Chapter 2

HDF Fundamentals

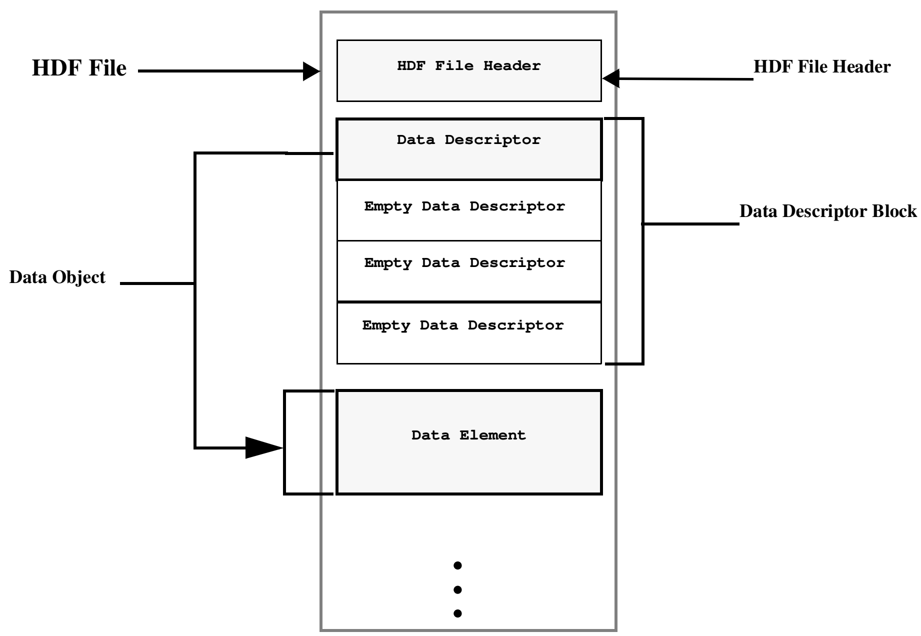
2.1 Chapter Overview

This chapter provides necessary information for the creation and manipulation of HDF files. It includes an overview of the HDF file format, basic operations on HDF files, and programming language issues pertaining to the use of Fortran and ANSI C in HDF programming.

2.2 HDF File Format

An HDF file contains a ***file header***, at least one ***data descriptor block***, and zero or more ***data elements*** as depicted in Figure 2a.

FIGURE 2a **The Physical Layout of an HDF File Containing One Data Object**



The ***file header*** identifies the file as an HDF file. A ***data descriptor block*** contains a number of ***data descriptors***. A data descriptor and a ***data element*** together form a ***data object***, which is the basic conglomerate structure for encapsulating data in the HDF file. Each of these terms is described in the following sections.

2.2.1 File Header

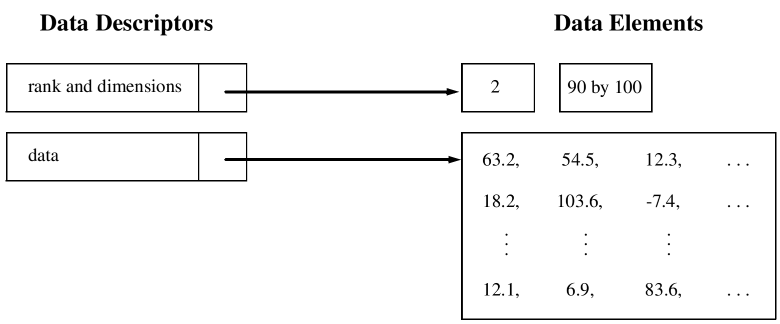
The first component of an HDF file is the file header, which takes up the first four bytes of the HDF file. Specifically, it consists of four one-byte values that are ASCII representations of control characters: the first is a control-N, the second is a control-C , the third is a control-S and the fourth is a control-A (^N^C^S^A).

Note that, on some machines, the order of bytes in the file header might be swapped when the header is written to an HDF file, causing these characters to be written in little-endian order. To maintain the portability of HDF file header data when developing software for such machines, this byte swapping must be counteracted by ensuring the characters are read and written in the desired order.

2.2.2 Data Object

A data object is comprised of a data descriptor and a data element. The data descriptor consists of information about the type, location, and size of the data element. The data element contains the actual data. This organization of HDF data makes HDF files ***self-describing***. Figure 2b shows two examples of data objects.

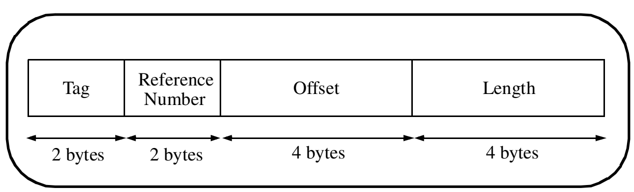
FIGURE 2b **Two Data Objects**



2.2.2.1 Data Descriptor

All data descriptors are twelve bytes long and contain four fields, as depicted in Figure 2c. These fields are: a 16-bit ***tag***, a 16-bit ***reference number***, a 32-bit ***data offset*** and a 32-bit ***data length***.

FIGURE 2c **The Contents of a Data Descriptor**



Tag

A ***tag*** is the data descriptor field that identifies the type of data stored in the corresponding data element. A tag is a 16-bit unsigned integer between 1 and 65,535, and is associated with a mnemonic name to promote ease to use and the readability of user programs.

If a data descriptor has no corresponding data element, the value of its tag is `DFTAG_NULL` (or `0`).

Tags are assigned by the HDF Group as part of the HDF specification. The following are the ranges of tag values and their descriptions:

```
1 to 32,767 - Tags reserved for HDF Group use

32,768 to 64,999 - User-definable tags

65,000 to 65,535 - Tags reserved for expansion of the HDF specification
```

A list of commonly-used tags and their descriptions is included in Appendix A of this document.

Reference Number

For each occurrence of a tag in an HDF file, a unique reference number is assigned by the library with the tag in the data descriptor. A ***reference number*** is a 16-bit unsigned integer and can not be changed during the life of the data object that the reference number specifies.

The combination of a tag and a reference number uniquely identifies the corresponding data object in the file.

Reference numbers are not necessarily assigned consecutively, so it cannot be assumed that the value of a reference number has any meaning beyond providing a way of distinguishing among objects with the same tag. While application programmers may find it convenient to impart some additional meaning to reference numbers in their code, it is emphasized that the HDF library will not internally recognize any such meaning.

Data Offset and Length

The data offset field points to the location of the data element in the file by storing the number of bytes from the beginning of the file to the beginning of the data element. The length field contains the size of the data element in bytes. The data offset and the length are both 32-bit unsigned integers.

2.2.2.2 Data Elements

The data element is the raw data portion of a data object.

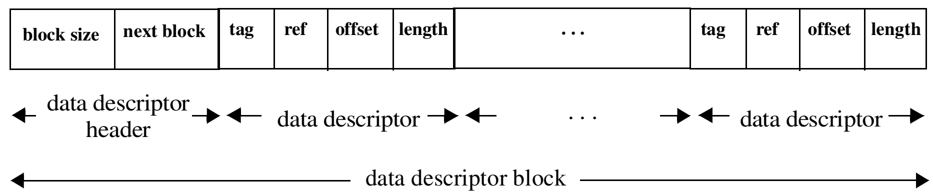
2.2.3 Data Descriptor Block

Data descriptors are physically stored in a linked list of blocks called data descriptor blocks. The relationship between the data descriptor block to the other components of an HDF file is illustrated in Figure 2a on page 7. The individual components of a data descriptor block are depicted in Figure 2d on page 10. Each data descriptor in a data descriptor block is assumed to be associated with a data element unless it contains the tag `DFTAG_NULL` (or `0`),which indicates that there is no associated data element. By default, a data descriptor block contains `16` (defined as `DEF_NDDS`) data descriptors. The user may reset this limit when creating the HDF file. Refer to Section 2.3.3 for more details.

In addition to data descriptors, each data descriptor block contains a ***data descriptor header***. The data descriptor header contains two fields: ***block size*** and ***next block***. The block size field is a 16-bit unsigned integer indicating the number of data descriptors in the data descriptor block. The next block field is a 32-bit unsigned integer indicating the offset of the next data descriptor block, if one exists. The last data descriptor header in the list contains a value of `0` in its next block field.

Figure 2d illustrates the layout of a data descriptor block.

FIGURE 2d **Data Descriptor Block**



2.2.4 Grouping Data Objects in an HDF File

Data objects containing related data in HDF files are usually grouped together by the library. These groups of data objects are called data sets. The HDF user uses the application interface to manipulate data sets in a file. As an example, an 8-bit raster image data set requires three objects: a group object identifying the members of the set, an image object containing the image data, and a dimension object indicating the size of the image.

Data objects are individually accessible even if they are included in a set, therefore data objects can belong to more than one set and sets can be included in larger groups. For example, a palette object included in one raster image set may also be a part of another raster image set if its tag and reference number are included in a data descriptor within that second set.

Additional information about data objects, including the options available for storing them, can be found in the *HDF Specifications Manual* and from the HDF WWW home page at `http://hdf.ncsa.uiuc.edu/`.

2.3 Basic Operations on HDF Files Using the Multifile Interfaces

This section describes the basic file operations, some of which are required in working with HDF files using the multifile interfaces. Except for the SD interface, all applications using other multi-file interfaces must explicitly use the routines **Hopen** and **Hclose** to control accesses to the HDF files. In an application using the HDF file format, the file is accessed via its identifier, referred to as ***file identifier***. The following subsections describe the file identifier and the basic file operations common to most multifile interfaces.

2.3.1 File Identifiers

The HDF programming model specifies that a data file is first explicitly created or opened by an application, manipulated, then explicitly closed by the application. A file identifier is a unique number that the HDF library assigns to an HDF file when creating or opening the file. The HDF library creates the file identifier for an HDF file when given its file name, as represented in the native file system. Interface routines use only the file identifier to access and manipulate the file. When all operations on the file are complete, the file identifier must be discarded by explicitly closing the file before terminating the application.

As every file is assigned its own identifier, the order in which files are accessed is very flexible. For example, it is valid to open a file and obtain an identifier for it, then open a second file without closing the first file or disposing of the first file identifier. The only requirement made by HDF is that all file identifiers be individually discarded before the termination of the calling program.

File identifiers created by the routine of one HDF interface can be used by the routines of any other interfaces, except SD's.

2.3.2 Opening HDF Files: Hopen

The routine **Hopen** creates or opens an HDF data file, depending on the access mode specified, and returns the file identifier that the HDF library has assigned to the file. The **Hopen** syntax is as follows:

```
C:          file_id = Hopen(filename, access_mode, num_dds_block);

FORTRAN:    file_id = hopen(filename, access_mode, num_dds_block)
```

The **Hopen** parameters are defined in Table 2A and the following discussion.

TABLE 2A **Hopen Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Hopen [int32] **(hopen)**	filename	char *	character*(*)	File name
	access_mode	intn	integer	File access mode
	num_dds_block	int16	integer	Number of data descriptors in a data descriptor block

The parameter `filename` is a character string representing the name of the HDF file to be accessed.

The parameter `access_mode` specifies how the file should be accessed. All the access modes are listed in Table 2B. If the access mode is `DFACC_CREATE` and the file already exists, the file will be replaced by the new one. If the access mode is `DFACC_READ` and the file does not exist, **Hopen** will return `FAIL` (or `-1`). If the access mode is `DFACC_WRITE` and the file does not exist, a new file will be created.

The parameter `num_dds_block` specifies the number of data descriptors in a block when the access mode specified is create. If the access mode is not create, the value of `num_dds_block` is ignored. The default number of data descriptors in a block is 16 (defined as `DEF_NDDS`) data descriptors. The user may specify `0` to keep the default or any non-negative integer to reset this limit when creating the HDF file.

Note that, in the SD interface, **SDstart** is used to open files instead of **Hopen**. (Refer to Chapter 3, *Scientific Data Sets (SD API)*, of this document for more information on **SDstart**.)

TABLE 2B **File Access Code Flags**

File Access Flag	Flag Value	Description
DFACC_READ	1	Read access
DFACC_WRITE	2	Read and write access
DFACC_CREATE	4	Create with read and write access

2.3.3 Closing HDF Files: Hclose

The **Hclose** routine closes the file designated by the file identifier specified by the parameter `file_id`. The **Hclose** syntax is as follows:

```
C:          status = Hclose(file_id);

FORTRAN:    status = hclose(file_id)
```

Hclose returns a value of `SUCCEED` (or `0`) if successful or `FAIL` (or `-1`) otherwise. The parameter name and type are listed in Table 2C. Refer also to the *HDF Reference Manual* for additional information regarding **Hclose**.

Note that **Hclose** is not used to close files in the SD interface. **SDend** is used for this purpose. (Refer to Chapter 3, *Scientific Data Sets (SD API)*, of this document for more information on **SDend**.)

TABLE 2C **Hclose Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Hclose [intn] **(hclose)**	file_id	int32	integer	File identifier

2.3.4 Getting the HDF Library and File Versions: Hgetlibversion and Hgetfileversion

Hgetlibversion returns the version of the HDF library currently being used, as well as additional textual information regarding the library. The parameter names and data types are listed in Table 2D. Refer also to the *HDF Reference Manual* for additional information regarding **Hgetlibversion**.

Hgetfileversion returns the version information of the HDF file specified by the parameter `file_id`, as well as additional textual information regarding the nature of the file. The parameter names and data types are listed in Table 2D. Refer also to the *HDF Reference Manual* for additional information regarding **Hgetfileversion**.

The syntax of these routines is as follows:

```
C:         status = Hgetlibversion(&major_v, &minor_v, &release, string);
           status = Hgetfileversion(file_id, &major_v, &minor_v,
                          &release, string);

FORTRAN:   status = hglibver(major_v, minor_v, release, string)
           status = hgfilver(file_id, major_v, minor_v, release, string)
```

Both routines return a value of `SUCCEED` (or `0`) if successful or `FAIL` (or `-1`) otherwise.

TABLE 2D **Hgetlibversion and Hgetfileversion Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Hgetlibversion [intn] **(hglibver)**	major_v	uint32*	integer	Major version number
	minor_v	uint32*	integer	Minor version number
	release	uint32*	integer	Complete library version number
	string	char*	character*(*)	Additional information about the library version
Hgetfileversion [intn] **(hgfilver)**	file_id	int32	integer	File identifier
	major_v	uint32*	integer	Major version number
	minor_v	uint32*	integer	Minor version number
	release	uint32*	integer	Complete library version number
	string	char*	character*(*)	Additional information about the library version

2.4 Programming Issues

This section introduces information relevant to the process of developing programs that use the HDF library, such as the names of necessary header files, lists of common definitions and issues concerning FORTRAN-77 and C programming.

2.4.1 Header File Information

The header file "hdf.h" must be included in every HDF application program written in C, except for programs that call routines in the SD interface. The header file "mfhdf.h" must be included in all programs that call SD interface routines.

Fortran programmers who use compilers that allow file inclusion can include the files "hdf.inc" and "dffunc.inc". If a Fortran compiler that does not support file inclusion is used, HDF library definitions must be explicitly defined in the Fortran program as they are included in the header files of the HDF library.

2.4.2 HDF Definitions

The HDF library provides several sets of definitions which can be used easily in the user applications. These sets include the definitions of the data types, the data type flags, and the limits that set various maximum values. The definitions of the data types supported by HDF are located in the "hdf.h" header file, and the data type flags are located in the "hntdefs.h" header file. Both are also included in Table 2E on page 14, Table 2F on page 14, and Table 2G on page 15. HDF data types are used for portability in the declaration of variables, and data type flags are used as parameters in various HDF interface routines.

2.4.2.1 Standard HDF Data Types

The definitions of the fundamental data types are in Table 2E on page 14. Although `DFNT_FLOAT` (or `5`), `DFNT_UCHAR` (or `3`), and `DFNT_CHAR` (or `4`) have not been added to this table, they are also supported by the HDF library for backward compatibility.

If the machine used is big-endian, using these data types will result in no byte-order conversion being performed. If the machine used is little-endian, the library will convert the byte-order of the variables to big-endian.

TABLE 2E **Standard HDF Data Types and Flags**

HDF Data Type	Data Type Flag and Value	Description
char8	**DFNT_CHAR8 (4)**	8-bit character type
uchar8	**DFNT_UCHAR8 (3)**	8-bit unsigned character type
int8	**DFNT_INT8 (20)**	8-bit integer type
uint8	**DFNT_UINT8 (21)**	8-bit unsigned integer type
int16	**DFNT_INT16 (22)**	16-bit integer type
uint16	**DFNT_UINT16 (23)**	16-bit unsigned integer type
int32	**DFNT_INT32 (24)**	32-bit integer type
uint32	**DFNT_UINT32 (25)**	32-bit unsigned integer type
float32	**DFNT_FLOAT32 (5)**	32-bit floating-point type
float64	**DFNT_FLOAT64 (6)**	64-bit floating-point type

Fortran programmers should refer to Section 2.4.3 on page 16 for a discussion of the Fortran data types.

2.4.2.2 Native Format Data Types

When a native format data type is specified, the corresponding numbers are stored in the HDF file exactly as they appear in memory, without conversion. For example, on a Cray Y-MP, 8 bytes of memory, or one Cray word, is used to store most integers. Therefore, an 8-bit signed integer, represented by the `DFNT_INT32` flag, on a Cray Y-MP uses 8 bytes of memory. Consequently, when the data type `DFNT_NATIVE | DFNT_INT32` (`DFNT_NATIVE` bytewise-ORed with `DFNT_INT32`) is used on a Cray Y-MP to specify the data type of an HDF SDS or vdata, each integer stored in the HDF file is 8 bytes.

The method for constructing the data type flag for each native data type described in the previous paragraph is used for any of the native data types: the `DFNT_NATIVE` flag is bitwise-ORed with the flag of the corresponding standard data type.

If the user is on a big-endian machine, using native data types will result in no conversion. If the user is on a little-endian machine, the HDF library will perform little-to-big-endian conversion.

The definitions of the native format data types and the corresponding data type flags appear in Table 2F.

TABLE 2F **Native Format Data Type Definitions**

HDF Data Type	HDF Data Type Flag and Value	Description
int8	**DFNT_NINT8 (4116)**	8-bit native integer type
uint8	**DFNT_NUINT8 (4117)**	8-bit native unsigned integer type
int16	**DFNT_NINT16 (4118)**	16-bit native integer type
uint16	**DFNT_NUINT16 (4119)**	16-bit native unsigned integer type
int32	**DFNT_NINT32 (4120)**	32-bit native integer type
uint32	**DFNT_NUINT32 (4121)**	32-bit native unsigned integer type
float32	**DFNT_NFLOAT32 (4101)**	32-bit native floating-point type
float64	**DFNT_NFLOAT64 (4102)**	64-bit native floating-point type

2.4.2.3 Little-Endian Data Types

HDF also provides a "little-endian" option to suppress any rearranging of byte ordering from little- to big-endian. This is primarily for users of Intel-based machines who do not want to incur the cost of reordering data when writing to an HDF file. Note that direct conversions are supported between little-endian and all other byte-order formats supported by HDF.

The method for constructing the data type flag for each little-endian data type is similar to the method for constructing native format data type flags: the `DFNT_LITEND` flag is bitwise-ORed with the flag of the corresponding standard data type.

If the user is on a little-endian machine, using these data types will result in no conversion. If the user is on a big-endian machine, the HDF library will perform big-to-little-endian conversion.

The definitions of the little-endian data types and the corresponding data type flags appear in Table 2G.

TABLE 2G **Little-Endian Format Data Type Definitions**

HDF Data Type	HDF Data Type Flag and Value	Description
int8	**DFNT_LINT8 (16404)**	8-bit little-endian integer type
uint8	**DFNT_LUINT8 (16405)**	8-bit little-endian unsigned integer type
int16	**DFNT_LINT16 (16406)**	16-bit little-endian integer type
uint16	**DFNT_LUINT16 (16407)**	16-bit little-endian unsigned integer type
int32	**DFNT_LINT32 (16408)**	32-bit little-endian integer type
uint32	**DFNT_LUINT32 (16409)**	32-bit little-endian unsigned integer type
float32	**DFNT_LFLOAT32 (16389)**	32-bit little-endian floating-point type
float64	**DFNT_LFLOAT64 (16390)**	64-bit little-endian floating-point type

2.4.2.4 Tag Definitions

These definitions identify the object tags defined and used by the HDF interface library. The concept of object tags is introduced in Section 2.2.2.1 on page 8, and a list of tags can be found in Appendix A of this manual. Note that tags can also identify properties of data objects.

2.4.2.5 Limit Definitions

These definitions declare the maximum size of specific data object parameters, such as the maximum length of a vdata field or the maximum number of objects in a vgroup. They are located in the header file "hlimits.h". A selection of the most-commonly-used limit definitions appears in Table 2H.

TABLE 2H **Limit Definitions**

Definition Name	Definition Value	Description
FIELDNAMELENMAX	128	Maximum length of a vdata field in bits - 16 characters
VSNAMELENMAX	64	Maximum length of a vdata name in bytes - 64 characters
VGNAMELENMAX	64	Maximum length of a vgroup name in bytes - 64 characters
VSFIELDMAX	256	Maximum number of fields per vdata (64 for Macintosh)
VDEFAULTBLKSIZE	4096	Default block size in a vdata
VDEFAULTNBLKS	32	Default number of blocks in a vdata
MAXNVELT	64	Maximum number of objects in a vgroup
MAX_ORDER	65535	Maximum order of a vdata field

MAX_FIELD_SIZE	65535	Maximum length of a field
MAX_NC_DIMS	5000	Maximum number of dimensions per file
MAX_NC_ATTRS	3000	Maximum number of file or variable attributes
MAX_NC_VARS	5000	Maximum number of file attributes
MAX_NC_DIMS	32	Maximum number of variable attributes
MAX_NC_NAME	256	Maximum length of a name - NC interface
MAX_PATH_LEN	1024	Maximum length of an external file name
MAX_FILE	32	Maximum number of open files
MAX_GROUPS	8	Maximum number of groups
MAX_GR_NAME	256	Maximum length of a name - GR interface
MAX_VAR_DIMS	32	Maximum number of dimensions per variable
MAX_REF	65535	The largest number that will fit into a 16-bit word reference variable
MAX_BLOCK_SIZE	65536	Maximum size of blocks in linked blocks

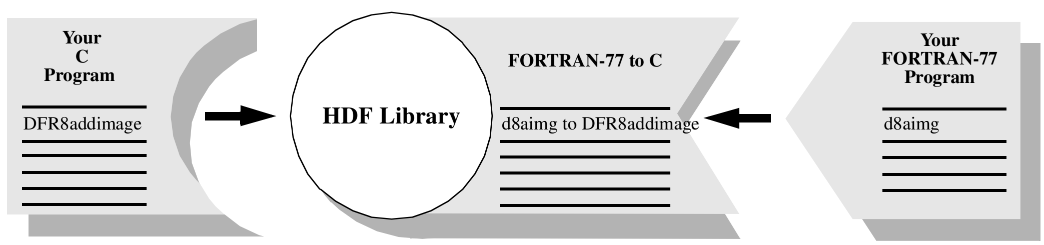
2.4.3 FORTRAN-77 and C Language Issues

HDF provides both FORTRAN-77 and C versions of most of its interface routines. In order to make the FORTRAN-77 and C versions of each routine as similar as possible, some compromises have been made in the process of simplifying the interface for both programming languages.

FORTRAN-77-to-C Translation

Nearly all of the HDF library code is written in C. A FORTRAN-77 HDF interface routine translates all parameter data types to C data types, then calls the C routine that performs the functionality of the interface routine. For example, **d8aimg** is the FORTRAN-77 equivalent for **DFR8addimage**. Calls to either routine execute the same C code that adds an 8-bit raster image to an HDF file. See Figure 2e.

FIGURE 2e **Use of a Function Call Converter to Route FORTRAN-77 HDF Calls to the C Library**



Case Sensitivity

FORTRAN-77 identifiers generally are not case sensitive, whereas C identifiers are. Although all of the FORTRAN-77 routines shown in this manual are written in lower case, FORTRAN-77 programs can generally call them using either upper- or lower-case letters without loss of meaning.

Name Length

Because some FORTRAN-77 compilers only interpret identifier names with seven or fewer characters, the first seven characters of the FORTRAN-77 HDF routine names are unique.

Header Files

The inclusion of header files is not generally permitted by FORTRAN-77 compilers. However, it is sometimes available as an option. On UNIX systems, for example, the macro processors `m4` and `cpp` let the compiler include and preprocess header files. If this capability is not available, the user may have to copy the declarations, definitions, or values needed from the files "dffunc.inc" and "hdf.inc" into the user application. If the capability is available, the files can be included in the Fortran code. These two files reside in the include directory after the library is installed on the user's system.

Data Type Specifications

When mixing machines, compilers, and languages, it is difficult to maintain consistent data type definitions. For instance, on some machines an integer is a 32-bit quantity and on others, a 16-bit quantity. In addition, the differences between FORTRAN-77 and C lead to difficulties in describing the data types found in the argument lists of HDF routines. To maintain portability, the HDF library expects assigned names for all data types used in HDF routines. See Table 2I.

TABLE 2I

Correspondence Between Fortran and HDF C Data Types

Data Type	FORTRAN	C
8-bit signed integer	`character*1` **	`int8`
8-bit unsigned integer	`character*1`	`uint8`
16-bit signed integer	`integer*2`	`int16`
16-bit unsigned integer	Not supported	`uint16`
32-bit signed integer	`integer*4` **	`int32`
32-bit unsigned integer	Not supported	`uint32`
32-bit floating point number	`real*4` **	`float32`
64-bit floating point number	`real*8` **	`float64`
Native signed integer	`integer`	`intn`
Native unsigned integer	Not supported	`uintn`
** if the compiler supports this data type		

When using a FORTRAN-77 data type that is not supported, the general practice is to use another data type of the same size. For example, an 8-bit signed integer can be used to store an 8-bit unsigned integer variable.

String and Array Specifications

The following conventions are followed in the specification of arrays in this manual:

- `character*(*)` defines a string of an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold the data to be stored in the string.
- `real x(*)` means that `x` refers to an array of reals of indefinite size and of indefinite rank. It is the responsibility of the calling program to allocate an actual array with the correct number of dimensions and dimension sizes.
- `<valid numeric data type> x` means that `x` may have one of the numeric data types listed in the Description column of Table 2I above.
- `<valid data type> x` means that `x` may have any of the data types listed in the Description column of Table 2I above.

FORTRAN-77 and ANSI C

As much as possible, we have ensured that the HDF interface routines conform to the implementations of Fortran and C that are in most common use today, namely FORTRAN-77 and ANSI C.

As Fortran-90 is a superset of FORTRAN-77, HDF programs should compile and run correctly when using a Fortran-90 compiler. However, an HDF library interface that makes full use of Fortran-90 enhancements is being considered.