most UNIX-based computers and the IBM PC) but is illegal on VAXes and DEC Alphas running OpenVMS. Attempting to use -0.0 results in a reserved operand fault on a VAX and a high performance arithmetic fault on a DEC Alpha running OpenVMS. Because of this the CDF library can be told to convert -0.0 to 0.0 when read from a CDF. The values physically stored in the CDF file(s) are not modified — only the values returned to an application are converted. This mode is available on all supported computers but is only really necessary on VAXes and DEC Alphas running OpenVMS. The CDF library is told to convert -0.0 to 0.0 for a CDF via the Internal Interface using the <SELECT_,CDF_NEGtoPOSfp0_MODE_> operation.

2.1.3 Limits

Open CDFs

The only limit on the number of CDFs that may be open simultaneously is the operating system's limit on the number of open files that an application may have. Each open CDF will always have at least one associated open file (the .cdf file). The CDF library will open and close the variable files of a multi-file CDF as needed (see Sections 2.3.2 and 2.3.3).

2.1.4 Caching Scheme

The CDF library reads and writes to open files in 512-byte blocks. A cache of 512-byte memory buffers is maintained by the CDF library for each open file. The CDF library attempts to keep in the cache the set of file blocks currently being accessed. This results in fewer actual I/O operations to the file if repeated accesses to these blocks would occur. When the cache is completely full and a new block of the file is accessed, one of the cache buffers is written back to the file (if it was modified) and the new block is read into that cache buffer (unless the file is being extended in which case the cache buffer is simply cleared). This process is known as paging. By optimizing the number of cache buffers for a file, improved performance can be achieved. There is tradeoff between having too few cache buffers and having too many. Having too few cache buffers will cause excessive paging while having too many cache buffers may slow performance because of the overhead involved in maintaining the cache.

The CDF library attempts to choose optimal default cache sizes based on a CDF's format and number of variables. This is difficult since the CDF library does not know how an application will access a CDF. For that reason an application may specify, via the Internal Interface, the number of cache buffers to be used for a file. The number of cache buffers may be changed as many times as necessary while a file is open (the first time will override the default used by the CDF library). The default numbers of cache buffers used by the CDF library in various situations are as follows:

Created CDF, single-file (.cdf)	MAXcacheCREATEDsingleCDF
Created CDF, multi-file (.cdf)	MAXcacheCREATEDmultiCDF
Opened CDF, single-file (.cdf)	The smaller of: "<total-number-of-variables> + PLUScacheOPENEDsingle" and MAXcacheOPENEDsingleCDF
Opened CDF, multi-file (.cdf)	MAXcacheOPENEDmultiCDF

Variable files (multi-file)	MAXcacheACCESSmultiVAR

The values of the constants shown may be configured for your CDF distribution when it is built and installed. Consult your system manager for the values of these defaults. Note that these values are the maximum number of cache buffers for a file. The cache buffers are only actually allocated as needed.

The situations in which it will be necessary to specify a cache size will depend on how a CDF is accessed. For example, consider a variable in a row-major CDF having a dimensionality of `2:[10,64]`, a data specification of `CDF_REAL8/1`, and variances of `T/TT`. This variable definition results in each record of the variable being written across 10 file blocks with the second dimension varying the fastest (since the CDF's variable majority is row-major). If single value reads were used to access this variable (see Section 2.3.12), only one cache buffer would be necessary if the second dimension were incremented the fastest (i.e., `[1,1]`, `[1,2]`,...`[10,63]`, `[10,64]`). This is because the values of a record would be accessed sequentially from the first block to the last block. If, however, the first dimension were incremented the fastest (i.e., `[1,1]`, `[2,1]`,...`[9,64]`, `[10,64]`), 10 cache buffers would improve performance. The values of a record are not being accessed sequentially but rather each read would be from a different block. Since the reads would be spread access 10 blocks, having 10 cache buffers would be optimal.

A similar situation arises when accessing variables in a single-file CDF. If values are accessed for each variable at a record number, then performance will be improved by setting the number of cache buffers to be the same as the number of variables. This is because the variable values will most likely be located in that many different file blocks for a particular record number.

In a C application cache sizes are specified using the `<SELECT_,CDF_CACHESIZE_>`, `<SELECT_,rVAR_CACHESIZE_>`, `<SELECT_,rVARs_CACHESIZE_>`, `<SELECT_,zVAR_CACHESIZE_>`, and `<SELECT_,zVARs_CACHESIZE_>` operations of the `CDFlib` function (Internal Interface). In a Fortran application cache sizes are specified using the same operations of the `CDF_lib` subroutine (Internal Interface).

2.2 CDFs

The following sections describe various aspects of a CDF.

2.2.1 Accessing

Both Version 1 and Version 2 CDFs may be accessed with the Version 2 CDF library. Restrictions are described in the following sections.

Version 2 CDFs

Any operation on a Version 2 CDF may be performed with the Standard and Internal interfaces. On VMS systems, when using the Obsolete Interface, all operations are allowed except when an incompatibility exists between the contents of the Version 2 CDF and the capabilities of the Obsolete Interface (see Section 2.1.1).

Version 1 CDFs

All types of read access are allowed on Version 1 CDFs using either the Standard or Internal Interface. The only write access allowed is the modification of existing variable values.

2.2.2 Creating

A CDF must be created by the CDF library. In a C application CDFs are created using either the `CDFcreate` function (Standard Interface) or the `<CREATE_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are created using either the `CDF_create` subroutine (Standard Interface) or the `<CREATE_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

2.2.3 Opening

An application must open an existing CDF before access to that CDF is allowed by the CDF library. In a C application CDFs are opened using either the `CDFopen` function (Standard Interface) or the `<OPEN_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are opened using either the `CDF_open` subroutine (Standard Interface) or the `<OPEN_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

2.2.4 Closing

It is absolutely essential that a CDF that has been created or modified by an application be closed before the program exits. If the CDF is not closed it will in most cases be corrupted and unreadable. This is because the cache buffers maintained by the CDF library will not have been written to the CDF file(s). An existing CDF that has been opened and only read from should also be closed. In a C application CDFs are closed using either the `CDFclose` function (Standard Interface) or the `<CLOSE_,CDF_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application CDFs are closed using either the `CDF_close` subroutine (Standard Interface) or the `<CLOSE_,CDF_>` operation of the `CDF_lib` subroutine (Internal Interface).

2.2.5 Naming

The pathname specified when opening or creating a CDF can be any legal pathname for the operating system being used. This includes logical symbols on VMS systems and environment variables on UNIX systems. Trailing blanks are also allowed but will be ignored. This is so Fortran applications do not have to be concerned with the trailing blanks of a Fortran `CHARACTER` variable. (C character string variables use terminating `NUL` characters.)

In almost all cases when a CDF pathname is specified, the `.cdf` extension should not be appended. (It will be appended automatically by the CDF library.) The exception to this is when a user has renamed an existing CDF with a different extension or with no extension (for whatever reason). When a CDF is opened, the CDF library first appends the `.cdf` extension to the pathname specified and then checks to see if that file

exists.[6] If not, the CDF library will also check to see if a file exists whose pathname is exactly as specified (without `.cdf` appended). If this is the case, the CDF must be single-file. If the CDF is multi-file, an error occurs since the CDF library would have no idea as to how the variable files had been renamed. Note also that the CDF library always appends `.cdf` to the pathname specified when creating a CDF.

NOTE: The CDF toolkit programs will in some cases not recognize a CDF if it does not have an extension of `.cdf`.[7]

2.2.6 Format

There are two CDF formats: multi-file and single-file. The choice of which format to use will depend on how the CDF is to be accessed. Note that the `CDFconvert` toolkit program can be used to change the format of an existing CDF (creating a new CDF).

The default format for a created CDF was determined when your CDF distribution was built and installed. Consult your system manager for this default. In a user application, the Internal Interface must be used to change the format of a CDF. The format of an existing CDF can be changed only if no variables have been created in the CDF. If the `CDFskeleton` toolkit program is used to create a CDF, the format is specified in the skeleton table (see Section 3.9).

In a C application a CDF's format is changed by using the `<PUT_,CDF_FORMAT_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application a CDF's format is changed by using the `<PUT_,CDF_FORMAT_>` operation of the `CDF_lib` subroutine (Internal Interface).

Multi-File CDFs

A multi-file CDF (`MULTI_FILE`) consists of one file (with extension `.cdf`) containing control information and attribute entry data and a separate file for each variable defined in the CDF (with extensions `.v0`,`.v1`,... for rVariables and `.z0`,`.z1`,... for zVariables). Each variable file contains the data values for the corresponding variable. (The control information for each variable is stored in the `.cdf` file.)

Single-File CDFs

A single-file CDF (`SINGLE_FILE`) consists of only one file (with extension `.cdf`). This file contains the control information for the entire CDF, the attribute entry data, and all of the variable data. An indexing scheme is used to provide efficient access to variable records.

Indexing Scheme. In single-file CDFs an indexing scheme is used to keep track of where a variable's records are located within the `.cdf` file. The order that variable (and attribute entry) values are written to a single-file CDF by an application may result in a variable's records being noncontiguous. There will be groups of contiguous records, but these groups will not be contiguous. This section will give a brief overview of how a variable's records are indexed.

[6] Actually, the CDF library will check several possible extensions: `.cdf`, `.cdf;1`, `.CDF`, and `.CDF;1`. These extensions are checked because some CD-ROM drivers (primarily on UNIX machines) do peculiar things when making the files (CDFs) on a CD-ROM visible.

[7] Or `.cdf;1` or `.CDF` or `.CDF;1`.

For each variable in a single-file CDF one or more index records will exist. Each of these index records will contain one or more index entries.[8] An index entry consists of the following fields:

`starting record`	The number of the first record in a group of contiguous records.
`ending record`	The number of the last record in a group of contiguous records.
`byte offset`	The byte offset within the `.cdf` file of the group of contiguous records.

To find a particular variable record the CDF library must search through the index entries for that variable. Improved performance will result if there are fewer index entries to search. This can be achieved by having a large number of records in each group of contiguous records (resulting in fewer groups/index entries). Sections 2.3.7 and 2.3.7 describe techniques that can be used to achieve this.

It is possible to inquire the indexing statistics for a variable. Using the Internal Interface, an application may inquire the number of index records and total number of index entries.

Performance

The most efficient access to CDF variables will usually occur when the CDF has the multi-file format. The extra overhead involved with the indexing scheme used in single-file CDFs is small, so the difference may not be significant (especially if hyper reads/writes are used). The drawback to using the multi-file format is that more than one file is associated with a CDF (which may or may not be a problem for your system management).

There is a case in which the single-file format may be more efficient. If a CDF has a large number of variables (larger than the number of files that may be open at once by an application) and the variables values are accessed variable-by-variable (rather than accessing an entire variable before going to the next variable), the multi-file format may be much slower than the single-file format. This is because the CDF library will have to close one variable file and then open another as each variable value is accessed by the application (since the operating system's open file limit will be reached). If the application was to access every value for a variable before going on to the next variable, this would not occur (but it might create complications for your application).

Note that the format of a CDF can always be converted using the `CDFconvert` toolkit program (which creates a new CDF with the specified format). Section 3.4 describes `CDFconvert`.

2.2.7 Encoding

The encoding of a CDF determines how attribute entry data and variable data values are stored on disk in the CDF file(s). An application program never has to concern itself with the encoding of the CDF being accessed. The CDF library performs all of the encoding and decoding of data values for the application.

A CDF's encoding is specified when the CDF is created. The encoding of an existing CDF may be changed with the Internal Interface if no variable values or attribute entries have been written (variables and attributes

[8] Actually, all but the last index record will be completely full of index entries. The last index record will contain one or more index entries.

may exist, however). If the `CDFskeleton` toolkit program is used to create a CDF the encoding is specified in the skeleton table (see Section 3.9).

The encoding specified when creating a CDF may be either the host encoding of the computer being used or network encoding. A CDF cannot be created with the host encoding of a different computer. A CDF with any encoding, however, may be read on any computer.[9]

Host Encoding

Host encoding (`HOST_ENCODING`) specifies that variable and attribute entry data values be written to the CDF in the native encoding of the computer being used. The following host encodings are supported:

`VAX_ENCODING`	VAX and microVAX computers running VMS. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation.
`ALPHAVMSd_ENCODING`	DEC Alpha computers running OpenVMS. Double-precision floating-point values are encoded in Digital's `D_FLOAT` representation.
`ALPHAVMSg_ENCODING`	DEC Alpha computers running OpenVMS. Double-precision floating-point values are encoded in Digital's `G_FLOAT` representation.
`SUN_ENCODING`	Sun computers.
`SGi_ENCODING`	Silicon Graphics Iris and Power Series computers.
`DECSTATION_ENCODING`	DECstation computers.
`ALPHAOSF1_ENCODING`	DEC Alpha computers running OSF/1.
`IBMRS_ENCODING`	IBM RS6000 series computers.
`HP_ENCODING`	HP 9000 series computers.
`IBMPC_ENCODING`	IBM PC personal computers.
`NeXT_ENCODING`	NeXT computers.

Note that any host encoding may be read on any computer supported by CDF.

Network Encoding

Network encoding (`NETWORK_ENCODING`) specifies that variable and attribute entry data values be written to the CDF in the XDR (External Data Representation) format. As values are written to the CDF, the CDF library encodes them into network encoding. Network encoded CDFs are readable on any machine supported by CDF.

[9] This is a change from previous releases of CDF.

Equivalent Encodings

While an encoding parameter exists for each supported computer, not every encoding is different. The following sections describe which computers use the same encoding for the various data types.

Character/1-Byte Integer Data Types. Since each supported computer uses the ASCII character set and orders the bits in a byte the same way, the character and 1-byte integer data types (`CDF_CHAR`, `CDF_UCHAR`, `CDF_BYTE`, `CDF_INT1`, and `CDF_UINT1`) are encoded in the same way on each.

Multiple-Byte Integer Data Types. The multiple-byte integer data types (`CDF_INT2`, `CDF_UINT2`, `CDF_INT4`, and `CDF_UINT4`) are encoded in one of two ways: big-endian or little-endian. Big-endian has the least significant byte (LSB) in the highest memory location while little-endian has the LSB in the lowest memory location. The supported computers use big-endian or little-endian as shown in Table 2.3. Network (XDR) encoding uses big-endian encoding for multiple-byte integer data types.

Big-Endian	Little-Endian
Sun	VAX
SGi Iris	DECstation
IBM RS6000	IBM PC
HP 9000	DEC Alpha (OSF/1)
NeXT	DEC Alpha (OpenVmS)

Table 2.3: Equivalent Byte Orderings

Single-Precision Floating-Point Data Types. The single-precision floating-point encodings on the supported computers are either IEEE 754 floating-point or Digital's `F_FLOAT` floating-point. There are also two different byte orderings for the computers that use IEEE 754 (big-endian and little-endian). The single-precision floating-point encodings for each supported computer are shown in Table 2.4. Network (XDR) encoding uses IEEE 754 (big-endian) encoding for single-precision floating-point data types.

IEEE 754 (Big Endian)	IEEE 754 (Little Endian)	Digital's `F_FLOAT`
Sun	DECstation	VAX
SGi Iris	IBM PC	DEC Alpha/OpenVMS
IBM RS6000	DEC Alpha/OSF1	
HP 9000		
NeXT		

Table 2.4: Equivalent Single-Precision Floating-Point

Double-Precision Floating-Point Data Types. The double-precision floating-point encodings on the supported computers are either IEEE 754 floating-point, Digital's `D_FLOAT` floating-point, or Digital's `G_FLOAT` floating-point. There are also two different byte orderings for the computers that use IEEE 754 (big-endian and little-endian). The double-precision floating-point encodings for each supported computer are shown in Table 2.5. Network (XDR) encoding uses IEEE 754 (big-endian) encoding for double-precision floating-point data types.

IEEE 754 (Big Endian)	IEEE 754 (Little Endian)	Digital's D_FLOAT	Digital's G_FLOAT
Sun	DECstation	VAX	DEC Alpha/OpenVMS/G
SGi Iris	IBM PC	DEC Alpha/OpenVMS/D	
IBM RS6000	DEC Alpha/OSF1		
HP 9000			
NeXT			

Table 2.5: Equivalent Double-Precision Floating-Point

Performance

The best performance when accessing (reading or writing) a CDF will occur when that CDF is in the host encoding of the computer being used (and host decoding is in effect — see Section 2.2.8). This is because no encoding or decoding has to be performed by the CDF library. A CDF that must be portable between two or more different types of computers should normally be network encoded. There may be cases, however, where it would be desirable to create a CDF with host encoding (e.g., on a slow machine) and then transfer it to a faster machine for processing or conversion to network encoding. Obviously, there are trade-offs as to which encoding should be used in any one particular case. Keep in mind that a CDF can always be converted to the host encoding of the machine being used (with `CDFconvert`) before being accessed.

2.2.8 Decoding

The decoding of a CDF determines how attribute entry and variable data values are passed to a calling application program from the CDF library. The default decoding when a CDF is initially opened is host decoding (the native encoding of the computer being used). When host decoding is in effect, all data values read by an application are immediately ready for manipulation and display. Almost all of your applications will simply use the default of host decoding and not be concerned with selecting a decoding. There are some situations, however, where selecting a different decoding will be advantageous. Some possibilities are as follows:

1. A client/server model where a number of CDFs are maintained on a server computer (in any of the supported encodings). Clients on different type computers could request data from a CDF on the server computer. The server computer would then select a decoding for the CDF based on the client's computer type and then read the data value(s). The value(s) could then be sent directly to the client computer by the server computer without a conversion being necessary by the server computer.

2. If data values were being read from a CDF and written in binary form to a file for use on a different type computer. The proper decoding could be selected for the CDF before any of the data values are read. No conversions would be necessary by the application program.

A CDF's decoding may be selected and reselected at any time after the CDF has been opened and as many times as necessary. A CDF's decoding is selected via the Internal Interface with the `<SELECT_,CDF_DECODING_>` operation. Also, a CDF's decoding does not affect the values that already exist in a CDF or any values subsequently written. A CDF's encoding determines how the values are written to the CDF file(s). Section 2.2.7 describes a CDF's encoding.

The supported decodings correspond to the supported encodings. They are as follows:

HOST_DECODING	The data representation of the host computer.
NETWORK_DECODING	The External Data Representation (XDR).
VAX_DECODING	VAX and microVAX data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation.
ALPHAVMSd_DECODING	DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation.
ALPHAVMSg_DECODING	DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's G_FLOAT representation.
SUN_DECODING	Sun data representation.
SGi_DECODING	Silicon Graphics Iris and Power Series data representation.
DECSTATION_DECODING	DECstation data representation.
ALPHAOSF1_DECODING	DEC Alpha running OSF/1 data representation.
IBMRS_DECODING	IBM RS6000 series data representation.
HP_DECODING	HP 9000 series data representation.
IBMPC_DECODING	IBM PC data representation.
NeXT_DECODING	NeXT data representation.

Performance

The best performance when reading a CDF will occur when the CDF's decoding is the same as the CDF's encoding since no conversion will have to be performed by the CDF library. Since host decoding is the only directly usable decoding by an application, CDFs with host encoding will provide the best performance. Care should be taken when selecting the encoding for a CDF.

2.2.9 Data Types

CDF supports a variety of data types consistent with the types available with C and Fortran compilers on most computers. All data types are based on an 8-bit byte.

Integer Data Types

CDF_BYTE	1-byte, signed integer.
CDF_INT1	1-byte, signed integer.
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.

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

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

Floating Point Data Types

`CDF_REAL4 & CDF_FLOAT`
: 4-byte, single-precision floating-point.

`CDF_REAL8 & CDF_DOUBLE`
: 8-byte, double-precision floating-point.

A special case exists with respect to the value `-0.0` (negative floating-point zero). This value is legal on those computers that use the IEEE 754 floating-point representation (e.g., most UNIX-based computers and the IBM PC) but is illegal on VAXes and DEC Alphas running OpenVMS. Attempting to use `-0.0` will result in a reserved operand fault on a VAX and a high performance arithmetic fault on a DEC Alpha running OpenVMS. A warning is returned whenever `-0.0` is read by an application on a VAX or DEC Alpha running OpenVMS. The CDF library can be put into a mode where `-0.0` will be converted to `0.0` when detected (see Section 2.1.2). If `-0.0` is not being converted to `0.0`, the CDF toolkit programs are designed to display `-0.0` in all cases. This includes those computers that normally suppress the negative sign.

Character Data Types

`CDF_CHAR`	1-byte, character.
`CDF_UCHAR`	1-byte, unsigned character.

Character data types are unique for variables in that they are the only data types for which more than one element per value is allowed. Each variable value consists of a character string with the number of elements being the number of characters. More than one element is allowed for any of the data types when dealing with attribute entries.

EPOCH Data Type

`CDF_EPOCH`	8-byte, double precision floating point.

The EPOCH data type is used to store time values referenced from a particular epoch. For NSSDC applications that epoch is `01-Jan-0000 00:00:00.000`.[10] `CDF_EPOCH` values are the number of milliseconds since

[10] I know what you're thinking. The year 0 AD never existed. If it makes you feel better, think of the epoch year as 1 BC rather than 0 AD.

the epoch. CDF toolkit programs will display and expect CDF_EPOCH values in the format

```
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, AD,
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.

Functions exist that parse, encode, compute, and break down CDF_EPOCH values. These functions are described in Section 4.9 for C applications and in Section 5.10 for Fortran applications.

Equivalent Data Types

Certain data types are considered equivalent with respect to their representation in memory and in a CDF. The following groups contain equivalent data types.

CDF_CHAR	CDF_INT2	CDF_INT4	CDF_REAL4	CDF_REAL8
CDF_UCHAR	CDF_UINT2	CDF_UINT4	CDF_FLOAT	CDF_DOUBLE
CDF_INT1				CDF_EPOCH
CDF_UINT1				
CDF_BYTE				

Note that while the signed and unsigned forms of a data type are considered equivalent by the CDF library, they must be correctly interpreted by an application to produce the desired results.

2.2.10 Limits

Limits within a CDF are defined in the appropriate include files: cdf.h for C applications and cdf.inc for Fortran applications. The following limits exist:[11]

CDF_MAX_DIMS	The maximum number of dimensions that rVariables/zVariables may have.
CDF_VAR_NAME_LEN	The maximum number of characters in a variable name.
CDF_ATTR_NAME_LEN	The maximum number of characters in an attribute name.
CDF_PATHNAME_LEN	The maximum number of characters in a pathname used to specify a CDF.

Most of these limits can be raised. Contact CDF User Support if that becomes necessary.

[11] Previous releases of CDF limited the number of variables a CDF could contain. That limit has been eliminated.

2.3 Variables

CDF variables are the mechanism for storing data. (Attributes are used to store metadata.) A new variable may be created in a CDF at any time. Two types of variables are supported: rVariables and zVariables.[12] The main difference is that all rVariables in a CDF have the same dimensionality whereas zVariables can have differing dimensionalities. In the following sections differences between the two types will be noted where appropriate.

2.3.1 Accessing

The Standard Interface deals exclusively with rVariables. No access to zVariables is provided. The Internal Interface may be used to access either rVariables or zVariables.

2.3.2 Opening

The CDF library automatically opens the variable files in a multi-file CDF as the variables are accessed. An application never has to concern itself with opening variables.

The opening of variables does not apply to single-file CDFs since individual files do not exist for each variable.

2.3.3 Closing

The CDF library automatically closes the variable files in a multi-file CDF when the CDF itself is closed by an application.[13] Variable files are also closed automatically by the CDF library as other variables are accessed if insufficient file pointers exist to keep all of the variables open at once. This would be due to an open file quota enforced by the operating system being used.

A case also exists where it may be beneficial for an application to close a variable in a multi-file CDF. Since each open variable file uses some number of cache buffers, a large amount of system memory could be in use (see Section 2.1.4). This may not be a problem on VAX or UNIX machines but could result in a program crashing on an MS-DOS machine. If memory is limited, an application may want to close variables after they have been accessed in order to minimize the total number of cache buffers being used. In a C application rVariables are closed using either the `CDFvarClose` function (Standard Interface) or the `<CLOSE_,rVAR_>` operation of the `CDFlib` function (Internal Interface). zVariables are closed using the `<CLOSE_,zVAR_>` operation of the `CDFlib` function (Internal Interface). In a Fortran application rVariables are closed using either the `CDF_var_close` subroutine (Standard Interface) or the `<CLOSE_,rVAR_>` operation of the `CDF_lib` subroutine (Internal Interface). zVariables are closed using the `<CLOSE_,zVAR_>` operation of the `CDF_lib` subroutine (Internal Interface).

The closing of variables does not apply to single-file CDFs since individual files do not exist for each variable.

[12] The letters "r" and "z" don't stand for anything in particular. "r" sort of stands for "regular" since rVariables have always been supported by CDF.

[13] It is required that an application close a CDF before exiting.