* Portability Issues

# Chapter Overview

The NCSA implementation of HDF is accessible to both C and FORTRAN programs and is implemented on many different machines and several operating systems. There are important differences between C and FORTRAN, and among implementations of each language, especially FORTRAN. There are also important differences among the machines and operating systems that HDF supports.

If HDF is to be a portable tool, these differences must be constructively addressed. This chapter describes many of these differences, discusses the problems and issues associated with them, and presents the methods employed in the HDF implementation to reduce their impact.

## The HDF Environment

The list of machines and operating systems on which HDF is implemented is steadily growing. For reasons that this chapter will make clear, the number of NCSA-supported HDF platforms is growing slowly. Every time a platform is added, additional code must be written to address concerns of memory management, operating system and file system differences, number representations, and differences in FORTRAN and C implementations on that system.

### Supported Platforms

As of this writing, NCSA supports the platforms listed in Table 11a.

* NCSA-supported HDF Platforms

|  |  |
| --- | --- |
| Hardware Platform | Operating System |
| Convex | Concentrix |
| Cray X-MP, Y-MP, Cray 2 | UNICOS |
| DEC Alpha | Ultrix |
| DECStation | Ultrix |
| HP 9000 | HPUX |
| IBM PC | MS DOS, Windows 3.1 |
| IBM RS/6000 | AIX |
| IBM RT | UNIX |
| Macintosh | MPW Shell |
| NeXT | NeXTStep |
| Silicon Graphics | UNIX |
| Sun Sparc | UNIX |
| Vax | VMS |

HDF has also been ported to several platforms that NCSA does not currently support. These include Alliant, Apollo (Domain), HP 3000, Stellar, Amiga, Symbolics, Fujitsu, and IBM 3090 (MVS).

### Language Standards

Unfortunately, not all compilers are the same. FORTRAN compilers often differ in the ways they pass parameters, in the identifier naming conventions they employ, and in the number types that they support. Similarly, though generally not as drastically, C compilers differ in the number types that they support and in their adherence to the ANSI C standard.

To minimize the difficulties caused by these differences, the HDF source code is written primarily in the following dialects:

* FORTRAN 77
* ANSI C
* The original C defined by Kernighan and Ritchie[[1]](#footnote-1), hereafter referred to as old C

Almost all platforms have C and FORTRAN compilers that adhere to at least one of these standards.

When time and resources permit, NCSA attempts to support features or variations in other dialects of C and FORTRAN, particularly on platforms that are important to NCSA users. Much of the remainder of this chapter addresses these efforts.

### Guidelines

One cannot over stress the importance of following the guidelines outlined in this chapter. It may take longer to write code and it may be difficult to adapt your coding style, but the long-term benefits, in terms of portability and maintenance costs, will be well worth the effort.

## Organization of Source Files

Three types of files appear in the HDF source code directory:

* Header files
* Source code files
* Configuration files

Header files and source code files are organized by application area. All of the functions that apply to a particular application area are stored in three source files, and all the definitions and declarations that apply to that application are stored in a corresponding header file. The makefile describes the dependencies among the source and header files and provides the commands required to compile the corresponding libraries and utilities.

### Header Files

Certain application modules require header files. The header file dfan.h, for example, contains definitions and declarations that are unique to the annotation interface.

There are also several general header files that are used in compiling the libraries for all application areas:

hdf.h and hdfi.h[[2]](#footnote-2)  
hdf.h contains declarations and definitions for the common data structures used throughout HDF, definitions of the HDF tags, definitions of error numbers, and definitions and declarations specific to the low level interface. Since hdf.h depends on hdfi.h, it includes hdfi.h via #include.

hdfi.h contains information specific to the various NCSA-supported HDF computing environments, environmental parameters that need to be set to particular values when compiling the HDF libraries, and machine dependent definitions of such things as number types and macros for reading and writing numbers.

When porting HDF to a new system, only hdfi.h and the makefile should need to be modified, though there may be exceptions.

It is normally a good idea to include hdf.h (and therefore indirectly hdfi.h) in user programs, though users usually need not be aware of its contents.

hproto.h This file contains ANSI C prototypes for all HDF C routines. It must be included in ANSI C programs that call HDF routines.

constants.i This file is for use in FORTRAN programs. It contains important constants, such as tag values, that are defined in hdf.h. Systems with FORTRAN preprocessors might be able to include this file via #include statements or their equivalent.

dffunc.i This file is for use in FORTRAN programs. It contains declarations of all HDF FORTRAN-callable functions. Systems with FORTRAN preprocessors might be able to include this file via #include statements or their equivalent.

### Source Code Files

All HDF operations are performed by routines written in C. Hence, even FORTRAN calls to HDF result in calls to the corresponding C routines. Because of the problems described below the relationships between the C routines and the corresponding FORTRAN routines can be confusing. This section discusses the C and FORTRAN source file organization. It is followed by discussions of problems users will face in the FORTRAN–C interface.

HDF interfaces typically have three or four associated files. For example, the scientific data set (SDS) interface is associated with the following files: dfsd.h, dfsd.c, dfsdf.c, and dfsdff.f.

These files fill the following roles:

Header files  
The \*.h files are header files.

Normal C routines  
These routines do the actual HDF work. The others are used to transfer control and data from a FORTRAN environment to a C environment.

These routines are in the \*.c files, as in dfsd.c. Every call to HDF, whether from C or FORTRAN, ultimately results in a call to one of these routines.

C routines that are directly callable from FORTRAN  
These routines provide recognizable function names to the linker. They may also perform operations on data they receive from the FORTRAN routines that call them, such as transferring a FORTRAN string to a local C data area. Examples are provided below.

These routines are in the \*f.c files, such as dfsdf.c. The f means that the routines can be called from FORTRAN; the .c means that they are C source code.

FORTRAN routines that perform some operation on the parameters that C would be unable to perform, before and/or after calling the corresponding C routine  
These routines are required, for example, when one of the parameters is a string. The corresponding C routine has no way of knowing the length of the string unless it is explicitly given the length by the FORTRAN routine.

These routines are in the \*ff.f files, such as dfsdff.f. The ff means that the routines perform some FORTRAN operation that C cannot perform and that they are to be called from FORTRAN; the .f means that they are FORTRAN source code.

The roles of these different types of source file types will become clearer as we look at some of the problems that arise in interfacing C and many different implementations of FORTRAN.

### File Naming Conventions

The naming conventions for HDF library source code files are complicated by several factors. Because HDF must accommodate a wide variety of platforms, all files that will compile to object modules must have names that are unique in the first 8 characters, ignoring case. The difficulties involved in maintaining a FORTRAN-callable interface to a library that is primarily written in C further complicate the naming of source code files.

## Passing Strings between FORTRAN and C

One of the most important differences between FORTRAN and C compilers is in the way strings are represented. Different compilers use different data structures for strings, and supply string length information in different ways.

### Passing Strings from FORTRAN to C

When strings are passed between FORTRAN and C routines, they may need to be converted from one representation to the other. C compilers store strings in an array of type char, terminated by a null byte (\0). The name of a string variable is equivalent to a pointer the first character in the string. FORTRAN compilers are not consistent in the ways that they store strings.

Two pieces of information must be acquired before FORTRAN can pass a string to C:

* The string’s length
* The string’s address

The string’s length is determined by invoking the standard FORTRAN function len (), which returns the length of a string. Since C expects a null byte at the end of a string, care must be taken that this null byte does not overwrite useful information in the FORTRAN string.

Determining the string’s address is more difficult because of the different ways that different FORTRAN implementations store strings. The macro \_fcdtocp (FORTRAN character descriptor to C pointer) is used to acquire this information. \_fcdtocp is one of the elements that must be customized for each platform. The following paragraphs discuss several existing customized implementations:

* UNICOS FORTRAN stores strings in a structure called \_fcd (FORTRAN character descriptor). \_fcdtocp is a built-in UNICOS function that returns the string’s address. (Since UNICOS provides this function, HDF omits the corresponding macro definition on UNICOS systems.
* VMS FORTRAN uses a string descriptor structure that provides the string’s address and length. When compiled under VMS, \_fcdtocp extracts the string's address from that structure.
* Most other FORTRAN compilers supported by HDF store strings just as C does, in character arrays with the array name identifying the array's address. In such situations, nothing special needs to be done to pass a string from FORTRAN to C, except to add a NULL byte.

An HDF FORTRAN call that involves passing a string result in the following sequence of actions:

* A FORTRAN filter routine determines the length and address in memory of the string. Since this filter is a FORTRAN routine, it can be found in the appropriate \*ff.f file.
* The FORTRAN filter then calls a C routine, to which it passes all parameters from the initial call the string's length.
* The C routine converts the FORTRAN string to a C string by copying it to a C array of type char and appending a null byte. Since this C routine serves as a link between a FORTRAN filter and the corresponding C interface call, it can be found in the appropriate \*f.c file.
* This C routine then calls the HDF C routine that performs the actual work.

This process is illustrated in Figure 11a, "Sequence of Events when a FORTRAN Call Includes a String as a Parameter."

* Sequence of Events when a FORTRAN Call Includes a String as a Parameter



### Passing Strings from C to FORTRAN

When strings are passed from C to FORTRAN, the reverse procedure is followed. First, a string pointer is allocated within the FORTRAN routine's data area. (It is assumed that the space pointed to has already been allocated, and is sufficiently large to hold the string.) The string is then copied from the C data area to the FORTRAN data area. Finally, the FORTRAN string's data area is padded with blanks, if necessary.

## Function Return Values between FORTRAN and C

When a FORTRAN routine calls a C function, it always expects a return value from that function. Unfortunately, C functions do not always return arguments in a FORTRAN-compatible format.

To solve this problem, some FORTRAN compilers offer the option of controlling the form of the return value from a function. For example, Language Systems FORTRAN for the Macintosh requires that all C function declarations be prepended by the word pascal so that the return value can be recognized by a FORTRAN routine that calls it, as in:

pascal int dsgrang(void \*pmax, void \*pmin)

Since C always expects return values to be passed by value rather than, say, by reference, it is important to coerce FORTRAN functions to do the same. This is accomplished by defining a macro FRETVAL that is prepended to the declaration of every FORTRAN-callable C function. For example:

FRETVAL(int)

dsgrang(void \*pmax, void \*pmin)

If Language Systems FORTRAN is to be used, FRETVAL is defined in hdfi.h as follows:

#if defined(MAC) /\* with LS FORTRAN \*/

# define FRETVAL(x) pascal x

#endif

## Differences in Routine Names

HDF generally employs standard C conventions in naming routines. But many FORTRAN compilers impose varying restrictions on the length, character set, and form of identifiers, some of which are considerable more restrictive than the C conventions. Therefore, an extra effort must be made to accommodate those FORTRAN compilers.

To address this issue, HDF defines a set of preprocessor flags in hdfi.h. Then conditional compilation, with #ifdef statements in the source code, produces routine names that the target system’s FORTRAN will understand.

### Case Sensitivity

C compilers are case sensitive; uppercase and lowercase letters are recognized as different characters. Many FORTRAN compilers are not case sensitive; they allow users to use uppercase and lowercase letters while naming routines in the source code, but the names are converted to all uppercase or all lowercase in the object module symbol tables. Routine name recognition problems are common when routines compiled by a case sensitive compiler are to be linked with routines compiled by a non-case sensitive compiler.

For example, the UNICOS FORTRAN compiler allows you to name routines without regard to case, but produces object module symbol tables with the routine names in all uppercase. UNICOS C, on the other hand, performs no such conversion.

Consider the HDF routine Hopen. Hopen is written in C, so the HDF library symbol table contains the name Hopen. Suppose you make the following call in your UNICOS FORTRAN program:

file\_id = Hopen('myfile', ...)

The FORTRAN compiler will create an object module symbol table with the routine name HOPEN. When you link it to the HDF library, it will find Hopen but not HOPEN, and will generate an unsatisfied external reference error.

HDF supports the following non-case sensitive compilers:

* VMS FORTRAN
* UNICOS FORTRAN
* Language Systems FORTRAN.

All of these compilers convert identifiers to all uppercase when building an object module symbol table. In the following discussion, they are referred to as all-uppercase compilers.

The HDF Solution

HDF addresses the all-uppercase compiler problem in the platform-specific section of hdfi.h where the DF\_CAPFNAMES flag is defined. With conditional compilation, HDF generates all-uppercase routine names and symbol table entries.

Once again, consider UNICOS. The UNICOS section of hdfi.h contains the following line:

#define DF\_CAPFNAMES

The \*f.c files contain corresponding conditional sections that produce all-uppercase routine names. For example, the function name Fun can be redefined as FUN:

#ifdef DF\_CAPFNAMES

define Fun FUN

#endif /\* DF\_CAPFNAMES \*/

### Appended Underscores

Differing compiler conventions create a similar problem in their use of the underscore ( \_ ) character. Many compilers, including most C compilers, prepend an underscore to all external symbols in the object module symbol table. The linker then looks for external symbols in other symbol tables with the prefixed underscore.

Many FORTRAN compilers also append an underscore to identify external symbols. Since C compilers do not generally do this, external references in FORTRAN-generated object modules will not recognize externals with the same names in C-generated modules.

For example, the FORTRAN compiler on the CONVEX system places an underscore both at the beginning and at the end of routine names, while the C compiler places an underscore only at the beginning.

Since FUN is a C function, it appears under the name \_FUN in the object module containing it. Now suppose you make the following call in a FORTRAN program:

x = FUN(y)

The FORTRAN compiler will create an object module symbol table with the routine name \_FUN\_. When you link it to the C module, the linker will be unable to link \_FUN and \_FUN\_ and will generate an unsatisfied external reference error.

The HDF Solution

Like the all-uppercase compiler problem, this issue is resolved in the platform-specific sections of hdfi.h and with conditional sections of code that append an underscore to C routine names on platforms where the FORTRAN compiler expects it.

This is implemented as follows: The FNAME\_POST\_UNDERSCORE flag is defined in the platform-specific section of hdfi.h for every platform whose FORTRAN compiler requires appended underscores. Similarly, the FNAME\_PRE\_UNDERSCORE flag is defined on platforms where the FORTRAN compiler expects prepended underscores. The macro FNAME is then defined to append and/or prepend underscores as required.

The FNAME macro is then applied to each routine in the module in which it is actually defined (including in hptroto.h), adding the appropriate underscores.

Consider the above example in which Fun was renamed FUN. The actual definition appears as follows:

#ifdef DF\_CAPFNAMES

define Fun FNAME(FUN)

#endif /\* DF\_CAPFNAMES \*/

### Short Names vs. Long Names

In the C implementations supported by HDF, identifiers may be any length with at least the first 31 characters being significant. FORTRAN compilers differ in the maximum lengths of identifiers that they allow, but all of those supported by HDF allow identifiers to be at least seven characters long.

To deal with the discrepancies between identifier lengths allowed by C and those allowed by the various FORTRAN compilers, a set of equivalent short names has been created for use when programming in FORTRAN. For every HDF routine with a name more than seven characters long, there is an identical routine whose name is seven or fewer characters long.

For example, the routines DFSDgetdims (in dfsd.c) and dsgdims (in dfsdff.f) are functionally identical.

## Differences Between ANSI C and Old C

The current HDF release supports both ANSI C and old C compilers. ANSI C is preferred because it has many features that help ensure portability; unfortunately, many important platforms do not support full ANSI C. The HDF code determines whether ANSI C is available from the flag \_\_STDC\_\_. If ANSI C is available on a platform, then \_\_STDC\_\_ is defined by the compiler.[[3]](#footnote-3)

The most noticeable difference between ANSI C and old C is in the way functions are declared. For example, in ANSI C the function DFSDsetdims() is declared with a single line:

int DFSDsetdims(intn rank, int32 dimsizes[])

In old C the same function is declared as follows:

int DFSDsetdims(rank, dimsizes)

intn rank;

int32 dimsizes[];

HDF accommodates these differences by defining the flag PROTOTYPE in hdfi.h. PROTOTYPE is used for every function declaration in a manner similar to the following example:

#ifdef PROTOTYPE

int DFSDsetdims(intn rank, int32 dimsizes[])

#else

int DFSDsetdims(rank, dimsizes)

intn rank;

int32 dimsizes[];

#endif /\* PROTOTYPE \*/

Note that prototypes are supported by some C compilers that are not otherwise ANSI-conformant. In such situations, PROTOTYPE is defined even though \_\_STDC\_\_ is not.

Another difference between old C and ANSI C is that ANSI C supports function prototypes with arguments. (Old C also supports function prototypes, but without the argument list.) This feature helps in detecting errors in the number and types of arguments. This difference is handled by means of a macro PROTO, which is defined as follows:

#ifdef PROTOTYPE

#define PROTO(x) x

#else

#define PROTO(x) ()

#endif

This macro is applied as in the following example:

extern int32 Hopen

PROTO((char \*path, intn access, int16 ndds));

When PROTOTYPE is defined, PROTO causes the argument list to stay as it is. When PROTOTYPE is not defined, PROTO causes the argument list to disappear.

## Type Differences

Platforms and compilers also differ in the sizes of numbers that they assign to different data types, in their representations of different number types, and in the way they organize aggregates of numbers (especially structures).

### Size differences

The same number type can be different sizes on different platforms. The type int, for example, is 16 bits to many IBM PC compilers, 48 bits to some supercomputer compilers, and 32 bits on most others. This can cause problems that are difficult to diagnose in code like the HDF code, which depends in many places on numbers being the right size.

HDF handles this problem by fully defining all variable types and function data types via typedef, including the number of bits occupied. All parameters, members of structures, and static, automatic, and external variables are so defined .

The HDF data types include the following (types with the prefix u are unsigned).

int8

uint8

int16

uint16

int32

uint32

float32

float64

intn

uintn

For each machine, typedefs are declared that map all of the data types used into the best available types. For example, int32 is defined as follows for Sun's C compiler:

typedef long int int32;

Unfortunately, the HDF data types do not always map exactly to one of the native data types. For example, the Cray UNICOS C compiler does not support a 16-bit data type. In such instances, HDF uses the best available match and care is taken to minimize potential problems.

The data types intn and uintn are for situations where it can be determined that number type size is unimportant and that a 16-bit integer is large enough to hold any value the number can have. In such cases, the native integer type (or unsigned integer type) of the host machine is used. Experience indicates that substantial performance gains can be achieved by using intn or uintn in certain circumstances.

### Number Representation

One of the keys to producing a portable file format is to ensure that numbers that are represented differently on different machines are converted correctly when moved from machine to machine. HDF provides conversion routines to convert between native representations and a standard representation that is actually used in the HDF file. This ensures that HDF data will always be interpreted correctly, regardless of the platform on which it is read or written. Details of this process will be included in a later edition of this manual.

### Byte-order and Structure Representations

Even when the basic bit-representation of constants or aggregates like structures is the same across platforms, the ways that the bits are packed into a word and the order in which the bits are laid out can differ. For example, DEC and Intel-based machines generally order bytes differently from most others. And the C compiler on a Cray, with a 64-bit word, packs structures differently from those on 32-bit word machines.

Differences in byte order among machines are handled in either of two ways. When the data to be written (or read) includes non-integer data and/or a large array of any type of data, conversion routines mentioned in the previous section, “Number Representation,” are invoked. When an individual integer is to be written (or read), an ENCODE or DECODE macro is used.

The following ENCODE and DECODE macros are available for 16-bit and 32-bit integers:

INT16ENCODE

UINT16ENCODE

INT32ENCODE

UINT32ENCODE

INT16DECODE

UINT16DECODE

INT32DECODE

UINT32DECODE

The ENCODE macros write integers to an HDF file in a standard format regardless of the word-size and byte order of the host machine.

Likewise, the DECODE macros read integers from a standard format in an HDF file and provide the integers in the required byte order and word size to the host machine.

Since the ENCODE and DECODE macros deal with both byte order and word size, they are also used in reading and writing record-like structures. For example, an HDF data descriptor consists of two 16-bit fields followed by two 32-bit fields, as implied by the following C declaration:

struct {

uint16 tag;

uint16 ref;

uint32 offset;

uint32 length;

}

Even though this structure might occupy 12 bytes on one platform or 32 bytes on another (e.g., a Cray), it must occupy exactly 12 bytes in an HDF file. Furthermore, some machines represent the numbers internally in different byte orders than others, but the byte order must always be big-endian in an HDF file. The ENCODE and DECODE macros ensure that these values are always represented correctly in HDF files and as presented to any host machine.

## Access to Library Functions

Despite standardization efforts, function libraries often differ in significant ways. At least three types of functions require special treatment in the HDF implementation:

File I/O  
Some platforms use 16-bit values for the element size and the number of elements to write or read, while others use 32-bit values. This must be considered when working with either stream or system level I/O functions (i.e., the functions associated with the fopen() and open() calls).

Memory allocation and release  
First, 16-bit machines use a 16-bit value to indicate the number of bytes to allocate or release at one time. Second, certain operating systems (notably MS Windows and MAC/OS) don't have malloc() and free() calls. These operating systems use handles for allocating memory and require different function calls.

Memory and string manipulation  
These functions (e.g., memcpy(), memcmp(), strcpy(), and strlen()) require slightly different function names under different memory models in MS DOS and under MS Windows than on most other systems.

HDF accommodates these special situations by defining appropriate macros in the machine-specific sections of hdfi.h.

1. The version of C described in the first edition of The C Programming Language, by Brian Kernighan and Dennis Ritchie, published by Prentice-Hall. [↑](#footnote-ref-1)
2. Prior to Version 3.2 of HDF, these files were called df.h and dfi.h. At the time of HDF Version 3.2, the low level interfaces, the general purpose layer of HDF, was completely rewritten and all routine names were changed from df\* to hdf\*. [↑](#footnote-ref-2)
3. \_\_STD\_\_ is generally defined by ANSI-conforming C compilers. Some C compilers are not entirely ANSI-conforming, yet they conform well enough that the HDF implementation can treat them as if they were. In such cases, it is permissible to define \_\_STDC\_\_ by adding the option -D\_\_STDC\_\_ to the cc line in the makefile. [↑](#footnote-ref-3)