```
for (recNum = 0; recNum <= maxRec; recNum++) {
   status = CDFlib (SELECT_, rVARs_RECNUMBER_, recNum,
                    GET_, rVAR_HYPERDATA_, values,
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
   /* process values */
   .
   .
}
.
.
```

Attribute Renaming. In this example the attribute named `Tmp` will be renamed `TMP`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;              /* Status returned from CDF library. */
.
.
status = CDFlib (SELECT_, ATTR_NAME_, "Tmp",
                 PUT_, ATTR_NAME, "TMP",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Sequential Access. In this example the values for a zVariable will be averaged. The values will be read using the sequential access method (see Section 2.3.14). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be `CDF_REAL4`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;             /* Status returned from CDF library. */
long        varNum;             /* zVariable number. */
long        recNum = 0;         /* Record number, start at first record. */
static long indices[2] = {0,0}; /* Dimension indices. */
float       value;              /* Value read. */
double      sum = 0.0;          /* Sum of all values. */
```

```
long        count = 0;          /* Number of values. */
float       ave;                /* Average value. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "FLUX", &varNum,
                NULL_);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (SELECT_, zVAR_, varNum,
                          zVAR_SEQPOS_, recNum, indices,
                GET_, zVAR_SEQDATA_, &value,
                NULL_);
while (status >= CDF_OK) {
  sum += value;
  count++;
  status = CDFlib (GET_, zVAR_SEQDATA_, &value,
                   NULL_);
}

if (status != END_OF_VAR) UserStatusHandler (status);

ave = sum / count;
.
.
```

Attribute rEntry Writes. In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;                  /* Status returned from CDF
                                           library. */
static float   scale[2] = {-90.0,90.0}; /* Scale, minimum/maximum. */
.
.
status = CDFlib (SELECT_, rENTRY_NAME_, "LATITUDE",
                          ATTR_NAME_, "FIELDNAM",
                PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                    "Latitude            ",
                SELECT_, ATTR_NAME_, "SCALE",
                PUT_, rENTRY_DATA_, CDF_REAL4, (long) 2, scale,
                SELECT_, ATTR_NAME_, "UNITS",
                PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                    "Degrees north       ",
                NULL_);
if (status != CDF_OK) UserStatusHandler (status);
```

```
.
.
```

Multiple zVariable Write. In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see Section 2.3.15). It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;             /* Status returned from CDF library. */
short       time;               /* 'Time' value. */
char        vectorA[3];         /* 'vectorA' values. */
double      vectorB[5];         /* 'vectorB' values. */
long        recNumber;          /* Record number. */
char        buffer[45];         /* Buffer of full-physical records. */
long        varNumbers[3];      /* Variable numbers. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "vectorB", &varNumbers[0],
                       zVAR_NUMBER_, "time", &varNumbers[1],
                       zVAR_NUMBER_, "vectorA", &varNumbers[2],
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
for (recNumber = 0; recNumber < 100; recNumber++) {
   .
   /* read values from input file */
   .
   memmove (&buffer[0], vectorB, 40);
   memmove (&buffer[40], &time, 2);
   memmove (&buffer[42], vectorA, 3);
   status = CDFlib (SELECT_, zVARs_RECNUMBER_, recNumber,
                    PUT_, zVARs_RECDATA_, 3L, varNumbers, buffer,
                    NULL_);
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

Note that it would be more efficient to read the values directly into `buffer`. The method shown here was used to illustrate how to create the buffer of full-physical records.

A Potential Mistake We Don't Want You to Make

The following example illustrates one of the most common mistakes made when using the Internal Interface in a C application. Please don't do something like the following:

```
.
.
#include "cdf.h"
.
.
CDFid          id;                      /* CDF identifier (handle). */
CDFstatus      status;                  /* Status returned from CDF
                                           library. */
long           varNum;                  /* zVariable number. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                 GET_, zVAR_NUMBER_, "EPOCH", &varNum,
                 SELECT_, zVAR_, varNum,                  /* ERROR! */
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

It looks like the current zVariable will be selected based on the zVariable number determined by using the `<GET_,zVAR_NUMBER_>` operation. What actually happens is that the zVariable number passed to the `<SELECT_,zVAR_>` operation is undefined. This is because the C compiler is passing `varNum` by value rather than reference.[11] Since the argument list passed to `CDFlib` is created before `CDFlib` is called, `varNum` does not yet have a value. Only after the `<GET_,zVAR_NUMBER_>` operation is performed does `varNum` have a valid value. But at that point it's too late since the argument list has already been created. Don't do this.

In this type of situation you would have to make two calls to `CDFlib`. The first would inquire the zVariable number and the second would select the current zVariable. Because the `<SELECT_,zVAR_NAME_>` operation allows you to select the current zVariable directly by name, this particular situation should never arise. It was simply used as an example. If you do encounter a situation like this that you think could be avoided with a new operation, contact CDFsupport; we're adding new operations all the time.

4.8 Interpreting CDF Status Codes

Most CDF functions return a status code of type `CDFstatus`. The symbolic names for these codes are defined in `cdf.h` and should be used in your applications rather than using the true numeric values. Appendix C explains each status code. When the status code returned from a CDF function is tested, the following rules apply.

`status > CDF_OK`	Indicates successful completion but some additional information is provided. These are informational codes.

[11] Fortran programmers can get away with doing something like this because everything is passed by reference.

`status = CDF_OK`	Indicates successful completion.
`CDF_WARN < status < CDF_OK`	Indicates that the function completed but probably not as expected. These are warning codes.
`status < CDF_WARN`	Indicates that the function did not complete. These are error codes.

The following example shows how you could check the status code returned from CDF functions.

```
CDFstatus status;
.
.
status = CDFfunction (...);   /* any CDF function returning CDFstatus */
if (status != CDF_OK) {
  UserStatusHandler (status, ...);
  .
  .
}
```

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

```
#include <stdio.h>

#include "cdf.h"

void UserStatusHandler (status)
CDFstatus status;
{
char message[CDF_STATUSTEXT_LEN+1];

if (status < CDF_WARN) {
  printf ("An error has occurred, halting...\n");
  CDFerror (status, message);
  printf ("%s\n", message);
  exit (status);
}
else
  if (status < CDF_OK) {
    printf ("Warning, function may not have completed as expected...\n");
    CDFerror (status, message);
    printf ("%s\n", message);
  }
  else
    if (status > CDF_OK) {
      printf ("Function completed successfully, but be advised that...\n");
      CDFerror (status, message);
      printf ("%s\n", message);
```

```
      }
   return;
}
```

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

4.9 CDF_EPOCH Values

Several functions exist that compute, break down, parse, and encode `CDF_EPOCH` values. These functions may be called by applications using the `CDF_EPOCH` data type and are included in the CDF library. Section 2.2.9 describes EPOCH values.

4.9.1 computeEPOCH

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

```
double computeEPOCH(     /* Out -- CDF_EPOCH value returned. */
long year,               /* In  -- Year (AD, e.g., 1994). */
long month,              /* In  -- Month (1-12). */
long day,                /* In  -- Day (1-31). */
long hour,               /* In  -- Hour (0-23). */
long minute,             /* In  -- Minute (0-59). */
long second,             /* In  -- Second (0-59). */
long msec);              /* In  -- Millisecond (0-999). */
```

4.9.2 EPOCHbreakdown

`EPOCHbreakdown` breaks down a `CDF_EPOCH` value into the individual components.

```
void EPOCHbreakdown(
double epoch,            /* In  -- The CDF_EPOCH value. */
long   *year,            /* Out -- Year (AD, e.g., 1994). */
long   *month,           /* Out -- Month (1-12). */
long   *day,             /* Out -- Day (1-31). */
long   *hour,            /* Out -- Hour (0-23). */
long   *minute,          /* Out -- Minute (0-59). */
long   *second,          /* Out -- Second (0-59). */
long   *msec);           /* Out -- Millisecond (0-999). */
```

4.9.3 parseEPOCH

parseEPOCH parses a standard date/time character string and returns a CDF_EPOCH value. The format of the string is defined in Section 2.2.9 (and is also the format produced by the encodeEPOCH function). If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH(                    /* Out -- CDF_EPOCH value
                                                 returned. */
char epString[EPOCH_STRING_LEN+1]);   /* In  -- The standard date/time
                                                 character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

4.9.4 encodeEPOCH

encodeEPOCH encodes a CDF_EPOCH value into a standard date/time character string. The format of the string is defined in Section 2.2.9 (and is also the format expected by the parseEPOCH function).

```
void encodeEPOCH(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH_STRING_LEN+1]);    /* Out -- The standard date/time
                                                  character string. */
```

EPOCH_STRING_LEN is defined in cdf.h.

4.9.5 encodeEPOCH1

encodeEPOCH1 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmdd.ttttttt, where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), and ttttttt is the fraction of the day (e.g., 5000000 is 12 o'clock noon).

```
void encodeEPOCH1(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH1_STRING_LEN+1]);   /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

4.9.6 encodeEPOCH2

encodeEPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymmddhhmmss where yyyy is the year, mm is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), and ss is the second (0-59).

```
void encodeEPOCH2(
double epoch;                              /* In  -- the CDF_EPOCH value. */
char epString[EPOCH2_STRING_LEN+1]);       /* Out -- The alternate date/time
                                                  character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

4.10 Custom C Functions

Most of the Standard Interface functions callable from C applications are implemented as C macros that call CDFlib (Internal Interface). For example, the CDFcreate function is actually defined as the following C macro:

```
#define CDFcreate(CDFpath,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFpath, numDims, dimSizes, id, \
        PUT_, CDF_ENCODING_, encoding, \
              CDF_MAJORITY_, majority, \
        NULL_)
```

These macros are defined in cdf.h. Where your application calls CDFcreate, the C compiler (preprocessor) expands the macro into the corresponding call to CDFlib.

The flexibility of CDFlib allows you to define your own custom CDF functions using C macros. For instance, a function that inquires the format of a CDF could be defined as follows:

```
#define CDFinquireFormat(id,format) \
CDFlib (SELECT_, CDF_, id, \
        GET_, CDF_FORMAT_, format, \
        NULL_)
```

Your application would call the function as follows:

```
.
.
CDFid     id;            /* CDF identifier. */
CDFstatus status;        /* Returned status code. */
long      format;        /* Format of CDF. */
.
.
status = CDFinquireFormat (id, &format);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Chapter 5

CDF Fortran Programming Reference

5.1 Introduction

This chapter provides a reference for using the CDF library from Fortran applications. A separate reference exists for writing CDF applications in C (Chapter 4).

5.2 Compiling

Each program, subroutine, or function that calls the CDF library or references CDF parameters must include one or more CDF include files. On VMS systems a logical name, `CDF$INC`, that specifies the location of the CDF include files is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_INC`, that serves the same purpose is defined in the definitions file named `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

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

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

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

On MS-DOS systems (assuming Microsoft Fortran is being used) the following lines would be included at/near the top of each routine/source file:

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

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

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

5.2.1 VMS Systems

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

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

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

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

5.2.2 UNIX Systems

An example of the command to compile a source file on UNIX systems would be as follows:[1]

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

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

The `-c` option specifies that only an object module is to be produced. (The link step is described in Section 4.3.) The object module created will be named `<source-name>.o`.

5.2.3 MS-DOS Systems

NOTE: Even though your application is written in Fortran and compiled with a Fortran compiler, compatible C runtime system libraries will be necessary to successfully link your application. This is because the

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

CDF library is written in C and calls C runtime system functions.

An example of the command to compile a source file on MS-DOS systems would be as follows (for Microsoft Fortran):[2]

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

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

The `/c` option specifies that only an object module is to be produced. (The link step is described in Section 5.3.) The object module will be named `<source-name>.obj`.

The `/AL` option specifies that the object module is to be compiled using the large memory model. The huge memory model could also be used if necessary. (The CDF library can be built using either the large or huge memory model.) All of your application source files must be compiled using the same memory model as that used for the CDF library. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large memory model.

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

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

5.3 Linking

Your applications must be linked with the CDF library (`LIBCDF.OLB` on VMS systems, `libcdf.a` on UNIX systems, and `libcdf.lib` on MS-DOS systems).[3] Both the Standard and Internal interfaces for Fortran applications are built into the CDF library. On VMS systems a logical name, `CDF$LIB`, that specifies the location of the CDF library is defined in the definitions file, `DEFINITIONS.COM`, provided with the CDF distribution. On UNIX systems an environment variable, `CDF_LIB`, that serves the same purpose is defined in the definitions file `definitions.<shell>` (where `<shell>` is the shell being used: `csh`, `ksh`, `sh`, or `tcsh`). This section assumes that you are using the appropriate definitions file.

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

[3] A shareable version of the CDF library is also available on VMS and some flavors of UNIX. Its use is described in Section 5.4.

5.3.1 VMS Systems

An example of the command used to link an application on a VMS system would be as follows:

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

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

5.3.2 OpenVMS Systems

An example of the command used to link an application on an OpenVMS system would be as follows:

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

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

5.3.3 UNIX Systems

An example of the command used to link an application on a UNIX system would be as follows:

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

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

Combining the Compile and Link

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

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

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

5.3.4 MS-DOS Systems

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

An example of the command used to link an application on MS-DOS systems would be as follows (for Microsoft Fortran):

```
> LINK /NOE <object-file(s)>,<executable>,nul.map,<lib-path>libcdf.lib;
```

where <object-file(s)> is your application's object module(s) (the .obj extension is not necessary); <executable> is the name of the executable file to be created (.exe will be appended); and <lib-path> is the pathname of the directory containing libcdf.lib. You will need to know where on your system libcdf.lib has been installed.

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

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

NOTE: The same memory model must have been used to compile your application's source files and the CDF library. The CDF library can be compiled using either the large or huge memory model. The CDF libraries supplied with the CDF distribution for MS-DOS systems are compiled using the large memory model.

5.4 Linking, Shared CDF Library

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

Machine/Operating System	Shared CDF Library
VAX/VMS & DEC Alpha/OpenVMS	LIBCDF.EXE
Sun/SunOS & Sun/SOLARIS	libcdf.so
HP 9000/HP-UX	libcdf.sl
IBM RS6000/AIX	libcdf.o

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

5.4.1 VAX/VMS

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

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

NOTE: On VMS systems the shared CDF library may also be installed in `SYS$SHARE`.

5.4.2 DEC Alpha/OpenVMS

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

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

NOTE: On OpenVMS systems the shareable CDF library may also be installed in `SYS$SHARE`.

5.4.3 Sun/SunOS

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

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

5.4.4 Sun/SOLARIS

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

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

5.4.5 HP 9000/HP-UX

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

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

5.4.6 IBM RS6000/AIX

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

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

5.5 Argument Passing

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

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

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

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

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

5.6 Item Referencing

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

5.7 CDF Parameters — Predefined Constants and Types

The following sections describe constants and types defined for use by all CDF application programs written in Fortran. These constants and types are defined in `cdf.inc`. The file `cdf.inc` should be `INCLUDE`ed in all application source files referencing CDF functions/parameters.

5.7.1 Status Code Constants

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

Section 5.9 describes how to use these constants to interpret status codes.

5.7.2 CDF Formats

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

5.7.3 CDF Data Types

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

`CDF_BYTE`	1-byte, signed integer.

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

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

5.7.4 Data Encodings

A CDF's data encoding affects how its attribute entry and variable data values are stored (on disk). Attribute entry and variable values passed into the CDF library (to be written to a CDF) should always be in the host machine's native encoding. Attribute entry and variable values read from a CDF by the CDF library and passed out to an application will be in the currently selected decoding for that CDF (see Section 2.2.8).

HOST_ENCODING	Indicates host machine data representation (native). This encoding will provide the greatest performance when reading/writing on a machine of the same type.
NETWORK_ENCODING	Indicates network transportable data representation (XDR).
VAX_ENCODING	Indicates VAX data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.
ALPHAVMSd_ENCODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's D_FLOAT representation.
ALPHAVMSg_ENCODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values are encoded in Digital's G_FLOAT representation.
SUN_ENCODING	Indicates SUN data representation.
SGi_ENCODING	Indicates Silicon Graphics Iris and Power Series data representation.

DECSTATION_ENCODING	Indicates DECstation data representation.
ALPHAOSF1_ENCODING	Indicates DEC Alpha running OSF/1 data representation.
IBMRS_ENCODING	Indicates IBMRS data representation (IBM RS6000 series).
HP_ENCODING	Indicates HP data representation (HP 9000 series).
IBMPC_ENCODING	Indicates IBM PC data representation.
NeXT_ENCODING	Indicates NeXT data representation.

When creating a CDF (via the Standard Interface) or respecifying a CDF's encoding (via the Internal Interface), you must specify NETWORK_ENCODING, HOST_ENCODING, or the specific encoding of the host machine (e.g., VAX_ENCODING). Specifying the host machine's encoding explicitly has the same effect as specifying HOST_ENCODING.

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

5.7.5 Data Decodings

A CDF's decoding affects how its attribute entry and variable data values are passed out to a calling application. The decoding for a CDF may be selected and reselected any number of times while the CDF is open. Selecting a decoding does not affect how the values are stored in the CDF file(s) — only how the values are decoded by the CDF library. Any decoding may be used with any of the supported encodings. Section 2.2.8 describes a CDF's decoding in more detail.

HOST_DECODING	Indicates host machine data representation (native). This is the default decoding.
NETWORK_DECODING	Indicates network transportable data representation (XDR).
VAX_DECODING	Indicates VAX data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation.
ALPHAVMSd_DECODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's D_FLOAT representation.
ALPHAVMSg_DECODING	Indicates DEC Alpha running OpenVMS data representation. Double-precision floating-point values will be in Digital's G_FLOAT representation.
SUN_DECODING	Indicates SUN data representation.
SGi_DECODING	Indicates Silicon Graphics Iris and Power Series data representation.
DECSTATION_DECODING	Indicates DECstation data representation.
ALPHAOSF1_DECODING	Indicates DEC Alpha running OSF/1 data representation.
IBMRS_DECODING	Indicates IBMRS data representation (IBM RS6000 series).
HP_DECODING	Indicates HP data representation (HP 9000 series).
IBMPC_DECODING	Indicates IBM PC data representation.

NeXT_DECODING	Indicates NeXT data representation.

The default decoding is HOST_DECODING. The other decodings may be selected via the Internal Interface with the <SELECT_,CDF_DECODING_> operation. Section 2.2.8 describes those situations in which a decoding other than HOST_DECODING may be desired.

5.7.6 Variable Majorities

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

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

For Fortran applications the compiler defined majority for arrays is column major. The first dimension of multidimensional arrays varies the fastest in memory.

Knowing the majority of a CDF's variables is necessary when performing hyper reads and writes. During a hyper read the CDF library will place the variable data values into the memory buffer in the same majority as that of the variables. The buffer must then be processed according to that majority. Likewise, during a hyper write the CDF library will expect to find the variable data values in the memory buffer in the same majority as that of the variables.

The majority must also be considered when performing sequential reads and writes. When sequentially reading a variable, the values passed out by the CDF library will be ordered according to the majority. When sequentially writing a variable, the values passed into the CDF library will be assumed (by the CDF library) to be ordered according to the majority.

5.7.7 Record/Dimension Variances

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

VARY	True record or dimension variance.
NOVARY	False record or dimension variance.

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

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

5.7.8 Attribute Scopes

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

GLOBAL_SCOPE	Indicates that an attribute's scope is global (applies to the CDF as a whole).
VARIABLE_SCOPE	Indicates that an attribute's scope is by-variable. (Each rEntry or zEntry corresponds to an rVariable or zVariable, respectively.)
GLOBAL_SCOPE_ASSUMED	Indicates that an attribute's scope is assumed global.
VARIABLE_SCOPE_ASSUMED	Indicates that an attribute's scope is assumed by-variable.

GLOBAL_SCOPE and VARIABLE_SCOPE are definite scopes. GLOBAL_SCOPE_ASSUMED and VARIABLE_SCOPE_ASSUMED are assumed scopes.

Section 2.4.3 explains attribute scopes and how assumed scopes can arise in a CDF.

5.7.9 Read-Only Modes

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

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

5.7.10 zModes

Once a CDF has been opened, it may be placed into one of two variations of zMode. zMode is fully explained in Section 2.1.2. A zMode is selected for a CDF via the Internal Interface using the <SELECT_,CDF_zMODE_> operation.

zMODEoff	Turns off zMode.
zMODEon1	Turns on zMode/1.
zMODEon2	Turns on zMode/2.

5.7.11 -0.0 to 0.0 Modes

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