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

  "Longitude"      CDF_REAL4      1          F          T F

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

    "VALIDMIN"     CDF_REAL4    { -180.0 }
    "VALIDMAX"     CDF_REAL4    { 180.0 } .

  ! NRV values follow...

  [ 1, 1 ] = -165.0
  [ 2, 1 ] = -150.0


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

  "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 `SkeletonCDF` 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 **SkeletonCDF** program when it read the skeleton table.

Chapter 2

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 programmer's guide.

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 programmer's guide.

[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.
`CDFvarClose()`	`CDF_var_close()`	Closes an rVariable.
`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

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 allowed. For example, inquiring the number of rVariables is allowed [but will always be zero].) 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.

---

[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.

[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).

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.

`-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., UNIX-based computers, the Macintosh, 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 or written to a CDF. When reading from a CDF the values physically stored in the CDF are not modified — only the values returned to an application are converted. When writing to a CDF the values physically stored are modified — `-0.0` is converted to `0.0` before being written to the CDF. 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. When this mode is disabled, a warning (`NEGATIVE_FP_ZERO`) is returned when `-0.0` is read from a CDF (and the decoding is that of a VAX or DEC Alpha running OpenVMS) or written to a CDF (and the encoding is that of a VAX or DEC Alpha running OpenVMS).

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.[6] 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). Default cache sizes may be configured for your CDF distribution when it is built and installed. Consult your system manager for the values of these defaults.

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 multi-file, 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 2.3.13), 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 greater than 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.

Cache sizes are specified using the `<SELECT_,CDF_CACHESIZE_>`, `<SELECT_,rVAR_CACHESIZE_>`, `<SELECT_,`

[6] As variables are created in a single-file CDF, the CDF library will attempt to adjust the cache size accordingly. This may be overridden by an application if desired.

rVARs_CACHESIZE_>, <SELECT_,zVAR_CACHESIZE_>, and <SELECT_,zVARs_CACHESIZE_>, operations of the Internal Interface.

NOTE: The default cache sizes used by the CDF library are fairly conservative in order to minimize the problems that can arise due to memory limitations (especially on computers having limited memory such as the IBM PC and Macintosh). If the performance of your application is critical, it is very important to experiment with using larger cache sizes. Significant gains in performance can be achieved with the proper cache size. It is also important to allocate records if you know how many are to be written. This will reduce the fragmentation that can occur in a single-file CDF (which degrades performance because of the increased indexing that occurs for a variable). Allocating records is described in Section 2.3.8.

2.2 CDFs

The following sections describe various aspects of a CDF.

2.2.1 Accessing

Only Version 2 CDFs may be accessed with the current CDF distribution. Version 1 CDFs must be converted to Version 2 CDFs using the `CDFconvert` program in a CDF distribution prior to CDF V2.5 before they will be readable.

All supported CDF operations are available using the Internal Interface. A subset of these operations are available using the Standard Interface. The Obsolete Interface is no longer supported. (Applications written for CDF Version 1 must be ported to the Standard or Internal Interface of CDF Version 2.)

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 Deleting

An open CDF may be deleted at any time. The .cdf file is deleted along with any variable files if a multi-file CDF. Note that if the CDF is corrupted and cannot be opened by the CDF library you will have to delete the CDF file(s) manually using the capabilities of the operating system being used. In a C application CDFs are deleted using either the CDFdelete function (Standard Interface) or the <DELETE_,CDF_> operation of the CDFlib function (Internal Interface). In a Fortran application CDFs are deleted using either the CDF_delete subroutine (Standard Interface) or the <DELETE_,CDF_> operation of the CDF_lib subroutine (Internal Interface).

2.2.6 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.[7] 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.[8]

[7] 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.

[8] Or .cdf;1 or .CDF or .CDF;1.

2.2.7 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 with the desired format).

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 `SkeletonCDF` toolkit program is used to create a CDF, the format is specified in the skeleton table (see Section 3.8).

A CDF's format is changed by using the `<PUT_,CDF_FORMAT_>` operation of the 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.

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.[9] 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

[9] 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.

large number of records in each group of contiguous records (resulting in fewer groups/index entries). Even better would be only one group of contiguous records per variable. Techniques used to achieve fewer index entries are described in the "Allocated Records" and "Extend Records" subsections of Section 2.3.8.

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 for rVariables and zVariables using the `<GET_,rVAR_nINDEXRECORDS>`, `<GET_,rVAR_nINDEXENTRIES>`, `<GET_,zVAR_nINDEXRECORDS>`, and `<GET_, zVAR_nINDEXENTRIES>` operations.

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 also be converted using the `CDFconvert` toolkit program (which creates a new CDF with the specified format). Section 3.4 describes `CDFconvert`.

2.2.8 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 when using the Standard Interface but is set to the default encoding for your CDF distribution when created using the Internal Interface. 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 may exist, however). If the `SkeletonCDF` toolkit program is used to create a CDF the encoding is specified in the skeleton table (see Section 3.8).

The encoding specified when creating/modifying a CDF may be any of the native encodings for the computers supported by CDF in addition to network (XDR) encoding.[10] A CDF with any supported encoding is also readable on any computer supported by CDF.

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

Host Encodings

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. In addition, the following explicit 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.
`MAC_ENCODING`	Macintosh computers.

When `HOST_ENCODING` is specified, it is translated to the actual host encoding from the above list.

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.

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)
Macintosh	

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		
Macintosh		

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			
Macintosh			

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 2.2.9). 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 another 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.

2.2.9 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 either the client or the server. The CDF library would perform the necessary conversions.
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 2.2.8 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.
`MAC_DECODING`	Macintosh 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 the host's encoding will provide the best performance. Care should be taken when selecting the encoding for a CDF.

2.2.10 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. The size of an element of a data type is the same regardless of the computer/operating system being used. The `<GET_,DATATYPE_SIZE_>` operation of the Internal Interface may be used to inquire the size in bytes of a particular data type.

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`.[11] `CDF_EPOCH` values are the number of milliseconds since the epoch. CDF toolkit programs will display and expect `CDF_EPOCH` values in the format

[11] 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.

```
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 the CDF C Reference Manual for C applications and in the CDF Fortran Reference Manual 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.

2.2.11 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:[12]

`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.

[12] Previous releases of CDF limited the number of variables a CDF could contain. That limit has been eliminated except for multi-file CDFs on an IBM PC because of the 8.3 naming convention.

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.[13] 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.[14] 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.

[13] 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.

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

2.3.4 Naming

Each variable in a CDF has a unique name. This applies to rVariables and zVariables together (i.e., an rVariable cannot have the same name as a zVariable). Variable names are case sensitive regardless of the operating system being used and may consist of up to CDF_VAR_NAME_LEN printable characters (including blanks). Trailing blanks, however, are ignored when the CDF library compares variable names. "LAT" and "LAT   " are considered to be the same name, so they cannot both exist in the same CDF. This was done because Version 1 of CDF padded variable names on the right with blanks out to eight characters. When a Version 1 CDF was converted to a Version 2 CDF these trailing blanks remained in the variable names. To allow CDF Version 2 applications to read such a CDF without having to be concerned with the trailing blanks, the trailing blanks are ignored by the CDF library when comparing variable names. The trailing blanks are returned as part of the name, however, when a variable is inquired by an application program.

2.3.5 Numbering

The rVariables in a CDF are numbered consecutively starting at one (1) for Fortran applications and starting at zero (0) for C applications. Likewise, the zVariables in a CDF are numbered consecutively starting at one (1) for Fortran applications and starting at zero (0) for C applications. The CDF library assigns variable numbers as the variables are created.

2.3.6 Deleting

A variable may be deleted from a single-file CDF.[15] Deleting a variable also causes the deletion of the corresponding attribute entries for the variable. The disk space used by the variable definition, the variable's data records, and the corresponding attribute entries becomes available for use as needed by the CDF library. Also, the variables which numerically follow the variable being deleted are renumbered immediately. (Each is decremented by one.) Variables are deleted using the <DELETE_,rVAR_> and <DELETE_,zVAR_> operations of the Internal Interface.

2.3.7 Dimensionality

Variable values are stored in arrays. A variable's dimensionality refers to the number of dimensions and the dimension sizes of these arrays.

Each rVariable in a CDF has the same dimensionality. An array of values exists for each rVariable at each record in a CDF. The values may not be physically stored but may be virtual (see Sections 2.3.8, 2.3.10, and 2.3.11).

A zVariable may have a dimensionality which is different from that of the rVariables and the other zVariables. An array of values exists for each zVariable at each record in a CDF. As with rVariables the values may not be physically stored but may be virtual. zVariables are intended for use in those situations where using an rVariable would waste disk space or not logically make sense.

A variable array having two or more dimensions also contains subarrays. For instance, in a 3-dimensional array with dimension sizes [10,20,30], each array consists of ten 2-dimensional subarrays of size [20,30],

[15] Variables may not currently be deleted from a multi-file CDF.

and each of those 2-dimensional subarrays consists of twenty 1-dimensional subarrays of size `[30]`. Subarrays will be referred to when discussing other properties of CDF variables.

2.3.8 Records

A CDF record is a set of variable arrays, one per rVariable and one per zVariable in the CDF. The variable arrays in a particular record are generally related to each other in some way (often time). This does not have to be the case and is not enforced by the CDF library in any way. A variable record is simply the corresponding variable array within a CDF record.

Physical variable records are actually stored in the CDF file(s). Virtual variable records are not actually stored but do exist in the conceptual view of the variable provided by CDF. Virtual records can occur in a CDF because of the following reasons:

1. If a variable's values do not vary from record to record (record variance of `NOVARY`), all of that variable's records beyond the first one are virtual and have the same values as the first record (only the first record is physically stored). If a record has not yet been written to that variable, then all of its records are virtual and contain the pad value for that variable.

2. If a variable's values do vary from record to record (record variance of `VARY`), then the records beyond the last record actually written are virtual and contain the pad value for that variable.

Record variance is described in Section 2.3.10.

The maximum record written is maintained by the CDF library for each variable in the CDF. The "maximum CDF record" is simply the maximum rVariable record written (of all the rVariables). This quantity is available through the Standard Interface when inquiring about a CDF. Because the Standard Interface does not allow access to zVariables, zVariables are not considered when determining the "maximum CDF record." The "maximum CDF record" would be used by applications dealing only with rVariables. The maximum record written for each rVariable and zVariable is available via the Internal Interface.

Figure 2.1 illustrates the relationships between physical and virtual variable records. Variable 1 has five records that were physically written. Only two records were physically written to variable 2 so the following records are virtual (containing the pad value for that variable). Only one record can be physically written to variable 3 because its record variance is `NOVARY`. The other records are virtual and contain the same values as the first record. Because a record has not been physically written to variable 4, all of its records are virtual containing the pad value for that variable. Likewise, since no records have been written to variable 5, all of its records are also virtual and contain the pad value for that variable. Note that if these are rVariables, the "maximum CDF record" would be five (when displayed by a toolkit program or in a Fortran application — it would be four in a C application).

Note that a variable's records do not have to be written sequentially starting at the first record. The records may be written in any order. For a variable with `VARY` record variance, if a new record more than one record beyond the current maximum record for a variable is written, the intervening records will be physically written and contain the pad value for that variable. Also, when one or more values are written to a new physical record, the entire record is physically written with the pad value for the variable being used for the unspecified values. Variable pad values are described in Section 2.3.17.

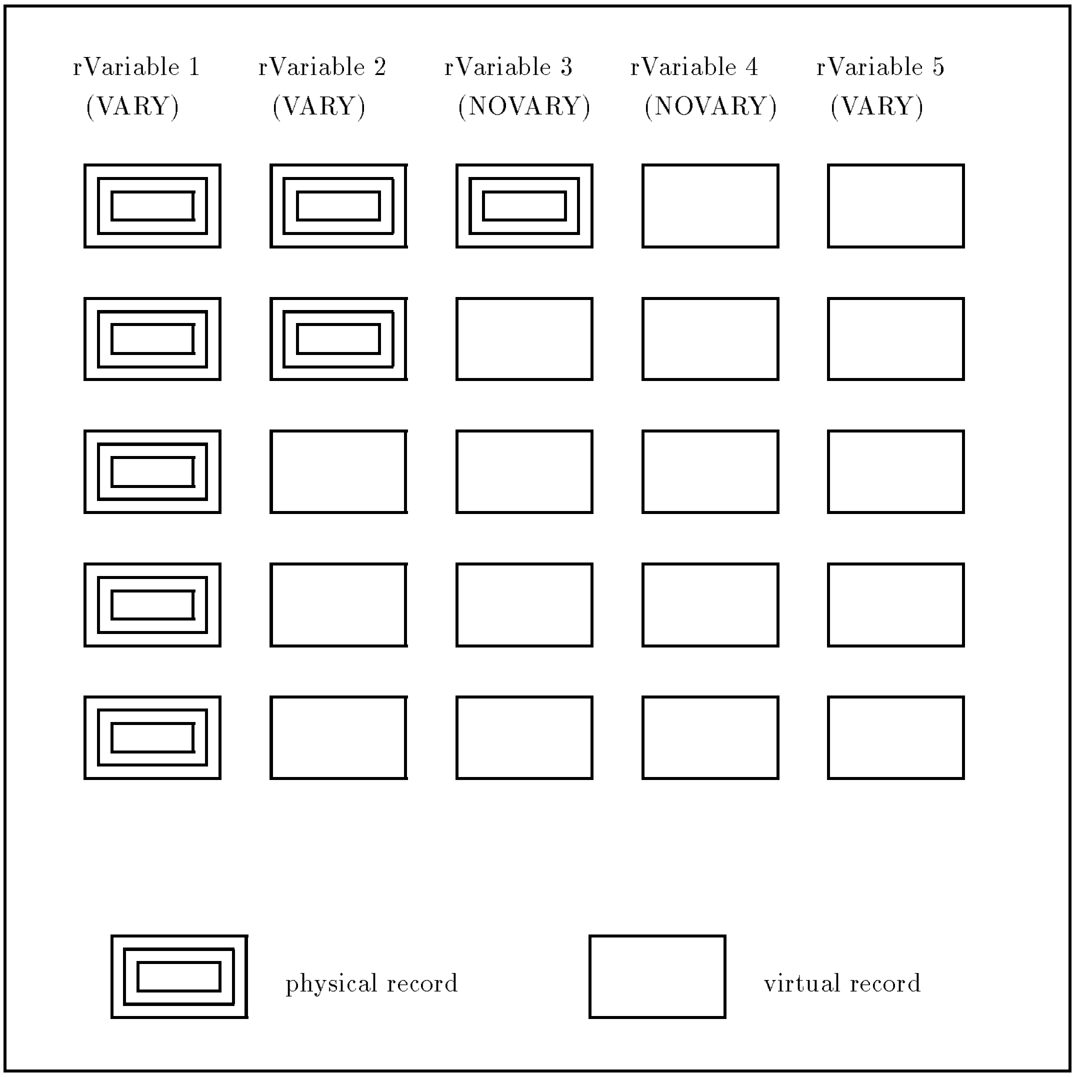

Figure 2.1: Physical vs. Virtual Records

Numbering

The record numbers in a CDF are numbered starting at one (1) for Fortran applications and starting at zero (0) for C applications.

Allocated Records

The Internal Interface may be used to allocate records for a variable in a single-file CDF.[16] Normally the number of records allocated would be the number that are to be written (assuming this can be determined). This can greatly improve performance when writing (and reading) values for the variable because of reduced overhead when searching the index entries (as described in Section 2.2.7). The application is expected to

[16] There is no reason to allocate records for a variable in a multi-file CDF.

write to all of the allocated records. When the CDF is closed, any allocated records for a variable which have not yet been written are written using the pad value for the variable. Records may only be allocated for a variable once but any number of records can eventually be written to the variable. For NRV variables, only one record may be allocated (because only one record will ever physically exist).

Performance is improved when using this method because the allocated records will be contiguous and will require only one index entry. This will greatly improve the time needed to locate a particular record when the variable is accessed. In addition, the CDF will be slightly smaller because of the reduced number of index records needed per variable.

Records are allocated using the `<PUT_,rVAR_ALLOCATERECS_>` and `<PUT_,zVAR_ALLOCATERECS_>>` operations of the Internal Interface.

Initial Records

The Internal Interface may be used to specify an initial number of records for a variable. The pad value for the variable is written at each record as if the application had done so itself. The Internal Interface allows this to be done more conveniently with only one function call. Note that the default pad value for the variable's data type will be used unless a pad value is specified for the variable (also using the Internal Interface). If a specific pad value is desired for a variable, then it must be specified before the number of initial records is specified.

Specifying a number of initial records for a variable would usually be done only for a CDF with the single-file format. Because the records would be allocated contiguously within the CDF file, the indexing scheme (see Section 2.2.7) would require fewer entries making the access to that variable more efficient. Note that this method is not as efficient as allocating records in those cases where all of the records are going to be written by the application. (The records would be written twice. Once with the pad value and then again by the application.)

The number of initial records specified would in most cases be the number of records planned for a variable. Note that additional records may be added to a variable at any time. After the initial records are written with the pad value, the actual data values would be written by the application. Note that for NRV variables the number of initial records must always be specified as one (1). This is because at most only one physical record will ever actually be written.

Initial records are written to rVariables and zVariables using the `<PUT_,rVAR_INITIALRECS_>` and `<PUT_, zVAR_INITIALRECS_>` operations of the Internal Interface.

Extend Records

As new records are written to a variable, additional records are allocated at the end of that variable. For CDFs with the multi-file format, only the records actually needed are appended to the corresponding variable file. The records always remain contiguous (allowing the location of a record to be determined very efficiently).

For single-file CDFs, however, the new records are allocated at the end of the `.cdf` file. The new records may not be located contiguously with the other records for the variable. This results in a new entry being created in the indexing scheme (see Section 2.2.7) used to locate variable records. The indexing scheme is more efficient if fewer entries are needed (resulting from larger blocks of contiguous records being allocated).

The CDF library attempts to optimize the indexing scheme by allocating more records than are actually needed with the assumption that they will eventually be used. A minimum number of records to allocate is maintained for each variable.

Because an application knows how records will be added to a variable more so than the CDF library, the Internal Interface may be used to explicitly specify the number of records to extend a variable when necessary. This only has an effect on CDFs with the single-file format and for variables that are not NRV (since at most only one physical record will ever be written for an NRV variable).

Note the distinction between records allocated and records actually written. The CDF library may allocate more records than are actually written by an application for the reasons stated above. When an application inquires the number of records for a variable, the number written, not the number allocated, will be returned by the CDF library.

Extend records are specified for rVariables/zVariables using the <PUT_,rVAR_EXTENDRECS_> and <PUT_,zVAR_EXTENDRECS_> operations of the Internal Interface.

2.3.9 Data Specification

Each variable in a CDF has a defined data specification. A variable's data specification consists of a data type and a number of elements of that data type. A variable's data specification is specified when the variable is created. The data specification of an existing variable may also be changed if either of the following conditions is true.

1. Values have not yet been written to the variable (including an explicitly written pad value — see Section 2.3.17).

2. The old data type and new data type are considered equivalent, and the number of elements for the variable are the same. Equivalent data types are described in Section 2.2.10.

Data Type

The supported data types are described in Section 2.2.10. Variables having any combination of data types may exist in the same CDF.

Number of Elements

In addition to a data type, each variable also has a number of elements. This refers to the number of elements of the data type at each variable value. For character string data types this is the number of characters in each string. (A variable value consists of the entire character string.) The character string can be thought of as an array of characters. For non-character string data types, this must always be one (1). An array of elements per variable value is not allowed for non-character string data types.

2.3.10 Record Variance

A variable's record variance specifies whether or not the variable's values change from record to record. The effect of a variable's record variance is defined as follows.

`VARY`	The values do change from record to record. Each variable record is physically written with no gaps between records (i.e., if a record more than one beyond the maximum record is written, the intervening records are also physically written and contain pad values). If a record is read beyond the maximum record written to a variable, the pad value for the variable is returned. Variables of this type are referred to as record-variant (RV).
`NOVARY`	The values do not change from record to record. Only one record is physically written to the variable. Each record contains the same values (including virtual records beyond the first record). Variables of this type are referred to as non-record-variant (NRV).

Section 2.3.8 describes variable records in more detail.

A variable's record variance is specified when the variable is created. The record variance of an existing variable may be changed only if values have not yet been written to that variable. (An explicit pad value may have been specified however.)

2.3.11 Dimension Variance

A variable's dimension variances specify whether or not the values change along the corresponding dimension. The effects of a dimension variance are defined as follows:

`VARY`	The values do change along the dimension. All of the values for the dimension (or all of the subarrays) are physically stored.
`NOVARY`	The values do not change along the dimension. Only one value (or subarray) is physically written for that dimension. Each value (or subarray) along that dimension is the same (including virtual values/subarrays beyond the first value/subarray).

Figure 2.2 illustrates the effect of dimension variances on a variable with 2-dimensional arrays (for a particular record). For variable 1 each value in the array is physically stored and therefore unique. Because variable 2 does not vary along the second dimension, each value along that dimension is the same so only one value for that dimension is physically stored (the other values are virtual). The same is true for variable 3 which does not vary along the first dimension. Variable 4 does not vary along either dimension. Only one value is physically stored for the array — all of the other values are the same (they are virtual).

A variable's dimension variances are specified when the variable is created. The dimension variances of an existing variable may be changed only if values have not yet been written to that variable. (An explicit pad value may have been specified, however.)

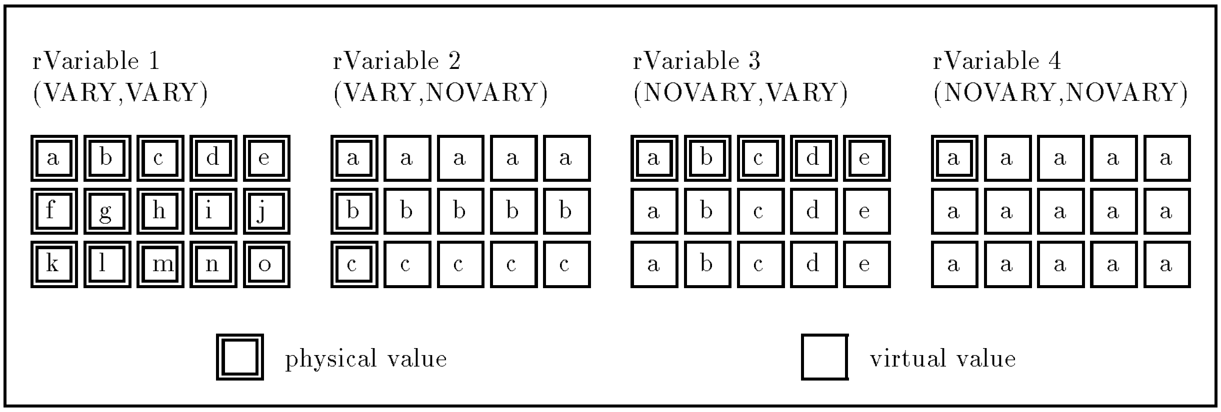

Figure 2.2: Physical vs. Virtual Dimensions

2.3.12 Majority

The variable majority of a CDF describes how variable values within each variable array (record) are stored. Each variable in a CDF has the same majority. The majority can be either row major or column major.

`ROW_MAJOR`	Row majority. The first dimension changes the slowest.
`COL_MAJOR`	Column majority. The first dimension changes the fastest.

For example, an array for an rVariable with `[VARY,VARY]` dimension variances in a 2-dimensional CDF with dimension sizes `[2,4]` and row majority would be stored as follows:

```
v(1,1), v(1,2), v(1,3), v(1,4), v(2,1), v(2,2), v(2,3), v(2,4)
```

where `v(i,j)` is the value at indices `(i,j)`. If the CDF had column majority, the array would be stored as follows:

```
v(1,1), v(2,1), v(1,2), v(2,2), v(1,3), v(2,3), v(1,4), v(2,4)
```

In each case `v(1,1)` is stored at the low address.

An application needs to be concerned with the majority of a CDF in the following cases:

1. When performing a variable hyper read, the values placed in the buffer by the CDF library will be in the variable majority of the CDF. The application must process the values according to that majority.

   When performing a variable hyper write, the CDF library expects the values in the buffer to be in the variable majority of the CDF. The application must place the values into the buffer in that majority.

2. When sequential access is used, the values are read/written in the order imposed by the variable majority of the CDF.

3. When single value reads/writes are performed, the majority could have an effect. The CDF library uses a caching scheme to optimize[17] the random access to variable values. If all of the values of a record are to be read/written, there may be an increase in performance if the values are accessed with (rather than against) the majority. For example, if the majority is row-major, increment the last index the fastest.

4. When performing a multiple variable read/write, the full-physical records in the buffer will/must be in the variable majority of the CDF.

A CDF's variable majority is specified when the CDF is created when using the Standard Interface but is set to the default variable majority for your CDF distribution when created using the Internal Interface. The majority of an existing CDF may be changed using the Internal Interface only if variable values have not yet been written. (Variables may exist and explicit pad values may have been specified, however.)

2.3.13 Single Value Access

Single value access allows only one value to be read from or written to a variable with a single call to the CDF library. Two parameters are specified when performing a single value read/write:

`record number`	The record number at which to perform the access.
`dimension indices`	The indices within the record at which to perform the access.

For 0-dimensional CDFs, the dimension indices are not applicable.

Single value access is sensitive to the record and dimension variances of a variable. For instance, if a variable has a record variance of `NOVARY` (with one record written) and a value is read from the fourth record, the CDF library will actually read the value from the first record (the record that is physically stored). If a value were written to the fourth record, the CDF library would actually write the value to the first record (the only record that actually physically exists). If the record variance is `VARY`, the values are written to the actual records. (The physical records are the same as the virtual records.) The same applies to any dimension variances that are `NOVARY`. When a set of indices is specified for a single value read/write, the index for a dimension whose variance is `NOVARY` is forced to the first index regardless of the actual index specified for that dimension (see Section 2.3.11).

In a C application single value access for rVariables is performed using either the `CDFvarGet` and `CDFvarPut` functions (Standard Interface) or the `<GET_,rVAR_DATA_>` and `<PUT_,rVAR_DATA_>` operations of the `CDFlib` function (Internal Interface). Single value access for zVariables is performed using the `<GET_,zVAR_DATA_>` and `<PUT_,zVAR_DATA_>` operations of the `CDFlib` function (Internal Interface). In a Fortran application single value access for rVariables is performed using either the `CDF_var_get` and `CDF_var_put` subroutines (Standard Interface) or the `<GET_,rVAR_DATA_>` and `<PUT_,rVAR_DATA_>` operations of the `CDF_lib` subroutine (Internal Interface). Single value access for zVariables is performed using the `<GET_,zVAR_DATA_>` and `<PUT_, zVAR_DATA_>` operations of the `CDF_lib` subroutine (Internal Interface).

[17] Since an application knows how it will be accessing a variable, it knows best how to optimize the caching scheme used. See Section 2.1.4 for details on how an application can control the CDF library caching scheme.

2.3.14 Hyper Access

Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library. In fact, the entire variable may be accessed at once (if a large enough memory buffer is available to your application). Hyper reads cause the CDF library to read from the variable record(s) in the CDF and place the values into a memory buffer provided by the application. Hyper writes cause the CDF library to take values from a memory buffer provided by the application and write them to the variable records in the CDF. Six parameters are specified when performing a hyper read/write:

`record number`	The record number at which to start the access.
`record count`	The number of records to access.
`record interval`	The interval between records being accessed. An interval of two (2) would indicate that every other record is to be accessed.
`dimension indices`	The indices within each record at which the access should begin.
`dimension counts`	The number of values along each dimension that should be accessed.
`dimension intervals`	For each dimension, the interval between values being accessed. An interval of three (3) would indicate that every third value is to be accessed.

For 0-dimensional CDFs, the dimension indices, counts, and intervals are not applicable.

A hyper access may or may not read/write a contiguous set of values stored for a variable in the CDF. However, the values in the memory buffer received/provided by the application are contiguous.

Hyper access is sensitive to the record and dimension variances of a variable. For instance, if a variable has a record variance of `NOVARY` (with one record written) and a hyper read of the first five records for that variable is requested, the CDF library will read the single record that is physically stored and place it five times (contiguously) into the memory buffer provided by the application. The same applies to any dimension variances that are `NOVARY`. For example, if the count for a dimension is three and the dimension variance is `NOVARY`, the one value (or subarray) physically stored will be read by the CDF library and placed into the application's memory buffer three times (contiguously).

Example (Fortran application)

Assume a 2-dimensional variable array with sizes `[2,4]`, row majority, a record variance of `VARY`, dimension variances of `[VARY,VARY]`, and hyper read parameters as follows:

record number	`5`
record count	`2`
record interval	`1`
dimension indices	`1,1`
dimension counts	`2,4`
dimension intervals	`1,1`

The values placed in the application's buffer would be as follows (with the first value being in low memory):

```
5(1,1) 5(1,2) 5(1,3) 5(1,4) 5(2,1) 5(2,2) 5(2,3) 5(2,4)
```

```
6(1,1) 6(1,2) 6(1,3) 6(1,4) 6(2,1) 6(2,2) 6(2,3) 6(2,4)
```

where `r(i,j)` is a physically stored value with `r` being the record number, `i` being the first dimension index, and `j` being the second dimension index. (`r`, `i`, and `j` are physical record numbers and dimension indices.)

If the dimension variances had been `[VARY,NOVARY]`, the values placed in the buffer would have been

```
5(1,1) 5(1,1) 5(1,1) 5(1,1) 5(2,1) 5(2,1) 5(2,1) 5(2,1)
6(1,1) 6(1,1) 6(1,1) 6(1,1) 6(2,1) 6(2,1) 6(2,1) 6(2,1)
```

If the record count had been `3` and the record interval `2`, the values placed in the buffer would have been

```
5(1,1) 5(1,2) 5(1,3) 5(1,4) 5(2,1) 5(2,2) 5(2,3) 5(2,4)
7(1,1) 7(1,2) 7(1,3) 7(1,4) 7(2,1) 7(2,2) 7(2,3) 7(2,4)
9(1,1) 9(1,2) 9(1,3) 9(1,4) 9(2,1) 9(2,2) 9(2,3) 9(2,4)
```

If the dimension counts had been `[2,2]` and the dimension intervals `[1,2]`, the values placed in the buffer would have been

```
5(1,1) 5(1,3) 5(2,1) 5(2,3)
6(1,1) 6(1,3) 6(2,1) 6(2,3)
```

If the CDF majority had been column major, the values placed in the buffer would have been...

```
5(1,1) 5(2,1) 5(1,2) 5(2,2) 5(1,3) 5(2,3) 5(1,4) 5(2,4)
6(1,1) 6(2,1) 6(1,2) 6(2,2) 6(1,3) 6(2,3) 6(1,4) 6(2,4)
```

Had these examples been for hyper writes, the CDF library would have expected to find the values in the application's buffer exactly as they were placed there during the corresponding hyper read. In the case where the record interval was `2`, the records being skipped would be written using the variable's pad value if they did not already exist. If they did already exist, they would not be affected.

In a C application hyper reads and writes for rVariables are performed using either the **`CDFvarHyperGet`** and **`CDFvarHyperPut`** functions (Standard Interface) or the **`<GET_,rVAR_HYPERDATA_>`** and **`<PUT_,rVAR_HYPERDATA_>`** operations of the **`CDFlib`** function (Internal Interface). Hyper reads and writes for zVariables are performed using the **`<GET_,zVAR_HYPERDATA_>`** and **`<PUT_,zVAR_HYPERDATA_>`** operations of the **`CDFlib`** function (Internal Interface). In a Fortran application hyper reads and writes for rVariables are performed using either the **`CDF_var_hyper_get`** and **`CDF_var_hyper_put`** subroutines (Standard Interface) or the **`<GET_, rVAR_HYPERDATA_>`** and **`<PUT_,rVAR_HYPERDATA_>`** operations of the **`CDF_lib`** subroutine (Internal Interface). Hyper reads/writes for zVariables are performed using the **`<GET_,zVAR_HYPERDATA_>`** and **`<PUT_,zVAR_HYPERDATA_>`** operations of the **`CDF_lib`** subroutine (Internal Interface).

2.3.15 Sequential Access

Sequential access provides a way to sequentially read/write the values physically stored for a variable. To use sequential access, a starting value must first be selected by specifying a record number and dimension indices. This selects the "current sequential value." A sequential read will return the value at the current sequential value and then automatically increment the current sequential value to the next value. Likewise, a sequential write will store a value at the current sequential value and then increment the current sequential value to the next value. Sequential reads are allowed until the end of the physical records has been reached (not the end of the virtual records [they never end]). Sequential reading will increment to the beginning of the next physical record if necessary. Sequential writing can be used to extend the physical records for a variable (as well as to overwrite existing values).

Example (Fortran application)

Assume a 2-dimensional array with sizes `[2,3]`, column majority, a record variance of `VARY`, dimension variances of `[VARY,VARY]`, nine (9) physical records written, and that the current sequential value has been set to record number `7` and indices `[2,2]`. Consecutive sequential reads would cause the following values to be read and returned to the application:

```
                     7(2,2) 7(1,3) 7(2,3)
8(1,1) 8(2,1) 8(1,2) 8(2,2) 8(1,3) 8(2,3)
9(1,1) 9(2,1) 9(1,2) 9(2,2) 9(1,3) 9(2,3)
END_OF_VAR
```

...where `r(i,j)` is a physically stored value with `r` being the record number, `i` the first dimension index, and `j` the second dimension index. (`r`, `i`, and `j` are physical record numbers and dimension indices.) The next sequential read after the last physical value would cause a status code indicating the end of the variable to be returned (`END_OF_VAR`).

Had the dimension variances been `[NOVARY,VARY]`, the values read would have been

```
       7(1,2) 7(1,3)
8(1,1) 8(1,2) 8(1,3)
9(1,1) 9(1,2) 9(1,3)
END_OF_VAR
```

Note that specifying the virtual value `7(2,2)` as the current sequential value caused physical value `7(1,2)` to actually be selected (because the first dimension variance is `NOVARY`).

Sequential access for rVariables is performed using the `<GET_,rVAR_SEQDATA_>` and `<PUT_,rVAR_SEQDATA_>` operations of the Internal Interface. Sequential access for zVariables is performed using the `<GET_,zVAR_SEQDATA_>` and `<PUT_,zVAR_SEQDATA_>` operations.

2.3.16 Multiple Variable Access

Multiple variable access allows an application to read from or write to multiple variables in a single operation. Multiple variable access works on either the rVariables or the zVariables of a CDF — not a mixture of the two. Up to all of the rVariables/zVariables may be accessed with a single call to the CDF library. For each variable specified in a multiple variable access, a full-physical record for that variable will be read/written. A full-physical record consists of all of the values exactly as they are physically stored in each variable record (the physical values). Virtual values do not apply when performing a multiple variable access (see Section 2.3.11). Three parameters are specified when performing a multiple variable read/write.

`variable count`	The number of rVariables/zVariables that are being accessed.
`variable list`	The rVariables/zVariables being accessed (specified by number).
`record numbers`	The record numbers at which the reads/writes will take place. For rVariables the record numbers must all be the same. For zVariables the record numbers can vary (but for most applications will all be the same).

Multiple variable access is sensitive to the record variances of the variables being accessed. (Dimension variances do not apply since full-physical records are being read/written.) If a variable has a record variance of `NOVARY`, then a read/write to that variable will always occur at the first record regardless of the actual record number specified (since at most only one physical record will ever exist). If the record variance were `VARY`, the reads/writes would take place at the actual record numbers specified.

For a multiple variable write operation an application must place into a memory buffer each of the full-physical records to be written. The order of the full-physical records must correspond to the order of the list of variables specified, and the memory buffer must be contiguous — there can be no gaps between the full-physical records. This memory buffer is then passed to the CDF library which scans through the buffer writing the full-physical records to the corresponding variables.

Likewise, for a multiple variable read operation the CDF library places into a memory buffer provided by the application the full-physical records read. The order of the full-physical records will correspond to the order of the list of variables specified and the full-physical records will be contiguous. The application must then process the buffer as needed.

Care must be used when generating and processing the memory buffer containing the full-physical records. If C `struct` objects or Fortran `STRUCTURE` variables are being used, it may be necessary to order the variables being read/written such that there are no gaps between elements of the structures (assuming you are defining structures containing one element per full-physical record where an element is a scalar variable or an array depending on the corresponding variable definition). On some computers the C and Fortran compilers will place gaps between the elements of these structures so that memory alignment errors are not generated when the elements are accessed. In general, defining the structures so that "larger" data types are before "smaller" data types should result in no gaps (e.g., the Fortran `REAL*8` data type is "larger" than a `INTEGER*2`, which is "larger" than a `BYTE`). The list of variables would be adjusted accordingly.

The variable majority must also be considered when performing a multiple variable read/write since full-physical records are being accessed. The majority of the values in the full-physical records retrieved from/placed into the memory buffer must be the same as the variable majority of the CDF.

For example, consider a column-major CDF containing the following three zVariables (as well as others):

zVariable Name	Data Specification	Dimensionality	Variances
`zVar1`	`CDF_INT2/1`[18]	`0:[]`	`T/`
`zVar2`	`CDF_CHAR/7`	`1:[5]`	`T/T`
`zVar3`	`CDF_REAL8/1`	`2:[2,4]`	`T/TT`

[18] This notation is used throughout this document. The data type is before the slash and the number of elements is after the slash. In this case the data type is (`CDF_INT2`) and the number of elements is one (1).

If a Fortran application were to perform a multiple variable read on these three zVariables, it could define a `STRUCTURE` to receive the physical records as follows:

```
STRUCTURE /inputStruct/
  REAL*8      zVar3values(2,4)
  INTEGER*2   zVar1value
  CHARACTER*7 zVar2values(5)
END STRUCTURE
```

Note that because a full-physical record for the zVariable `zVar2` is an odd number of bytes it would most likely cause a gap in the `STRUCTURE` if not placed at the end (on some computers). An approach that would work on all computers would be to use `EQUIVALENCE` statements as follows:

```
INTEGER*2   zVar1value
CHARACTER*7 zVar2values(5)
REAL*8      zVar3values(2,4)
BYTE        buffer(101)
EQUIVALENCE (zVar3values,buffer(1))
EQUIVALENCE (zVar1value,buffer(65))
EQUIVALENCE (zVar2values,buffer(67))
```

The `EQUIVALENCE` statements ensure that the full-physical records will be contiguous. In each of the above examples, the order of the zVariables would be `zVar3`, `zVar1`, `zVar2`.

C applications must also be concerned with the ordering of full-physical records in the memory buffer. Even if a `void` memory buffer is used with type casting to access individual values, the alignment of the values in the memory buffer is important (on some computers).

2.3.17 Variable Pad Values

Variable pad[19] values are used in several situations.

1. When the first value is written to a new record, the other values in that record will contain the pad value. This also applies to hyper writes if less than the entire record is written. (The unwritten values will contain the pad value.)
2. When a new record is written that is more than one record beyond the last record already written, the intervening records will contain pad values. This will only happen if the record variance is `VARY`. (The variable values vary from record to record.)
3. When reading a record beyond the last record for a variable, pad values will be returned.
4. Allocated records for a variable which have not yet been written when a CDF is closed will be written using the variable's pad value.

[19] These were previously known as fill values but were renamed to avoid confusion with the `FILLVAL` attribute (an NSSDC standard attribute).