NCSA HDF
Specification and Developer's Guide

Version 3.2

September 1993

University of Illinois at Urbana-Champaign

READ ME NOW

If you want to see more software like NCSA HDF, you need to send us a letter, email or US mail, telling us what you are doing with NCSA HDF. We need to know: (1) What science you are working on—an abstract of your work would be fine; and (2) How NCSA HDF has helped you, for example, by increasing your productivity or allowing you to do things you could not do before.

We encourage you to cite the use of NCSA HDF, and any other NCSA software you have used, in your publications. A bibliography of your work would be extremely helpful.

NOTE: This is a new kind of shareware. You share your science and successes with us, and we can get more resources to share more software like NCSA HDF with you.

NCSA Contacts

Mail user feedback, bugs, and software and manual suggestions to:

NCSA Software Tools Group
HDF
152 Computing Applications Bldg.
605 E. Springfield Ave.
Champaign, IL 61820

Send commmunications via electronic mail to one of the following:

Bug Suggestions
bugs@ncsa.uiuc.edu
bugs@ncsavmsa.bitnet

All Other Communications
softdev@ncsa.uiuc.edu
softdev@ncsavmsa.bitnet

Disclaimer

Trademark Acknowledgments

Table of contents

I Introduction

Overview

The Hierarchical Data Format (HDF) was designed to be an easy, straight-forward, and self-describing means of sharing scientific data among people, projects, and types of computers. An extensible header and carefully crafted internal layers provide a system that can grow as scientific data-handling needs evolve.

This document, the *NCSA HDF Specification and Developer's Guide*, fully defines HDF and its interfaces, discusses criteria employed in its development, and provides guidelines for developers working on HDF itself or building applications that employ HDF.

This introduction provides a brief overview of HDF capabilities and design.

Why HDF?

A fundamental requirement of scientific data management is the ability to access as much information in as many ways, as quickly and easily as possible. A data storage and retrieval system that facilitates these capabilities must provide the following features:

Support for scientific data and metadata
Scientific data is characterized by a variety of data types and representations, data sets (including images) that can be extremely large and complex, and the need to attach accompanying attributes, parameters, notebooks, and other metadata. Metadata, supplementary data that describes the basic data, includes information such as the dimensions of an array, the number type of the elements of a record, or a color lookup table (LUT).

Support for a range of hardware platforms
Data can originate on one machine only to be used later on many different machines. Scientists must be able to access data and metadata on as many hardware platforms as possible

Support for a range of software tools
Scientists need a variety of software tools and utilities for easily searching, analyzing, archiving, and transporting the data and metadata. These tools range from a library of routines for reading and writing data and metadata, to small utilities that simply display an image on a console, to full-blown database retrieval systems that provide multiple views of thousands of sets of data and metadata.

Rapid data transfer
Both the size and the dispersion of scientific data sets require that mechanisms exist to get the data from place to place rapidly.

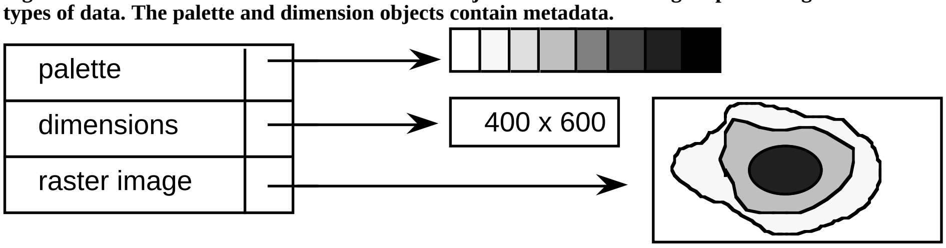
Extendibility
As new types of information are generated and new kinds of science are done, a means must be provided to support them.

What is HDF?

The HDF Structure

HDF is a self-describing extensible file format using tagged objects that have standard meanings. The idea is to store both a known format description and the data in the same file. HDF tags describe the format of the data because each tag is assigned a specific meaning: the tag `DFTAG_LUT` stands for color palette, the tag `DFTAG_RI` stands for 8bit raster image, and so on (see Figure 1). A program that has been written to understand a certain set of tag types can scan the file for those tags and process the data. This program also can ignore any data that is beyond its scope.

Figure I.1 Raster Image Set in an HDF File . The set has three data objects with different tags representing three different types of data. The palette and dimension objects contain metadata.



The set of available data objects encompasses both primary data and metadata. Most HDF objects are machine- and medium-independent, physical representations of data and metadata.

HDF Tags

The HDF design assumes that we cannot know *a priori* what types of data objects will be needed in the future, nor can we know how scientists will want to view that data. As science progresses, people will discover new types of information and new relationships among existing data. New types of data objects new tags will be created to meet these expanding needs. To avoid unnecessary proliferation of tags and to ensure that all tags are available to potential users who need to share data, a portable public domain library is available that interprets all public tags. The library contains user interfaces designed to provide views of the data that are most natural for users. As we learn more about the way scientists need to view their data, we can add user interfaces that reflect data models consistent with those views.

Types of Data and Structures

HDF currently supports the most common types of data and metadata that scientists use, including multidimensional gridded data, 2-dimensional raster images, polygonal mesh data, multivariate data sets, finite-element data, non-Cartesian coordinate data, and text.

In the future there will almost certainly be a need to incorporate new types of data, such as voice and video, some of which might actually be stored on other media than the central file itself. Under such circumstances, it may become desirable to employ the concept of a *virtual file*. A virtual file functions like a regular file but does not fit our normal notion of a monolithic sequence of bits stored entirely on a single disk or tape.

HDF also makes it possible for the user to include annotations, titles, and specific descriptions of the data in the file. Thus, files can be archived with human-readable information about the data and its origins

One collection of HDF tags supports a hierarchical grouping structure called *Vset* that allows scientists to organize data objects within HDF files to fit their views of how the objects go together, much as a person in an office or laboratory organizes information in folders, drawers, journal boxes, and on their desktops.

Backward and Forward Compatibility

An important goal of HDF is to maximize backward and forward compatibility among its interfaces. This is not always achievable, because data formats must sometimes change to enhance performance, to correct errors, or for other reasons. However, whenever possible, HDF files should not become out of date. For example, suppose a site falls far behind in the HDF standard so its users can only work with the portions of the specification that are three years old. Users at this site might produce files with their old HDF software then read them with newer software designed to work with more advanced data files. The newer software should still be able to read the old files.

Conversely, if the site receives files that contain objects that its HDF software does not understand, it should still be able to list the types of data in the file. It should also be able to access all of the older types of data objects that it understands, despite the fact that the older types of data objects are mixed in with new kinds of data. In addition, if the more advanced site uses the text annotation facilities of HDF effectively, the files will arrive with complete human-readable descriptions of how to decipher the new tag types.

Calling Interfaces

To present a convenient user interface made up of something more usable than a list of tag types with their associated data requirements, HDF supports multiple calling interfaces.

The *low level calling interfaces* are used to manipulate tags and raw data, for error handling, and to control the physical storage of data. These interfaces are designed to be used by developers who are providing the higher level interfaces for applications like raster image storage or scientific data archiving.

The *application interfaces*, at the next level, include several modules specifically designed to simplify the process of storing and accessing specific types of data. For example, the palette interface is designed to handle color palettes and lookup tables while the scientific data interface is designed to handle arrays of scientific data. If you are primarily interested in reading or writing data to HDF files, you will spend most of your time working with the application interfaces.

The *HDF utilities* and *NCSA applications*, at the top level, are special purpose programs designed to handle specific tasks or solve specific problems. The utilities provide a command line interface for data management. The applications provide solutions for problems in specific application areas and often include a graphic user interface. Several *third party applications* are also available at this level.

Machine Independence

An important issue in data file design is that of machine independence or transportability. The HDF design defines standard representations for storing all data types that it supports. When data is written to a file, it is typically written in the standard HDF representation. The conversion is handled by the HDF software and need not concern the user. Users may override this convention and install their own conversion routines, or they may write data to a file in the native format of the machine on which it was generated.

Some History

In 1987 a group of users and software developers at NCSA searched for a file format that would satisfy NCSA's data needs. There were some interesting candidates, but none that were in the public domain, were targeted to scientific data, and yet were sufficiently general and extensible. In the course of several months, borrowing concepts from several existing formats, the group designed HDF.

The first version of HDF was implemented in the spring and summer of 1988. It included a general purpose interface and an 8-bit raster image interface. In the fall of 1988, a scientific data set interface was designed and implemented, enabling HDF users to store multidimensional arrays and related data. Soon thereafter interfaces were implemented for storing color palettes, 24-bit raster images, and annotations.

In 1989, it became clear that there was a need to support a general grouping structure and unstructured data such as that used to represent polyhedra in graphical applications. This led to Vsets, whose interface routines were implemented as a separate HDF library.

Also in 1989 it became clear that the existing general purpose layer was not sufficiently powerful to meet anticipated future needs and that the coding could use a substantial overhaul. From this, the long process of redesigning the lower layers of HDF began. The first version incorporating extended tags and the new lower layers of HDF was released in the summer of 1992 as HDF Version 3.2.

This release, HDF Version 3.3, provides alternative physical storage methods (external and linked block data elements) through extended tags, JPEG data compression, changes to some Vset interface functions, access to netCDF files through a complete netCDF interface,[1] hyperslab access routines for old-style SDS objects, and various performance improvements.

About This Document

This document is designed for software developers who are designing applications or routines for use with HDF files and for users who need detailed information about HDF. Users who are interested in using HDF to store or manipulate their data will not normally need the kind of detail presented in this manual. They should instead consult one of the user-level documents:

- Versions 3.2 and earlier
  - *NCSA HDF Calling Interfaces and Utilities*
  - *NCSA HDF Vset*
- Version 3.3
  - *Getting Started with NCSA HDF*
  - *NCSA HDF User's Guide*
  - *NCSA HDF Reference Manual*

Someone using third-party software that uses HDF may also have to consult a manual for that software.

Document Contents

The *NCSA HDF Specification and Developer's Guide* contains the following chapters and appendix:

Chapter 1: Basic Structure of HDF Files
: Introduces and describes the components and organization of HDF files

Chapter 2: Software Overview
: Describes the organization of the software layers that make up the basic HDF library and provides guidelines for writing HDF software

Chapter 3: General Purpose Interface
: Describes the low level HDF routines that make up the general purpose interface

Chapter 4: Sets and Groups
: Explains the roles of sets and groups in an HDF file, and describes raster image sets, scientific data sets, and Vsets

Chapter 5: Annotations
: Explains the use of annotations in HDF files

Chapter 6: Tag Specifications
: Describes the tag identification space, the extended tag structure, and all of the NCSA-supported tags

Chapter 7: Portability Issues
: Describes the measures taken to maximize HDF portability across platforms and to ensure that HDF routines are available to both C and FORTRAN programs

1 NetCDF is a network-transparent derivative of the original CDF (Common Data Format) developed by the National Aeronautics and Space Administration (NASA). It is used widely in atmospheric sciences and other disciplines requiring very large data structures. NetCDF is in the public domain and was developed at the Unidata Program Center in Boulder, Colorado.

Appendix A: Tags and Extended Tag Labels
Presents a list of NCSA-supported HDF tags and a list of labels used with extended tags

Conventions Used in This Document

Most of the descriptive text in this guide is printed in 10 point New Century Schoolbook. Other typefaces have specific meanings that will help the reader understand the functionality being described.

New concepts are sometimes presented in italics on their first occurrence to indicate that they are defined within the paragraph.

Cross references within the specification include the title of the referenced section or chapter enclosed in quotation marks. (E.g., See Chapter 1, "The Basic Structure of HDF Files," for a description of the basic HDF file structure.)

References to documents italicize the title of the document. (E.g., See the guide *Getting Started with NCSA HDF* to familiarize yourself with the basic principles of using HDF.)

Literal expressions and *variables* often appear in the discussion. Literal expressions are presented in Courier while variables are presented in italic Courier. A literal expression is any expression that would be entered exactly as presented, e.g., commands, command options, literal strings, and data. A variable is an expression that serves as a place holder for some other text that would be entered. Consider the expression `cp` *`file1 file2`*. `cp` is a command name and would be entered exactly as it appears, so it is printed in bold Courier. But *`file1`* and *`file2`* are variables, place holders for the names of real files, so they are printed in italic bold Courier; the user would enter the actual filenames.

This guide frequently offers sample *command lines*. Sometimes these are examples of what might be done; other times they are specific instructions to the user. Command lines may appear within running text, as in the preceding paragraph, or on a separate line, as follows:

```
cp file1 file2
```

Command lines always include one or more literal expressions and may include one or more variables, so they are printed in Courier and italic Courier as described above.

Keys that are labeled with more than one character, such as the RETURN key, are identified with all uppercase letters. Keys that are to be pressed simultaneously or in succession are linked with a hyphen. For example, “press CONTROL-A” means to press the CONTROL key then, without releasing the CONTROL key, press the A key. Similarly, “press CONTROL-SHIFT-A “ means to press the CONTROL and SHIFT keys then, without releasing either of those, press the A key.

Table I.1 summarizes the use of typefaces in the technical discussion (i.e., everything except references and cross references).

Table I.1 Meaning of entry format notations

Type	Appearance	Example	Entry Method
Literal expression (commands, literal strings, data)	Courier	`dothis`	Enter the expression exactly as it appears.
Variables	Italic Courier	*`filename`*	Enter the name of the file or the specific data that this expression represents.
Special keys	Uppercase	RETURN	Press the key indicated.
Key combinations	Uppercase with hyphens between key names	CONTROL-A	While holding down the first one or two keys, press the last key.

Program listings and *screen listings* are presented in a boxed display in Courier type such as in Figure I.2, "Sample Screen Listing." When the listing is intended as a sample that the reader will use for an exercise or model, variables that the reader will change are printed in italic Courier.

Figure I.2 Sample screen listing

```
mars_53% ls -F
MinMaxer/                    net.source
mars_54% cd MinMaxer
mars_55% ls -F
list.MinMaxer                minmaxer.v1.04/
mars_56% cd minmaxer.v1.04
mars_57% ls -F
COPYRIGHT                    minmaxer.bin/           source.minmaxer/
README                       sample/                 source.triangulation/
mars_58%
```

Chapter 1 Basic Structure of HDF Files

Chapter Overview

This chapter introduces and describes the components and organization of Hierarchical Data Format (HDF) files.

File Header

The first component of an HDF file is the *file header* (FH), which takes up the first four bytes in an HDF file. The file header is a signature that indicates that the file is an HDF file. Specifically, it is a 32-bit magic number with the hexadecimal value 0e031301.

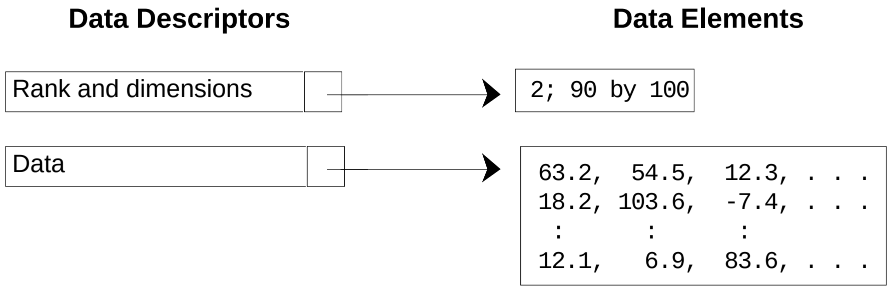
> **Note:** To ensure portability, the programmer must ensure that the hexadecimal value in an HDF file header is written in big-endian order.

HDF assumes big-endian order in reading and writing files. The order of bytes in the file header might be swapped on some machines when the HDF file header is written, causing these characters to be written in little-endian order. To maintain HDF file portability when developing software for such machines, you must make sure the characters are read and written in the exact order shown.

Data Objects

The basic building block of an HDF file is the *data object*, which contains both data and information about the data. A data object has two parts: a 12-byte *data descriptor* (DD) and a *data element*. Figure 1.1 illustrates two data objects.

Figure 1.1 Two Data Objects



As the names imply, the data descriptor provides information about the data; the data element is the data itself. In other words, all data in an

HDF file has information about itself attached to it. In this sense, HDF files are *self-describing* files.

Data Descriptor (DD)

A data descriptor (DD) has four fields: a 16-bit tag, a 16-bit reference number, a 32-bit data offset, and a 32-bit data length. These are depicted in Figure 1.2 and are briefly described in Table 1.1. Explanations of each part appear in the paragraphs following Table 1.1.

Figure 1.2 A Data Descriptor (DD)

Table 1.1 Parts of a Data Descriptor

Part	Description	
Tag/ref (data identifier)	Unique identifier for each data element	
	Tag	Type of data in a data element
	Reference number	Number distinguishing data element from others with the same tag
Offset	Byte offset of data element from beginning of file	
Length	Length of data element	

Note: Only the full tag/ref uniquely identifies a data element.

Tag/ref (Data Identifier)

A tag and its associated reference number (abbreviated as tag/ref) uniquely identify a data element in an HDF file. The tag/ref combination is also known as a *data identifier*.

Tag

A *tag* is the part of a data descriptor that tells what kind of data is contained in the corresponding data element. A tag is actually a 16-bit unsigned integer between 1 and 65535, but every tag is also given a name that programs can refer to instead of the number. If a DD has no corresponding data element, its tag is `DFTAG_NULL`, indicating that no data is present. A tag may never be zero.

Tags are assigned by NCSA as part of the specification of HDF. The following ranges are to be used to guide tag assignment:

00001 – 32767 reserved for NCSA use
32768 – 64999 user-definable
65000 – 65535 reserved for expansion of the format

Chapter 6, "Tag Specifications," provides full specifications for all currently supported HDF tags. Appendix A, "Tags and Extended Tag Labels," lists the current tag assignments. See the section "Some HDF Conventions" in Chapter 2, "Software Overview," for more information on allocating tags.

Reference Number

Tags are not necessarily unique in an HDF file; there may be more than one data element of a given type. Therefore, each tag is associated with a unique *reference number* in the data descriptor.

Reference numbers are not necessarily assigned consecutively, so you cannot assume that the actual value of a reference number has any meaning beyond providing a way of distinguishing among elements with the same tag. Furthermore, reference numbers are only unique for data elements with the same tag; two 8-bit raster images will never have the same reference number but an 8-bit raster image and a 24-bit raster image might.

Reference numbers are 16-bit unsigned integers.

Data Offset and Length

Note: All offsets are from the beginning of the file; they are not relative.

The *data offset* states the byte position of the corresponding data element from the beginning of the file. The *length* states the number of bytes occupied by the data element.

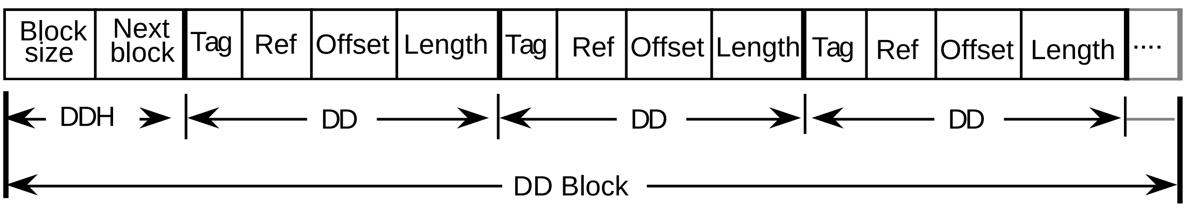
Offset and length are both 32-bit unsigned integers.

DD Blocks

Data descriptors are stored physically in a linked list of blocks called *data descriptor blocks* or DD blocks. The individual components of a DD block are depicted in Figure 1.3. All of the DDs in a DD block are assumed to contain significant data unless they have the tag DFTAG_NULL (no data).

In addition to its DDs, each data descriptor block has a *data descriptor header* (DDH). The DDH has two fields: a *block size* field and a *next block* field. The block size field is a 16-bit unsigned integer that indicates the number of DDs in the DD block. The next block field is a 32-bit unsigned integer giving the offset of the next DD block, if there is one. The DDH of the last DD block in the list contains a 0 in its next block field.

Figure 1.3 Model of a Data Descriptor Block



Since the default number of DDs in a DD block is defined when the HDF library is compiled, changing the default requires recompilation.

Data Element

A *data element* is the raw data portion of a data object. Its data type can be determined by examining its tag, but other interpretive information may be required before it can be processed properly.

Each data element is stored as a set of contiguous bytes starting at the offset and with the length specified in the corresponding DD.[1]

Exceptions

Note that the data object identified by the tag `DFTAG_MT` does not adhere to the standards described above; it consists of the tag immediately followed by four number types. Since there can be only one `DFTAG_MT` tag in an HDF file, there is no need for a reference number. Since all the data can be stored in the DD with the tag, there is no need for a data element and the offset and length are unnecessary.

Several other tags, such as `DFTAG_NULL` and `DFTAG_JPEG`, serve as binary flags and convey all the required information by the mere fact of their presence in an HDF file. These tags therefore point to no data element and have offset and length values of `0`. Consider these examples: `DFTAG_NULL` indicates a data object containing no data; `DFTAG_JPEG` indicates that an associated data object, indicated by another tag, contains a JPEG data image. The descriptions of these tags include a *sink pointer* ( )in the diagrams in Chapter 6.

See the related entries in Chapter 6, "Tag Specifications," for a complete descriptions of these tags.

Physical Organization of HDF Files

The file header, DD blocks, and data elements appear in the following order in an HDF file:

- File header
- First DD block
- Data elements
- If necessary, more DD blocks, more data elements, etc.

These relationships are summarized in Table 1.2.

The only rule governing the distribution of DD blocks and data elements within a file is that the first DD block must follow immediately after the file header. After that, the pointers in the DD headers connect the DD blocks in a linked list and the offsets in the individual DDs connect the DDs to the data elements.

Table 1.2 Summary of the Relationships among Parts of an HDF File

Part	Constituents
HDF file	FH, DD block, data, DD block, data, DD block, data...
FH	0x0e031301 [32-bit HDF magic number]
DD block	DDH, DD, DD, DD, ...
DDH	Number of DDs [16 bits], offset to next DD block [32 bits]
DD	Tag [16 bits], ref [16 bits], offset [32 bits], length [32 bits]
Data	Data element, data element, data element ...

FH = file header, DD = data descriptor, DDH = DD header

1 Some HDF software provides the capability of storing objects as a series of linked blocks or external elements, but this occurs at a higher level. At the lowest level each object with a tag/ref is stored contiguously.

Sample HDF File

We are now ready to examine a sample file. Consider an HDF file that contains two 400-by-600 8-bit raster images as described in Table 1.3.

Table 1.3 Sample Data Objects in an HDF File

Tag	Ref	Data
DFTAG_FID	1	File identifier: user-assigned title for file
DFTAG_FD	1	File descriptor: user-assigned block of text describing overall file contents
DFTAG_LUT	1	Image palette (768 bytes)
DFTAG_ID	1	*x*- and *y*-dimensions of the 2-dimensional arrays that contain the raster images (4 bytes)
DFTAG_RI	1	First 2-dimensional array of raster image pixel data (*x**y* bytes)
DFTAG_RI	2	Second 2-dimensional array of pixel data (also *x**y* bytes)

Assuming that a DD block contains 10 DDs, the physical organization of the file could be described by Figure 1.5.

In this instance, the file contains two raster images. The images have the same dimensions and are to be used with the same palette, so the same data objects for the palette (`DFTAG_IP8`) and dimension record (`DFTAG_ID8`) can be used with both images.

Figure 1.5 Physical Representation of Data Objects

Section	Item	Offset	Contents
Header	FH	0	`0e031301` *(HDF magic number, in hexadecimal)*
DD block	DDH	4	`10 0`
	DD	10	`DFTAG_FID 1 130 4`
	DD	22	`DFTAG_FD 1 134 41`
	DD	34	`DFTAG_LUT 1 175 768`
	DD	46	`DFTAG_ID 1 943 4`
	DD	58	`DFTAG_RI 1 947 240000`
	DD	70	`DFTAG_RI 2 240947 240000`
	DD	82	`DFTAG_NULL` *(Empty)*
	DD	94	`DFTAG_NULL` *(Empty)*
	DD	106	`DFTAG_NULL` *(Empty)*
	DD	118	`DFTAG_NULL` *(Empty)*
Data	Data	130	`sw3`
	Data	134	`solar wind simulation: third try. 8/8/88`
	Data	175	`....` *(Data for the image palette)*
	Data	943	`400 600` *(Image dimensions)*
	Data	947	`....` *(Data for the first raster image)*
	Data	240947	`....` *(Data for the second raster image)*

Chapter 2 Software Overview

Chapter Overview

This chapter describes the HDF software organization and provides guidelines for writing HDF software.

HDF is an amalgam of code and functionality from many sources. For example, the netCDF code came from the Unidata Program Center, and data compression and conversion software has been acquired from a variety of third parties. NCSA staff wrote the code for the basic HDF functionality and perfomed all of the integration work.

This document contains specifications for the NCSA-developed code and functionality. It does not include specifications for code or functionality from non-NCSA sources, though it does sometimes refer to specifications provided by other sources. Only the HDF interface to such code is specified in this document.

HDF Software Layers

There are three basic levels of HDF software:

- The HDF low level interface
- The HDF application interfaces
- HDF applications and utilities

The lowest layer, the *low level interface*, includes general purpose routines that form the basis of all higher-level HDF development. The low level routines directly execute functions such as file I/O, error handling, memory management, and physical storage.

The *application interfaces* support higher level views of data and provide the interfaces for building user-level applications. Routines to handle raster images, palettes, annotations, scientific data sets, Vdatas and netCDF appear at this level.

The *applications and utilities* are implemented at the highest level. NCSA utilities, NCSA applications, and third party applications are all implemented at this level.

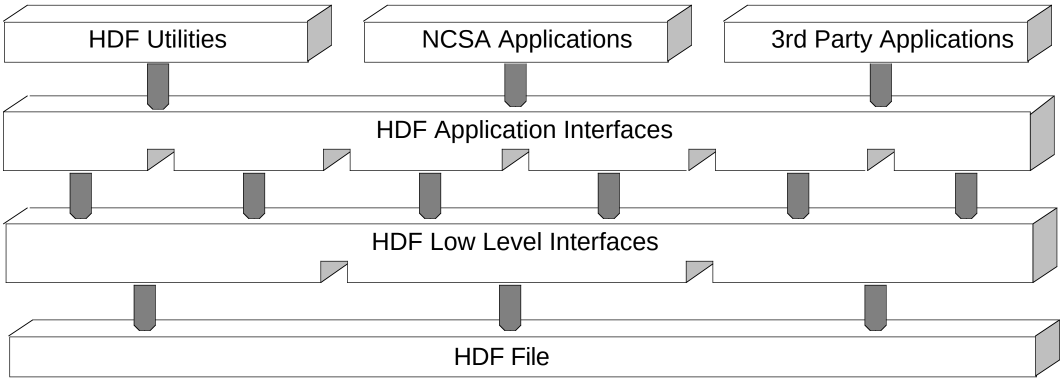
The utilities perform general functions, such as listing the contents of an HDF file, and more specialized functions, such as converting data from one HDF data type to another (e.g., raster images to scientific data sets). In general, the utilities have simple command line interfaces and perform data management tasks.

The applications usually perform data analysis tasks and have polished interactive user interfaces. They include the NCSA Visualization Tool

Suite, commercial software packages that use HDF, and other packages created at NCSA and by various third party projects.

Figure 2.1 illustrates this layered implementation.

Figure 2.1. HDF Software Layers [1]



The general purpose interfaces are described in detail in this document. The application interfaces and command line utilities are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3. Other HDF-based software tools should have their own manuals.

Since the NCSA user community writes programs primarily in C and FORTRAN, all of the HDF application interfaces developed at NCSA are callable from both C and FORTRAN programs. Since the general purpose interface is primarily for program development, not for applications, it provides C-callable routines only.

Software Organization

Versions and Release Numbers

Since HDF is under continual development, new releases are periodically made available. Each new release of the HDF library is identified by a *version number*.

The version number consists of three elements:

`majorv`	Major version number
`minorv`	Minor version number
`rn`	Release number

The version number is presented in the following format:

`majorv.minorvrrn` (e.g., Version 3.2r1)

These elements are interpreted as follows:

Major version number

A new major version number is assigned when there is some fundamental difference between a new version of the library and the

1 This is a simplified illustration of the HDF software layers. Though the basic principles illustrated here continue to apply, the introduction of netCDF and multiple-file HDF data structures renders the implementation considerably more complex.

previous version. When a new major version is released, HDF users and developers are strongly encouraged to obtain the new source code and documentation. There will probably be added functionality in successive major versions of the library and some obsolete code may be deleted. Some user code may have to be modified to use the new library.

Minor version number
: A new minor version number indicates an intermediate release between one major version and the next. Changes will probably be significant. When a new minor version is released, users and developers are strongly encouraged to obtain the new source code and documentation.

Release number
: A new release number is assigned when bug fixes or other small modifications have been made. Using a new release of the same version of the library will not usually require modifying existing user code.

ANSI C and Portability

To ensure that HDF can be easily ported to new platforms, all versions of the HDF source code from Version 3.2 on will be written in ANSI standard C, with special provisions for non-ANSI compilers. For more information about porting HDF and writing portable HDF-based code, refer to Chapter 7, "Making HDF Portable."

Modules and Interfaces

The HDF distribution contains many source files or modules that can be grouped into families. For example, `dfp.c`, `dfpf.c`, and `dfpff.f` all share the root name `dfp` and, therefore, all belong to the `dfp` family. In general, each family of source modules represents one HDF applications interface; the `dfp` family represents the HDF Palette Interface. Exceptions to this rule will be discussed later in this section.

For each interface, there is necessarily one file that contains the C code that provides the basic functionality of that interface. But some interfaces may have one or two additional code modules that provide FORTRAN callability for the interface, so families may have one, two, or three files:

1 file
: Modules of this sort are generally not calling interfaces themselves; they provide useful support functions for actual calling interfaces. Since they are not meant to be called by any routine outside the HDF library, they do not need to be FORTRAN-callable. Example: `hblocks.c` is called only by internal HDF routines and has only the C-callable interface.

2 files
: Although there are currently no two-file families, it is conceivable (and desirable) that some future interface will need only one extra source module to provide FORTRAN compatibility. If this were to happen, there would only be two source modules for the interface. Example: `dfnew.c` and `dfnewf.c` would make up the New Interface.

3 files
: Most current implementations of FORTRAN-callable HDF interfaces require that character string arguments be passed to some of their functions. Due to differences in the way C and

FORTRAN represent strings, passing strings requires that there be a small amount of special purpose FORTRAN code written for each function that takes a string argument.

Therefore, most FORTRAN-callable HDF interfaces consist of three source modules:

- The primary C module
- A FORTRAN-callable C module
- A FORTRAN module

Example: `dfsd.c`, `dfsdf.c`, and `dfsdff.f` make up the Scientific Data Set Interface. `dfsd.c` contains the basic functionality of the interface. `dfsdf.c` provides the major part of FORTRAN callability. And `dfsdff.f` contains the special purpose FORTRAN code that enables passing character string arguments.

Header Files

In addition to the source code modules discussed above, some interfaces also have C header files associated with them that are meant to be included by C applications programmers with the `#include` preprocessor directive. They contain useful constants and data structures for interaction with the interface from C programs. The header files can be identified by the same name as the root name for the rest of the family with the `.h` extension. For example, `dfsd.h` is the header file for the Scientific Data Set Interface.

Of particular importance among the C header files are `hdf.h` and `hdfi.h`:

`hdf.h` Contains all the symbolic constants and public data structures required by HDF. `hdf.h` should be included by any program that uses any of these constants or data structures.

`hdfi.h` Contains specific portability information about each platform on which HDF is supported. `hdfi.h is` automatically included in programs when `hdf.h` is included, so programmers need not explicitly include it.

Refer to Chapter 7, "Making HDF Portable," for more information on `hdfi.h` and other portability issues.

By way of illustration, Table 2.1 lists selected families of source code modules and header files from of HDF Version 3.3.

Table 2.1 Sample HDF Version 3.3 Source Code Modules

General headers	General purpose	Grouping (non-Vset)	Utilities	Annota-tions	General rasters	Scientific data sets	Vsets
hdf.h hdfi.h hproto.h dfivms.h	hfile.c hfilef.c hfileff.f hkit.c hblocks.c hextelt.c herr.c herrf.c hfile.h herr.h	dfgroup.c dfgroup.h	dfutil.c dfutilf.c dfutilff.f dfutil.h	dfan.c dfanf.c dfanff.f dfan.h	dfgr.c dfgr.h dfcomp.c dfimcomp.c dfrig.h	dfsd.c dfsdf.c dfsdff.f dfsd.h	vg.c vgf.c vgff.f vfp.c vgi.h vio.c vconv.c vparse.c vrw.c vsfld.c vg.h vproto.h

The HDF Test Suite

In addition to the source code for the HDF library, versions 3.2 and higher include a test suite. There are two test modules: one for C and one for FORTRAN. Each module tests all of the routines in all of the application interfaces and in the general purpose interface. The exact form of these test modules may vary from one release to the next; consult the release code and online test documentation for details.

Every effort has been made to ensure that the test programs provide a thorough and accurate assessment of the health of the HDF library. Although the test suite will greatly improve the reliability of HDF code, it is almost inevitable that some parts of the code will remain untested. Therefore, no guarantees can be made on the basis of test suite performance.

Sample HDF Programs

Each HDF release includes several sample programs to help users write HDF programs. They illustrate some of the common techniques employed by HDF programmers.

Some HDF Conventions

The HDF specification described in the previous chapter is not sufficient to guarantee its success. It is also important that HDF programmers and users adhere to certain conventions. Some guidelines are implicit in the discussions in other sections of this document. Others are presented in the document *NCSA HDF Calling Interfaces and Utilities* (for Versions 3.2 and earlier) or in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* (for Version 3.3).

Guidelines not covered elsewhere are introduced in this section.

Naming and Assigning Tags

Tags that are to be made available to a general population of HDF users should be assigned and controlled by NCSA. Tags of this type are given numbers in the range 1 to 32,767. If you have an application that fits this criterion, contact NCSA at the address listed in the front matter at the beginning of this manual and specify the tags you would like. For each tag, your specifications should include a suggested name, information about the type and structure of the data that the tag will refer to, and information about how the tag will be used. Your specifications should be similar to those contained in Chapter 6, “Tag Specifications.” NCSA will assign a set of tags for your application and will include your tag descriptions in the HDF documentation.

Tags in the range 32,768 to 64,999 are user-definable. That is, you can assign them for any private application. If you use tags in this range, be aware that they may conflict with other people's private tags.

Using Reference Numbers to Organize Data Objects

Note: Users are discouraged from assigning any meaning to reference numbers beyond that imparted by the HDF library.

The HDF library itself uses reference numbers solely to distinguish among objects with the same tag. While application programmers may find it convenient to impart some meaning to reference numbers, they should be forewarned that the HDF library will be ignorant of any such meaning.

Multiple References

Multiple references to a single data element are quite common in HDF. The general purpose routine `Hdupdd` generates a new reference to data that is already pointed to by another DD. If `Hdupdd` is used several times, there may be several DDs that point to the same data element.

It is important to note that when a multiply-referenced data element is deleted or moved, the various DDs that previously pointed to the data element are *not* automatically deleted or adjusted to point to the data element in its new location. Consequently, each DD to be deleted or moved should be checked for multiple references and handled appropriately.

Chapter 3 General Purpose Interface

Chapter Overview

This chapter provides a detailed description of the routines that make up the HDF general purpose interface.

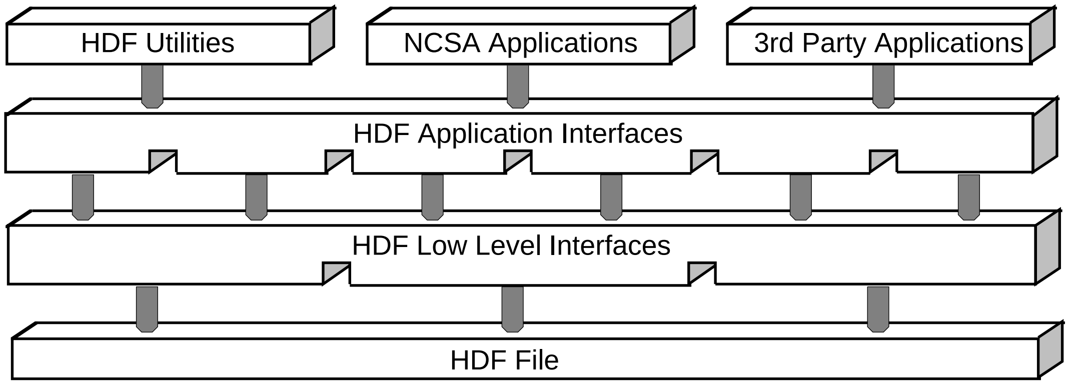
Introduction

HDF supports several interfaces which can be categorized as high level and general purpose interfaces:

- High level interfaces support utilities and applications.
- General purpose interfaces perform basic operations on HDF files.

These levels are illustrated in Figure 3.1, “HDF Software Layers.”

Figure 3.1. HDF Software Layers



This chapter is concerned only with the general purpose routines.

Using these routines, you will be able to build and manipulate HDF objects of any type, including those of your own design. All HDF applications developed at NCSA use them as basic building blocks.

The general purpose routines are all written in C but are typically accessible from FORTRAN.

New General purpose Routines with Version 3.2

The general purpose routines described in this chapter were new with HDF Version 3.2, released in June 1992; they replace the routines provided with earlier versions. The new routines provide better performance and increased functionality and users are strongly advised to use them in new applications. The old routines are supported through emulation, but may be eliminated from the HDF library in a future release.

The new lower layer incorporates the following improvements:
- More consistent data and function types
- More meaningful and extensive error reporting
- Simplification of key lower level functions
- Simplified techniques to facilitate portability
- Support for alternate forms of physical storage, such as linked blocks storage and storage of the data portion of an object in an external file
- A version tag to indicate which version of the HDF library last changed a file
- Support for simultaneous access to multiple files
- Support for simultaneous access to multiple objects within a single file

The previous lower layer was called the *DF layer* because all routines began with the letters `DF` (e.g., `DFopen` and `DFclose`). The new lower layer is called the *H layer* because all routines begin with the letter `H` (e.g., `Hopen`, `Hclose`, and `Hwrite`). The source modules containing these routines begin with the letter `h` (see Table 2.1, "HDF Version 3.2 source code modules"):

`hfile.c`	Basic I/0 routines
`herr.c`	Error-handling routines
`hkit.c`	General purpose routines
`hblocks.c`	Routines to support linked block storage
`hextelt.c`	Routines to support external storage of HDF data elements

Overview of the Interface

This section provides specifications and descriptions of the public functions of the general purpose interface.

Opening and Closing HDF Files

These calls are used to open and close HDF files:

`Hopen`	Provides an access path to an HDF file and reads all of the DD blocks in the file into memory
`Hclose`	Closes the access path to a file

Locating Elements for Access and Getting Information

These routines locate elements or acquire other information about an HDF file or its data objects. Except for `Hendaccess`, they initialize the element that they locate and return an *access ID* that is used in later references to the data element. Calls can include wildcards so that one can search for unknown tags and reference numbers (tag/refs).

`Hstartread`	Locates an existing data element with matching tag/ref and returns an access ID for reading it
`Hnextread`	Continues the search with the same access ID
`Hendaccess`	Disposes of access ID for tag/ref
`Hinquire`	Returns access information about a data element

`Hishdf`	Determines whether a file is an HDF file.
`Hnumber`	Returns the number of occurrences of a specified tag/ref in a file
`Hgetlibversion`	Returns version information for the current HDF library
`Hgetfileversion`	Returns version information for an HDF file

Reading and Writing Entire Data Elements

There are two sets of routines for reading and writing data elements. The routines described here are used to store and retrieve entire data elements.

`Hputelement`	Adds or replaces elements in a file
`Hgetelement`	Reads data elements in a file

A second set of routines, described in the next section, may be used if you wish to access only part of a data element.

Reading and Writing Part of a Data Element

The second set of routines for reading and writing data elements makes it possible to read or write all or part of a data element. One of the access routines `Hstartread` or `Hstartwrite` must be called before these `Hwrite`, `Hread`, or `Hseek`:

`Hstartwrite`
: Sets up writing to the object with the supplied tag/ref. If the object exists, it will be modified; otherwise it will be created.

`Hwrite`
: Writes data to a data element where the last write or Hseek() stopped. If the space reserved is less than the length to write, then only as much as can fit is written.

`Hread`
: Reads a portion of a data element. It starts at the last position left by an `Hread` or `Hseek` call and reads any data that remains in the element up to a specified number of bytes.

`Hseek`
: Sets the access pointer to an offset within a data element. The next time `Hread` or `Hwrite` is called, the access occurs from the new position. The location to seek can be specified as an offset from the current location, from the start of the element, or from the end of the element..

Manipulating Data Descriptors (DDs)

These routines perform operations on DDs without doing anything with the data to which the DDs refer:

`Hdupdd`	Generates new references to data that is already referenced from somewhere else
`Hdeldd`	Deletes a tag/ref from the list of DDs
`Hnewref`	Returns the next available reference number for the HDF file

Creating Special Data Elements

HDF 3.2 introduces two alternate methods of storing HDF objects: *linked blocks* and *external elements*. In previous releases, any data element had to be stored contiguously and all of the objects in an HDF file had to be in the same physical file. The contiguous requirement caused many problems, especially with regard to appending to existing objects. If you wanted to append data to an object, the entire data element had to be deleted and rewritten to the end of the file.

Linked blocks allow elements in a single HDF file to be non-contiguous.

External elements allow a single HDF object to be stored in an external file.

It is not currently possible to store a single object (such as a very large data set) in multiple files. Nor can multiple objects be stored in one external file.

Once they are created with the following routines, these special data elements can be accessed with the routines used for normal data elements:

`HLcreate`	Creates a new linked block special data element
`HXcreate`	Creates a new external file special data element

These routines have two modes of operation. Calling `HLcreate` with a tag/ref that does not exist in a file will create a new element with the given tag/ref which will be stored as linked blocks. On the other hand, if the tag/ref already exists in the file, the referenced object will be promoted to linked block status. All data which had been stored in the object before the promotion will be retained. `HXcreate` behaves similarly.

Development Routines

The HDF library provides the following developer-level routines that simplify the task of writing HDF applications. Most of these routines mirror basic C library functions which are, unfortunately, not always completely portable in their library form:

`HDgettagname`	Returns a pointer to a text string describing a given tag
`HDgetspace`	Allocates space
`HDfreespace`	Frees space
`HDstrncpy`	Copies a string from one location to another up to a given number of characters

Error Reporting

The HDF library incorporates the notion of an *error stack.* This allows much of the context to be known when trying to decipher an error message.

Error reporting is handled by the following routines:

`HEprint`	Prints out all of the errors on the error stack to a specified file
`HEclear`	Clears the error stack
`HERROR`	Reports an error Pushes the following information onto the error stack:

Error type
source file name
Line number and the name of the function reporting the error

`HEreport` Adds a text string to the description of the most recently reported error (only one text string per error)

Standard C does not enable the code inside a function to know the name of the function. Therefore, to use the macro `HERROR` to report errors, there must exist a variable `FUNC` which points to a string containing the name of the reporting function.

Other

The `Hsync` routine has been defined and implemented to synchronize a file with its image in memory. Currently it is not very useful because the HDF software includes no buffering mechanism and the two images are always identical. `Hsync` will become useful when buffering is implemented:

`Hsync` Synchronizes the stored version of an HDF file with the image in memory

Function Specifications

The terms IN: and OUT: are used as follows in this discussion:

IN: Value as input parameter
OUT: Value as output parameter

Opening and Closing Files

Hopen

`int32 Hopen(char *path, int access, int16 ndds)`

path	IN:	Name of file to be opened
access	IN:	`DFACC_READ`, `DFACC_RDWR`, `DFACC_CREATE`, `DFACC_ALL`, or `DFACC_WRITE`
ndds	IN:	Number of DDs in a block if this file needs to be created

Purpose
Provides an access path to an HDF file and reads all of the DD blocks in the file into primary memory.

Return value
Returns file ID if successful and FAIL (`-1`) otherwise.

Description
Opens an HDF file.

The following events occur on successful exit:

- `File_rec` members are filled in. (`File_rec` is an internal HDF structure containing information about the opened file.)
- The requested file is opened with the relevant permission.
- Information about DDs is set up in memory.
- The file headers and initial information are set up for new files.

Access privilege codes
HDF provides several constants for use as access privilege codes as listed below. Note that these constants are not bit-flags and should not be ORed together to combine access modes. Doing so may cause odd behavior and, in some cases, loss of data:

Recommended:

`DFACC_READ`	Open for read only. If file does not exist, error.
`DFACC_RDWR`	Open for read/write. If file does not exist, create it.
`DFACC_CREATE`	Force creation. If file exists, delete it, then open a new file for read/write (in the spirit of the UNIX System command `clobber`).

Others:

`DFACC_ALL`	Same as `DFACC_RDWR` (obsolete but still supported).
`DFACC_WRITE`	Same as `DFACC_RDWR` (obsolete but still supported).

`Hclose`

`intn Hclose(int32 id)`

`id`	IN:	The file ID of the file to be closed

Purpose Closes the access path to the file.

Return value Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description *`id`* is first validated. If valid, the function closes the access path to the file.

If there are still access elements attached to the file, the error `DFE_OPENAID` is pushed onto the error stack and the file is not closed. This is a fairly common error when developing new interfaces. See the discussion of `Hendaccess` below for debugging hints.

Locating Elements for Access and Getting Information

Hstartread

`int32 Hstartread(int32 file_id, uint16 tag, uint16 ref)`

file_id	IN:	ID of file to attach access element to
tag	IN:	Tag to search for
ref	IN:	Reference number to search for

Purpose: Locates an existing data element with matching tag/ref and returns an access ID for reading it.

Return value: Returns access element ID if successful and FAIL (`-1`) otherwise.

Description: Searches the DDs for a particular tag/ref combination. If the search is successful, an access element is created, attached to the file, and positioned at the start of that data element; otherwise an error is returned. Searching on wildcards begins from the beginning of the DD list. Wildcards can be used for the tag or reference number (`DFTAG_WILDCARD` and `DFREF_WILDCARD`) and they match any values.

Hnextread

`intn Hnextread(int32 access_id, uint16 tag, uint16 ref, int origin)`

access_id	IN:	ID of a READ access element
tag	IN:	Tag to search for
ref	IN:	Reference number to search for
origin	IN:	Position at which to start searching

Purpose: Locates and positions a read access ID on next occurrence of tag/ref.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Searches for the next DD that fits the tag/ref. Wildcards apply. If *origin* is `DF_START`, searches from start of DD list; if *origin* is `DF_CURRENT`, searches from current position. Searching from the end of the file via `DF_END` is not yet implemented.

If the search is successful, then the access element is positioned at the start of that tag/ref; otherwise, the access ID is not modified.

Hstartwrite

`int32 Hstartwrite(int32 file_id, uint16 tag, uint16 ref, int32 length)`

file_id	IN:	ID of file to write to
tag	IN:	Tag to write to
ref	IN:	Reference number to write to
length	IN:	Length of the data element

Purpose: Creates or replaces data element with matching tag/ref.

Return value: Returns access element ID if successful and FAIL (`-1`) otherwise.

Description: Sets up an access element to write a data element. The DD list of the file is searched first; if the tag/ref is found, the data element can be modified. If an object with the corresponding tag/ref is not found, a new one is created.

Hendaccess

`int32 Hendaccess(int access_id)`

access_id	IN:	ID of access element to dispose of

Purpose: Disposes of access element for tag/ref.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Disposes of an access element. Only a finite number of access elements can be active at a given time, so it is important to call `Hendaccess` whenever you are done using an element.

When developing new interfaces, a common mistake is to fail to call `Hendaccess` for all of the elements accessed. When this happens, `Hclose` will return FAIL and the dump of the error stack (see `HEprint` below) will tell how many access elements are still active.

This can be a difficult problem to debug, as the low levels of the HDF library have no idea who or what opened an access element and forgot to release it. A tedious but effective means of debugging this problem is to annotate with comments the locations where the attached count of a file record is changed. This occurs in the files `hfile.c`, `hblocks.c`, and `hextelt.c`.

Hinquire

```
intn Hinquire(int32 access_id, int32 *pfile_id, uint16 *ptag,
      uint16 *pref, int32 *plength, int32 *poffset, int32 *pposn,
      int *paccess, int16 *pspecial)
```

access_id	IN:	Access element ID
pfile_id	OUT:	File ID
ptag	OUT:	Tag of the element pointed to
pref	OUT:	Reference number of the element pointed to
plength	OUT:	Length of the element pointed to
poffset	OUT:	Offset of element in the file
pposn	OUT:	Position pointed to within the data element
paccess	OUT:	Access type of this access element
pspecial	OUT:	Special code

Purpose: Returns access information for a data element.

Return value: Returns SUCCEED (0) if the access element points to some data element and FAIL (-1) otherwise.

Description: Inquires for the statistics of the data element pointed to by the access element. If a piece of information is not needed, a NULL can be sent in for that value. Convenience macros for calls to `Hinquire` (`HQueryposition`, `HQuerylength`, etc.) are defined in `hdf.h`.

Hishdf

```
int32 Hishdf(char *path)
```

path	IN:	Name of file

Purpose: Determines whether a file is an HDF file.

Return value: Returns TRUE (non-zero) if file is an HDF file and FALSE (`0`) otherwise.

Description: The decision as to whether a file is an HDF file is based solely on the magic number stored in the first four bytes of an HDF file. `Hishdf` may sometimes identify a file as an HDF file that `Hopen` is unable to open (e.g., an HDF file with a corrupted DD list).

Note: `Hishdf` only determines whether a file is an HDF file. It does not verify that the file is readable.

Hnumber

```
int Hnumber(int32 file_id, uint16 tag)
```

file_id	IN:	File ID
tag	IN:	Tag to be counted

Purpose: Counts the number of occurrences of a tag in a file.

Return value: The number of occurrences of a tag in a file.

Hgetlibversion

```
Hgetlibversion(uint32 *majorv, uint32 *minorv, uint32 *release,
      char string[])
```

majorv	OUT:	Major version number
minorv	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose: Gets version information for current HDF library.

Return value: Returns SUCCEED (`0`).

Description: Returns the version of the HDF library. The version information is compiled into the HDF library, so it is not necessary to have any open files for this function to execute.

Hgetfileversion

```
Hgetfileversion(uint32 file_id, uint32 *majorv, uint32 *minorv,
      uint32 *release, char *string)
```

file_id	IN:	File ID
majorv	OUT:	Major version number
minorv	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose: Gets version information for an HDF file.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Returns the HDF version information stored in the given file.

Reading and Writing Entire Data Elements

Hputelement

`int Hputelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data, int32 length)`

file_id	IN:	File ID
tag	IN:	Tag of data element to put
ref	IN:	Reference number of data element to put
data	IN:	Pointer to buffer
length	IN:	Length of data

Purpose: Adds or replaces an element in a file.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Writes a new data element or replaces an existing data element in a HDF file. Uses `Hwrite` and its associated routines.

Hgetelement

`int Hgetelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data)`

file_id	IN:	ID of the file to read from
tag	IN:	Tag of data element to read
ref	IN:	Reference number of data element to read
data	OUT:	Buffer to read into

Purpose: Obtains the data referred to by the passed tag/ref.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Reads a data element from an HDF file and puts it into the buffer pointed to by *data*. The space allocated for the buffer is assumed to be large enough.

Note: `Hgetelement` assumes that the buffer is large enough to hold the data being read. It is the user's responsibility to prevent data loss by ensuring that this is the case.

Reading and Writing Part of a Data Element

`Hread`

`int32 Hread(int32 access_id, int32 length, uint8 *data)`

access_id	IN:	Read access element ID
length	IN:	Length of segment to read in
data	OUT:	Pointer to data array to read to

Purpose: Reads a portion of a data element.

Return value: Returns length of segment actually read if successful and FAIL (-1) otherwise.

Description: Reads in the next segment in the data element pointed to by the access element. `Hread` starts at the last position left by an `Hread` or `Hseek` call and reads any data that remains in the element up to *length* bytes. If the data element is too short (less than *length* bytes long), `Hread` reads to the end of the data element.

`Hwrite`

`int32 Hwrite(int32 access_id, int32 length, uint8 *data)`

access_id	IN:	Write access element ID
length	IN:	Length of segment to write
data	IN:	Pointer to data to write

Purpose: Writes next data segment to data element.

Return value: Returns length of segment successfully written and FAIL (-1) otherwise.

Description: Writes the data to the data element where the last `Hwrite` or `Hseek` stopped.

`Hwrite` starts at the last position left by an `Hwrite` or `Hseek` call, writes up to a specified number of bytes, and leaves the write pointer at the end of the data written. If the space reserved is less than the length to write, then only as much as can fit is written.

It is the user's responsibility to ensure that no two access elements are writing to the same data element. Note that a user can interlace writes to multiple data elements in the same file.

Hseek

```
intn Hseek(int32 access_id, int32 offset, int origin)
```

access_id	IN:	Access element ID
offset	IN:	Offset to seek to
origin	IN:	Position to seek from: `DF_START (0)` *offset* from beginning of data element `DF_CURRENT (1)` *offset* from current position `DF_END (2)` *offset* from end of data element

Purpose Sets the access pointer to an offset within a data element. The next time `Hread` or `Hwrite` is called, the read or write occurs from the new position.

Return value Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description Sets the position of an access element in a data element so that the next `Hread` or `Hwrite` will start from that position. *origin* determines the position from which *offset* should be counted.

This routine fails if the access element is not associated with a data element or if the position sought is outside of the data element.

Seeking from the end of a data element is not currently supported.

Manipulating Data Descriptors

`Hdupdd`

int Hdupdd(int32 *file_id*, uint16 *tag*, uint16 *ref*, uint16 *old_tag*,
uint16 *old_ref*)

file_id	IN:	File ID
tag	IN:	Tag of new data descriptor
ref	IN:	Reference number of new data descriptor
old_tag	IN:	Tag of data descriptor to duplicate
old_ref	IN:	Reference number of data descriptor to duplicate

Purpose: Generates new references to data that is already referenced from somewhere else.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Duplicates a data descriptor so that the new tag/ref points to the same data element pointed to by the old tag/ref.

`Hdeldd`

int Hdeldd(int32 *file_id*, uint16 *tag*, uint16 *ref*)

file_id	IN:	File ID
tag	IN:	Tag of data descriptor to delete
ref	IN:	Reference number of data descriptor to delete

Purpose: Deletes a tag/ref from the list of DDs.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Deletes the data descriptor of tag/ref from the DD list of the file. This routine is unsafe and may leave a file in a condition that is not usable by some routines. Use with care.

`Hnewref`

`uint16 Hnewref(int32` *`file_id`*`)`

`file_id`	IN:	File ID

Purpose	Returns the next available reference number.
Return value	Returns the reference number if successful and `0` otherwise.
Description	Returns a reference number that can be used with any tag to produce a unique tag/ref. Successive calls to `Hnewref` will generate a strictly increasing sequence until the highest possible reference number has been returned; then `Hnewref` will return unused reference numbers starting from `1`.

Creating Special Data Elements

`HLcreate`

```
int32 HLcreate(int32 file_id, uint16 tag, uint16 ref,
      int32 block_length, int32 number_blocks)
```

file_id	IN:	File ID
tag	IN:	Tag of new data element (or object)
ref	IN:	Reference number of new data element (or object)
block_length	IN:	Length of blocks to be used
number_blocks	IN:	Number of blocks to use per linked block record

Purpose: Creates a new linked block special data element.

Return value: Returns access ID for special data element if successful and FAIL (-1) otherwise.

Description: Appending to existing HDF elements was a problem prior to HDF Version 3.2 because HDF objects had to be stored contiguously. When appending, the HDF library forced the user to delete the existing element and rewrite it at the end of the file. HDF Version 3.2 introduced the concept of linked blocks, which allow unlimited appending to existing elements without copying over existing data.

This routine can be used to create an object with the given tag/ref as a linked block element or to promote an existing element to be stored in linked blocks.

Initially, a table is set up to accommodate *`number_blocks`* linked blocks for the specified data object. Each block has *`block_length`* bytes. If an existing object is being promoted, *`block_length`* does not have to be the same size as the original element.

`HLcreate` returns an active access ID with write permission to the linked block element.

HXcreate

```
int32 HXcreate(int32 file_id, uint16 tag, uint16 ref,
      char *extern_file_name)
```

file_id	IN:	file record ID
tag	IN:	Tag of the special data element to create or promote
ref	IN:	Reference number of the special data element to create/promote
extern_file_name	IN:	name of the external file to use for the data element

Purpose	Creates a new external file special data element.
Return value	Returns access ID for special data element if successful and FAIL (`-1`) otherwise.
Description	Creates a new element in an external file or promotes an existing element to be stored in an external file. If an existing element is to be promoted, it is deleted (using Hdeldd) from the original file and copied into the new external file.

Distributing a single object over multiple external files is not currently supported. In addition, one cannot place multiple objects in the same external file.

This routine returns an active access ID with write permission to the external element.

Development Routines

HDgettagname

`char *HDgettagname(uint16 tag)`

tag	IN:	Tag to look up

Purpose: Gets a meaningful description of a tag.

Return value: Returns a pointer to a string describing this tag or NULL if the tag is unknown.

Description: To reduce the amount of duplicated code, this routine can be used to map a tag to a character string containing the name of the tag.

The string returned by this routine is guaranteed to be 30 characters or less.

HDgetspace

`void *HDgetspace(uint32 qty)`

qty	IN:	Number of bytes to allocate

Purpose: Allocates space.

Return value: If successful, returns a pointer to space that was allocated; otherwise returns NULL .

Description: Uses an appropriate allocation routine on the local machine to get space.

HDfreespace

`void *HDfreespace(void *ptr)`

ptr	IN:	Pointer to previously-allocated space that is to be freed

Purpose: Frees space.

Return value: Returns NULL.

Description: Uses an appropriate routine on the local machine to free space. This routine is platform dependent.

HDstrncpy

```
char *HDstrncpy(register char *dest, register char *source,
      int32 length)
```

dest	OUT:	Pointer to area to copy string to
source	IN:	Pointer to area to copy string from
length	IN:	Maximum number of bytes to copy

Purpose Copies a string with maximum length `length`.

Return value Returns address of `dest`.

Description Creates a string in `dest` that is at most `length` characters long. The number of characters must *include* the NULL terminator for historical reasons. Hence, if you are working with the string `Foo,` you must call this copy function with the value `4` (three characters plus the NULL terminator) in `length`.

Error Reporting

HEprint

`void HEprint(FILE *stream, int32 level)`

stream	IN:	Stream to print error messages on
level	IN:	Level of the error stack to print

Purpose: Prints information on the error stack.

Return value: Has no return value.

Description: Prints information on reported errors. If *level* is zero, all of the errors currently on the error stack are printed. Output from this function is sent to the file pointed to by *stream*.

The following information printed:
- An ASCII description of the error
- The reporting routine
- The reporting routine’s source file name
- The line at which the error was reported

If the programmer has supplied extra information by means of `HEreport`, this information is printed as well.

HEclear

`void HEclear(void)`

Purpose: Clears all information on reported errors off of the error stack.

Return value: Has no return value.

Description: Clears all of the information off of the error stack.

HERROR

`void HERROR(int16 number)`

number	IN:	Error number

Purpose: Reports an error.

Return value: Has no return value.

Description: Reports an error. Any function calling `HERROR` must have a variable `FUNC` which points to a string containing the name of the function.

`HERROR` is implemented as a macro.

`HEreport`

`void HEreport(char *`*`format`*`, ....)`

`format`	IN:	`printf`-style format and arguments

Purpose — Provides extra information to the error reporting routines.

Return value — Has no return value.

Description — Provides further annotation to an error report. Only one such annotation is remembered for each error report. The arguments to this routine follow the style of `printf`.

Consider the following example from `hfile.c`:

```
char *FUNC = "Hclose";
....
if (file_rec->attach > 0) {
    file_rec->refcount++;
    HERROR(DFE_OPENAID);
    HEreport("There are still %d active aids attached", file_rec->attach);
    return FAIL;
}
```

Other

`Hsync`

`int Hsync(int32 *file_id*)`

file_id	IN:	ID of the file to synchronize

Purpose
: Synchronizes on-disk HDF file with image in memory.

Return value
: Returns SUCCEED.

Description
: `Hsync` is not included in the current HDF library release because the on-disk representation of an HDF file is always the same as its in-memory representation. `Hsync` will be provided when future releases implement buffering schemes.

Chapter 4 Sets and Groups

Chapter Overview

This chapter discusses the roles of the following sets and groups in organizing data stored in an HDF file:

- Raster image sets (RIS)
  - Raster image groups (RIG)
- Scientific data sets (SDS)
  - Scientific data groups (SDG)
  - Numeric data groups (NDG)
  - SDG-like NDGs
- Vsets
  - Vgroups
- Raster-8 sets (obsolete)

This chapter introduces several tags used in support of sets and groups. All of these tags are fully described in Chapter 6, "Tag Specifications," and are listed in the table in Appendix A, "NCSA HDF Tags."

Data Sets

HDF files frequently contain several closely related data objects. Taken together, these objects form a *data set* which serves a particular user requirement. For example, five or six data objects might be used to describe a raster image; eight or more data objects might be used to describe the results of a scientific experiment.

The HDF mechanism for specifying and controlling data sets is the *group*. The data element of a group consists of a single record listing the tag/refs for all the objects contained in the data set. For example, the raster image groups described in the following sections each contain three tag/refs that point to three data objects that, taken as a set, fully describe an 8-bit raster image.

Types of Sets

The current HDF implementation supports three kinds of sets:

Raster image set
: A set containing a raster image and descriptive information such as the image dimensions and an optional color lookup table

Scientific data set
: A set containing a multidimensional array and information describing the data in the array

Vset
A general grouping structure containing any kinds of HDF objects that a user wishes to include

Each HDF set is defined with a minimum collection of data objects that will make sense when the set is used. For example, every raster image set must contain at least the following data objects:

Raster image group
The list of the members of the set

Image dimension record
The width, height, and pixel size of the raster image

Raster image data
The pixel values that make up the image

In addition to the required objects, a set may include optional data objects. An 8-bit raster image set, for instance, often contains a palette, or color lookup table, which defines the red, green, and blue values associated with each pixel in the raster image.

Calling Interfaces for Sets

NCSA provides calling interfaces for all the HDF sets that it supports. These interfaces provide routines for reading and writing the data associated with each set. The libraries currently supported by NCSA are callable from either C or FORTRAN programs.

In addition to the libraries, a growing number of command-line utilities are available to manipulate sets. For example, a utility called `r8tohdf` converts one or more raw raster images to HDF 8-bit raster image set format.

The calling interfaces are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Groups

As discussed above, HDF data objects are frequently associated as sets. But without some explicit identifying mechanism, there is often no way to tie them together. To address this problem, HDF provides a grouping mechanism called a *group*. A group is a data object that explicitly identifies all of the data objects in a set.

Since a group is just another type of data object, its structure is like that of any other data object; it includes a DD and a data element. But instead of containing the pixel values for a raster image or the dimensions of an array, a group data element contains a list of tag/refs for the data objects that make up the corresponding set.

A *group tag* can be defined for any set. For instance, the *raster image group* tag (RIG, `DFTAG_RIG`) is used to identify members of raster image sets; the RIG data element lists the tag/refs for a particular raster image set.

An Example

Suppose that the two images shown in Figure 1.5, “Physical Representation of Data Objects,” are organized into two sets with group tags. Since they are raster images, they may be stored as RIGs. Figure 4.1 illustrates the use of RIGs with these images.

Figure 4.1 Physical Organization of Sample RIG Groupings

Offset	Item	Contents
0	FH	`0e031301` *(HDF magic number)*
4	DDH	`10 0L`
10	DD	`DFTAG_FID 1 130 4`
22	DD	`DFTAG_FD 1 134 41`
34	DD	`DFTAG_LUT 1 175 768`
46	DD	`DFTAG_ID 1 943 4`
58	DD	`DFTAG_RI 1 947 240000`
70	DD	`DFTAG_ID 2 240947 4`
82	DD	`DFTAG_RI 2 240951 240000`
94	DD	`DFTAG_RIG 1 480951 12`
106	DD	`DFTAG_RIG 2 480963 12`
118	DD	`DFTAG_NULL` *(Empty)*
130	Data	`sw3`
134	Data	`solar wind simulation: third try. 8/8/88`
175	Data	`...` *(Data for image palette)*
943	Data	`400, 600 ...` *(Data for 1st image dimension record)*
947	Data	`...` *(Data for 1st raster image)*
240947	Data	`400, 600 ...` *(Data for 2nd image dimension record)*
240951	Data	`...` *(Data for 2nd raster image)*
480951	Data	`DFTAG_IP8/1, DFTAG_ID/1, DFTAG_RI/1` *(Tag/refs for 1st RIG)*
480963	Data	`DFTAG_IP8/1, DFTAG_ID/2, DFTAG_RI/2` *(Tag/refs for 2nd RIG)*

The file depicted in Figure 4.1 contains the same raster image information as the file in Figure 1.5, but the information is organized into two sets. Note that there is only one palette (`DFTAG_IP8/1`) and that it is included in both groups.

General Features of Groups

Figure 4.1 also illustrates a number of important general features of groups:

- The contents of a group must be consistent with one another. Since the palette (`DFTAG_IP8`) is designed for use with 8-bit images, the image must be an 8-bit image.
- An application program can easily process all of the images in the file by accessing the groups in the file. The non-RIG information in the example can be used or ignored, depending on the needs and capabilities of the application program.
- There is usually more than one way to group sets. For example, an extra copy of the image palette (`DFTAG_IP8`) could have been stored

in the file so that each grouping would have its own image palette. That is not necessary in this instance because the same palette is to be used with both images. On the other hand, there are two image dimension records in this example, even though one would suffice.

- Group status does not alter the fundamental role of an HDF object; it is still accessible as an individual data object despite the fact that it also belongs to a larger set.

- A group provides an index of the members of a set. There is nothing to prevent the imposition of other groupings (indexes) that provide a different view of the same collection of data objects. In fact, HDF is designed to encourage the addition of alternate views.

The following sections formally describe raster image sets (RIS), scientific data sets (SDS), Vsets, and several related groups. The last section of this chapter discusses an obsolete structure known as the raster-8 set.

Raster Image Sets (RIS)

The raster image set (RIS) provides a framework for storing images and any number of optional image descriptors. An RIS always contains a description of the image data layout and the image data. It may also contain color look-up tables, aspect ratio information, color correction information, associated matte or other overlay information, and any other data related to the display of the image.

Raster Image Groups (RIG)

Tying everything together is the raster image group (RIG, see Figure 4.1 and the related discussion for an example). An RIG contains a list of tag/refs that point in turn to the data objects that make up and describe the image.

The number of entries in an RIG is variable and most of the descriptive information is optional. Complex applications may include references to image-modifying data, such as the color table and aspect ratio, along with the reference to the image data itself. Simple applications may use simple application-level calls and ignore specialized video production or film color correction parameters.

NCSA currently supports two RIG calling interfaces: *RIS8* and *RIS24*. These interfaces are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

RIS Tags

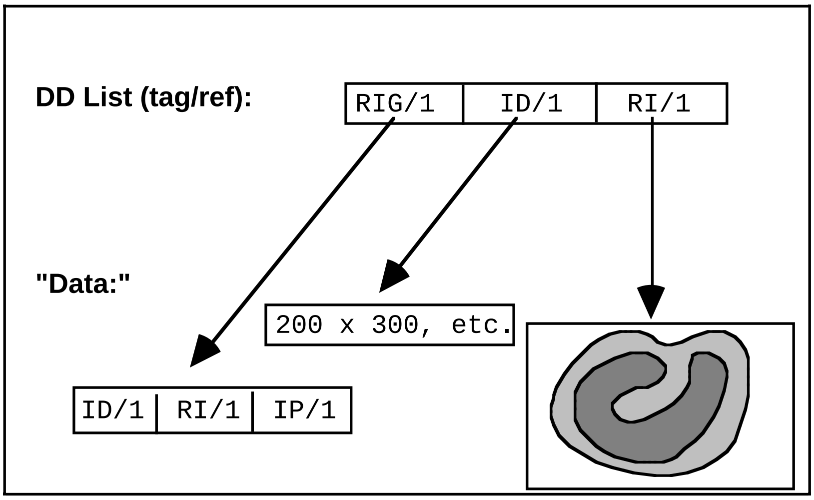
RIS implementations must fully support all of the tags presented in Table 4.1.

Table 4.1 RIS Tags

Tag	Contents of Data Element
`DFTAG_RIG`	Raster image group
`DFTAG_ID`	Image dimension record
`DFTAG_RI`	Raster image data

With these tags, images can be stored and read from HDF files at any bit depth, with several different component ordering schemes. As illustrated in Figure 4.1, the RIG tag points to the collection of tag/refs that fully describe the RIS. The data element attached to the tag `DFTAG_ID` specifies the dimensions of the image, the number type of the elements that make up its pixels, the number of elements per pixel, the interlace scheme used, and the compression scheme used, if any. The data element attached to the tag `DFTAG_RI` contains the actual raster image data.

Figure 4.1 RIS Tags



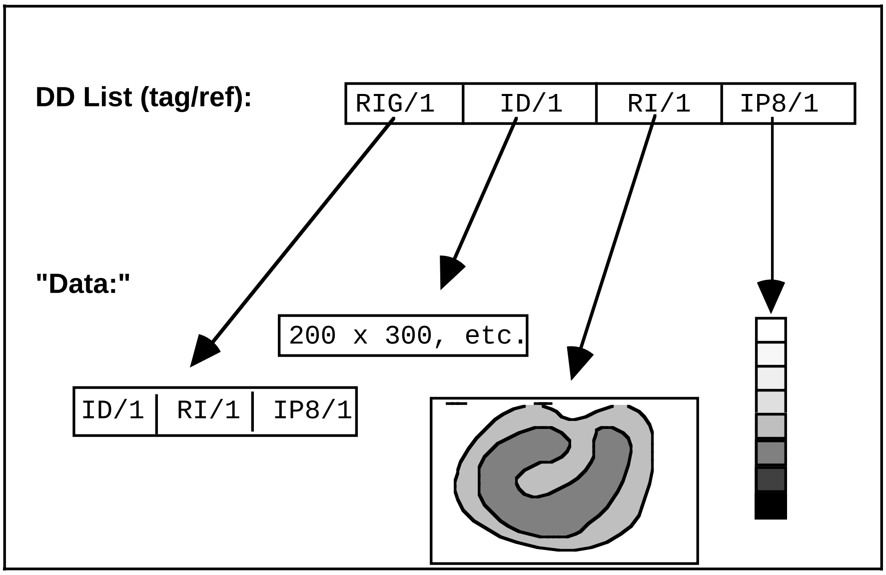
The tags listed in Table 4.2 identify optional RIS information such as color properties and aspect ratio. Note that the RI interface supports only `DFTAG_LUT` at this time; the other tags in Table 4.2 are defined but the interfaces have not been implemented.

Table 4.2 Optional RIS Tags

Tag	Contents of Data Element
`DFTAG_XYP`	XY position of image
`DFTAG_LD`	Look-up table dimension record
`DFTAG_LUT`	Color look-up table for non true-color images
`DFTAG_MD`	Matte channel dimension record
`DFTAG_MA`	Matte channel data
`DFTAG_CCN`	Color correction factors
`DFTAG_CFM`	Color format designation
`DFTAG_AR`	Aspect ratio
`DFTAG_MTO`	Machine-type override

Figure 4.2 illustrates the structure of an RIS that contains an image palette (`DFTAG_IP8`).

Figure 4.2 RIS Tags for Sets Containing a Palette



Raster Image Compression

HDF currently supports two raster image compression tags:

`DFTAG_RLE`	Run-length encoding
`DFTAG_IMCOMP`	Aerial averaging
`DFTAG_JPEG`	JPEG compression

RIG support does not require support for all compression tags. Be sure to provide a suitable error message to the user when an unknown compression tag is encountered.

Since new forms of data compression can be added to HDF raster images, incompatibilities can arise between old libraries and files created by newer libraries. For example, HDF Version 3.3 includes JPEG compression for images. A JPEG-compressed raster image in a file created by an HDF Version 3.3 library cannot be read by an HDF Version 3.2 library.

Scientific Data Sets

The scientific data set (SDS) provides a framework for storing multidimensional arrays of data with descriptive information that enhances the data. Current specifications support the following types of numbers in SDS arrays.

- 8-bit, 16-bit, and 32-bit signed and unsigned integers
- 32-bit and 64-bit floating point numbers

Data in an SDS can be stored either as two's complement big endian integers, as IEEE Standard floating point numbers, or in *native mode*, the format used by the machine from which they were written.

The user interface for storing and retrieving SDSs is fully described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Backward and forward compatibility

One of NCSA's concerns in HDF development is always to maximize backward and forward compatibility; as much as possible, any application written to use HDF should be able to read data files written with an older or a newer version of the libraries. To maximize this compatibility, NCSA had to consider the following factors in upgrading the SDS capabilities:

- Support for future variations (e.g., new number types, data compression, and new physical arrangements for SDS storage)

- Older versions of the library should be able to read new data files if the data itself can be interpreted by the older version. To do so, the older version must be able to determine whether the data in a given data object will be comprehensible to it. For example, if a newly created file contains 32-bit IEEE floating point or Cray floating point data objects, older versions of the library should be able determine that fact then read and interpret the data.

- New libraries must be able to read and interpret files created by older versions.

Unfortunately, such compatibility concerns yield an SDS structure somewhat more complex than would otherwise be the case. Two examples illustrate the problem:

- HDF 3.2 development had to accommodate the fact that HDF Version 3.1 and previous versions only supported 32-bit IEEE floating-point numbers and Cray floating point numbers in SDSs. SDSs in HDF versions since Version 3.2 support 8-bit, 16-bit, and 32-bit signed and unsigned integers, 32-bit and 64-bit floating-point numbers, and the local machine format (*native mode*) for all supported architectures.

- HDF 3.3 includes support for the netCDF data model, which involved the creation of an entire new structure for supporting netCDF objects, based on Vgroups and Vdatas. At the same time, a goal of HDF 3.3 was to harmonize the SDS and the netCDF data model, which was best accomplished by storing SDS objects in the same way that netCDF objects are stored. In order to maintain backward compatibility, two structures had to be created for every SDS or netCDF object: one that could be recognized by older HDF libraries, and the new structure.

In the following sections we describe how the first problem was solved. A later issue of this manual will describe how the second problem was addressed.

Internal Structures

The SDS capability was substantially enhanced for HDF Version 3.2. Previous versions employed a structure known as a *scientific data group* (SDG); Version 3.2 and subsequent versions use the *numeric data group* (NDG). To accommodate the enhanced structure and to remain compatible with previous releases, the current HDF library supports the following scientific and numerical data groups:

SDGs — Created by old libraries and containing 32-bit IEEE and Cray floating-point data.

NDGs — Created by the newer libraries (Version 3.2 and later) and containing any acceptable floating-point or non-floating-point data. This data group will not be recognized by old libraries.

SDG-like NDGs — Created by the new library and containing IEEE 32-bit floating-point data only. The old libraries will recognize and interpret these numerical data groups correctly.

The NDG structure supports 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating-point numbers. It also supports *native mode,* data sets written to HDF files in the local machine format.

The following sections describe the SDG, NDG, and SDG-like NDG structures.

SDG Structures

SDGs must contain at least the data objects listed in Table 4.3.

Table 4.3 Required SDG Tags

Tag	Contents of Data Element
`DFTAG_SDG`	Scientific data group.
`DFTAG_SDD`	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number type of the array data and of each dimension. All SDG number types are 32-bit IEEE floating-point.
`DFTAG_SD`	Scientific data.

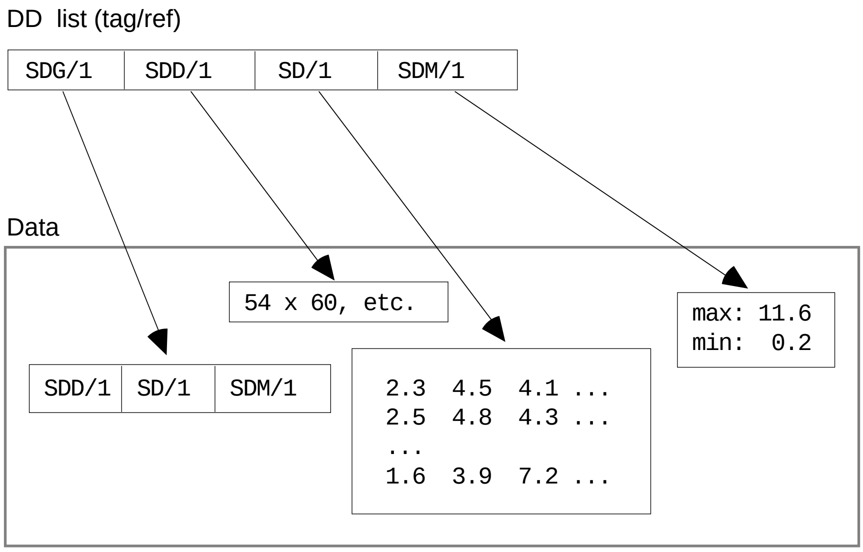
In addition to the required data objects listed above, SDGs may contain any of the objects listed in Table 4.4. Note that the optional data objects are the same for SDGs, NDGs, and SDG-like NDGs; the only differences are the number types that may be used.

Table 4.4 **Optional SDG, NDG, and SDG-like NDG Tags**

Tag	Contents of Data Element
DFTAG_SDS	Scales of the different dimensions. To be used when interpreting or displaying the data (32-bit floating point numbers only for SDGs and SDG-like NDGs).
DFTAG_SDL	Labels for all dimensions and for the data. Each of the dimension labels can be interpreted as an independent variable; the data label is the dependent variable.
DFTAG_SDU	Units for all dimensions and for the data.
DFTAG_SDF	Format specifications to be used when displaying values of the data.
DFTAG_SDM	Maximum and minimum values of the data. (32-bit floating point numbers only for SDGs and SDG-like NDGs.)
DFTAG_SDC	Coordinate system to be used when interpreting or displaying the data.

As illustrated in Figure 4.3, the SDG tag points to the collection of tag/refs that define the SDG.

Figure 4.3 **SDG Structure**



NDG Structures

NDGs must contain at least the data objects listed in Table 4.5

Table 4.5 **Required NDG Tags**

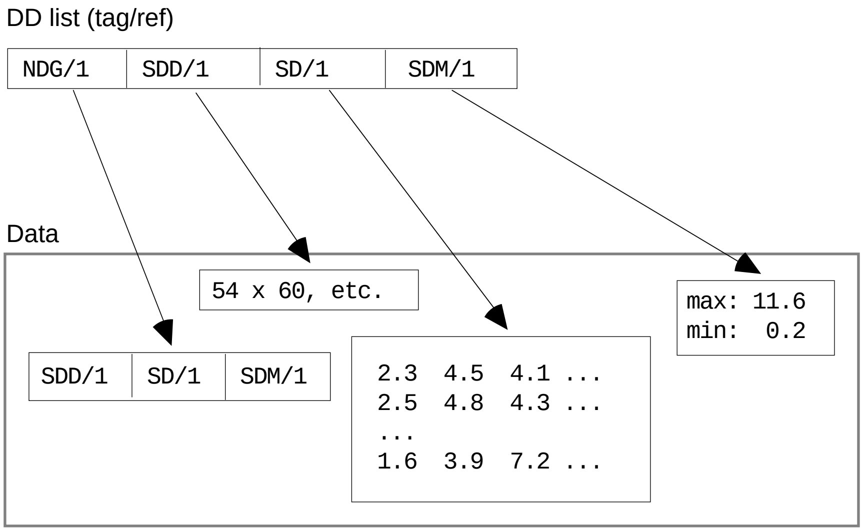
Tag	Contents of Data Element
DFTAG_NDG	Numerical data group.
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number types of the data and of each dimension.

	In HDF 3.2 , the number types of dimension scales must be the same as that of the array-stored data. Later implementations allow dimension scales to be typed separately.
DFTAG_SD	Scientific data.
DFTAG_NT	Number type of the data set. Default is the most recent DFSDsetNT() setting. If DFSDsetNT() has not been called, the default will be 32-bit IEEE floating-point.

In addition to these required data objects, an NDG may contain any of the data objects listed in Table 4.4, “Optional SDG, NDG, and SDG-like NDG Tags.”

As illustrated in Figure 4.4, the basic NDG and SDG structures are identical. The first clue to the difference is that the NDG tag replaces the SDG tag. This is a flag to prevent older libraries from stumbling over the more important difference; the NDG data element can accommodate data that pre-Version 3.2 libraries cannot interpret. The new tag ensures that older libraries will not recognize the data object and thus will not try to interpret the new data types. For example, NDG data can include number types or a data compression scheme that a pre-Version 3.2 library will not recognize.

Figure 4.4 NDG Structure



SDG-like NDG Structures

As we have said earlier,

- SDGs, the SDS grouping structure available prior to HDF Version 3.2, could include only 32-bit floating point and Cray floating point numbers.
- NDGs, available since Version 3.2, can include 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating point numbers.
- SDG-like NDGs, also available since Version 3.2, distinguish SDSs that can still be read by the older versions of the library.

This backward compatibility is achieved by examining every SDS that is written to an HDF file. If the SDS is compatible with older libraries, it is written to the file with both SDG and NDG structures. If it is not compatible with older libraries, only the NDG structure is used.

Table 4.6 lists the objects that SDG-like NDGs must contain.

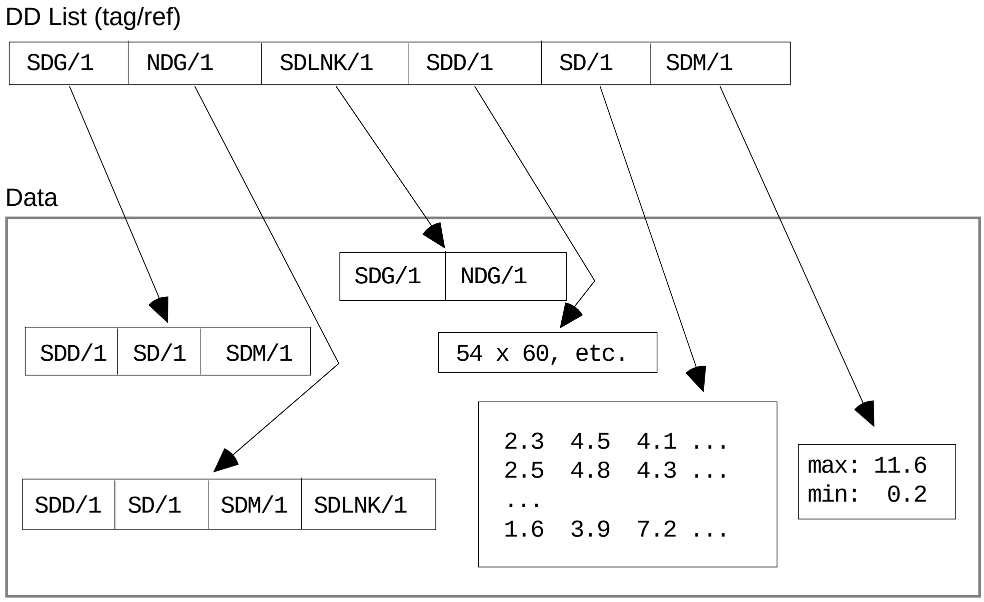
Table 4.6 Required SDG-like NDG Tags

Tag	Contents of Data Element
DFTAG_NDG	Numerical data group.
DFTAG_SDG	Scientific data group.
DFTAG_SDLNK	The NDG and SDG linked to the scientific data set in this group.
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number types of the data and of each dimension. In an SDG-like NDG, the number types are all 32-bit IEEE floating-point.
DFTAG_SD	Scientific data.

SDG-like NDGs can include the same optional data objects as described for SDGs and NDGs in Table 4.4, "Optional SDG, NDG, and SDG-like NDG Tags."

Figure 4.5 illustrates the SDG-like NDG structure.

Figure 4.5 SDG-like NDG Structure



Compatibility with Future NDG Structures

Future HDF releases will probably support additional optional SDS features. These features will fall into the following categories:

Optional and compatible features
: Optional features that are compatible with older HDF versions even though they may not be supported in the older libraries.

For example, a new time stamp attribute might be added. The time stamp would not be understood by older libraries, but it would not render them unable to read the SDS data either

Optional and incompatible features
: Optional new features that may render the data unreadable by older HDF libraries.

  For example, a compression attribute could be added. Older HDF libraries that contain no compression routines would not be able to read the compressed data.

A tag numbering convention has been developed to address this problem:

Required tags
: These tags are listed in Table 4.3, “Required SDG Tags,” Table 4.5, “Required NDG Tags,” and Table 4.6, “Required SDG-like NDG Tags.” All SDSs must contain all of the tags in at least one of these sets. (See Chapter 6, “Tag Specifications,” for the assigned tag numbers.)

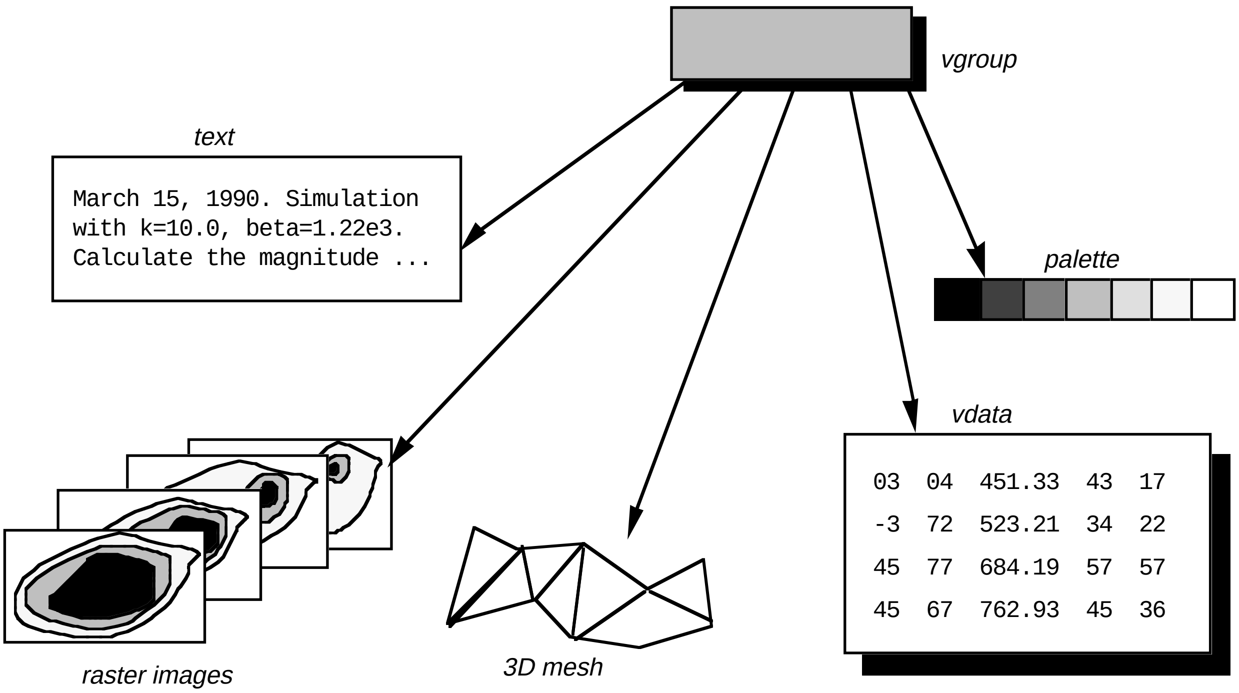
Optional-incompatible tags
: Tags for new SDS features that might render the data set unreadable by older libraries are each assigned a number `t` that falls in a special range determined by the constants `DFTAG_EREQ` and `DFTAG_BREQ`. That is, `t` must have a value such that `DFTAG_EREQ < t < DFTAG_BREQ`. When old software encounters a tag in this range that it is not able to interpret, it should not process the group.

Optional-compatible tags
: These tags can have any valid tag number not allocated to one of the other two categories.

Vsets, Vdatas, and Vgroups

Vsets, Vdatas, and Vgroups enable users to create their own grouping structures. Unlike RIGs, SDGs, and NDGs, HDF imposes no required structure; they are implemented almost entirely at the user level and are not specified in detail in HDF or in this document.* The only specifications define `DFTAG_VG`, `DFTAG_VH`, and `DFTAG_VS` and the formats of their respective data elements. A detailed discussion similar to that for the other grouping structures is, therefore, inappropriate here. Detailed information regarding the `DFTAG_VG`, `DFTAG_VH`, and `DFTAG_VS` tags can be found in Chapter 6, "Tag Specifications." Conceptual and usage information can be found in the document *NCSA HDF Vset Version 2.0* for HDF Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and the *NCSA HDF Reference Manual* for HDF Version 3.3.

Figure 4.6. Illustration of a Vset



An HDF Vset can contain any logical grouping of HDF data objects within an HDF file. Vsets resemble the UNIX file system in that they impose a basically hierarchical structure but also allow cross-linked data objects. Unlike SDSs and RISs, Vsets have no prespecified content or structure; users can use them to create structural relationships among HDF objects according to their needs. Figure 4.6 illustrates a Vset.

A Vset is identified by a *Vgroup*, an HDF object that contains information about the members of the Vset. The tag `DFTAG_VG` identifies the Vgroup which contains the tag/refs of its members, an

* Specialists in various fields are developing application program interfaces (APIs) that are becoming accepted standard interfaces within their fields. Since these APIs are implemented with high level HDF functionality and using the standard HDF user interface, they are user-level applications from the HDF development team's point of view. From the final enduser's point of view, however, these APIs create a new level of user interface. When necessary, technical specifications for these APIs and the associated interfaces will be presented by the specialized developers.

optional user-specified name, an optional user-specified class, and fields that enable the Vgroup to be extended to contain more information.

The only required Vgroup tag is the tag that defines the Vgroup itself.

Table 4.7 The Vgroup Tag

Tag	Contents of Data Element
DFTAG_VG	Vgroup

Vgroups are fully described in the document *NCSA HDF Vset, Version 2.0* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

The Raster-8 Set (Obsolete)

Current HDF versions use the raster image set (RIS) to manage raster images. But before the RIS was implemented, a simpler, less flexible set called the *raster-8 set* was used for storing 8-bit raster images. This set is no longer supported in the HDF software, although it may turn up in some older HDF files.*

Raster-8 Sets

The *raster-8 set* is defined by a set of tags that provide the basic information necessary to store 8-bit raster images and display them accurately without requiring the user to supply dimensions or color information. The raster-8 set tags are listed in Table 4.9.

Table 4.9 Raster-8 Set Tags

Tag	Contents of Data Element
`DFTAG_RI8`	8-bit raster image data
`DFTAG_CI8`	8-bit raster image data compressed with run-length encoding
`DFTAG_II8`	IMCOMP compressed image data
`DFTAG_ID8`	Image dimension record
`DFTAG_IP8`	Image palette data

Software that does not support `DFTAG_CI8` or `DFTAG_II8` must provide appropriate error indicators to higher layers that might expect to find these tags.

Compatibility Between Raster-8 and Raster Image Sets

To maintain backward compatibility with raster-8 sets, the RIS interface stores tag/refs for both types of sets. For example, if an image is stored as part of a raster image set, there is one copy each of the image dimension data, the image data, and the palette data. But there were two sets of tag/refs pointing to each data element: one for the RIS and one for the raster-8 set. The image data, for example, is associated with the tags `DFTAG_RI8` and `DFTAG_RI`.

Note: Raster-8 set support will not be maintained in future HDF releases.

Note that future HDF releases will phase out support for the raster-8 set. Therefore, new software should not expect to find both raster-8 and RIS structures supporting 8-bit raster images. Eventually, only RIS structures will be supported.

* In fact, during the first three years that RIS was used, the HDF software stored raster images in both RIS and raster-8 sets.

Chapter 5 Annotations

Chapter Overview

This chapter introduces annotations, HDF data objects used to annotate HDF files and objects.

The tags introduced in this chapter are fully described in Chapter 6, “Tag Specifications,” and are listed in the table in Appendix A, “Tags and Extended Tag Labels.”

General Description

It is often useful to attach a text annotation to an HDF file or its contents and to store that annotation in the same HDF file. HDF provides this capability through the *annotation* data object.

The data element of an annotation is a sequence of ASCII characters that can be associated with any of three types of objects:
- The file itself
- An individual HDF data object in the file
- A tag that identifies a data element

The current annotation interface supports only the first two.

Annotations come in two forms:

Label	A short, NULL-terminated string. Labels may include no embedded NULLs.
Description	A longer and more complex body of text of a pre-defined length. Descriptions may contain embedded NULLs.

Annotations are never required; they are used strictly at the discretion of the creator or user of an HDF file.

Table 5.1 shows the currently defined annotation types and their assigned tags.

Table 5.1 Annotation Tags

	Label Types	Description Types
File annotations	`DFTAG_FID`	`DFTAG_FD`
Object annotations	`DFTAG_DIL`	`DFTAG_DIA`
Tag annotations	`DFTAG_TID`	`DFTAG_TD`

The annotation interface is fully described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3

File Annotations

Any HDF file can include label annotations (`DFTAG_FID`) and/or description annotations (`DFTAG_FD`). The file annotation interface routines provided in the HDF software read and write file labels and file descriptions.

Object Annotations

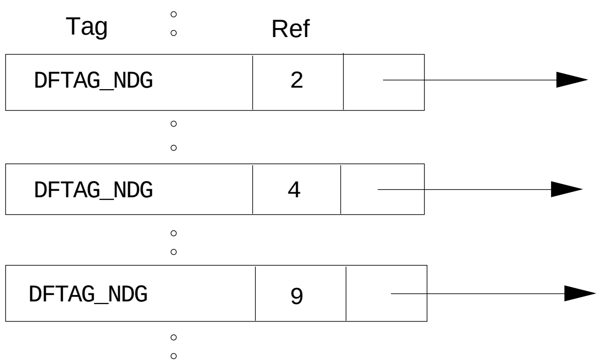
HDF data object annotation is complicated by the fact that you must uniquely identify the object being annotated. Since a tag/ref uniquely identifies a data object, the data object that a particular annotation refers to can be identified by storing the object's tag and reference number with the annotation.

Note that an HDF annotation is itself a data object, so it has its own DD. This DD has a tag/ref that points to the data element containing the annotation. The annotation data element contains the following information:

- The tag of the annotated object
- The reference number of the annotated object
- The annotation itself

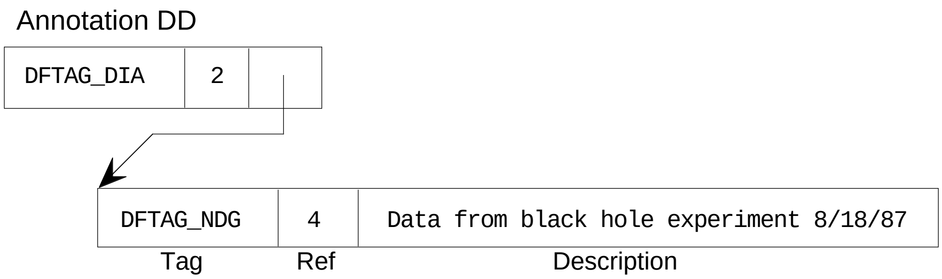
For example, suppose you have an HDF file that contains three scientific data sets (SDSs). Each SDS has its own DD consisting of the SDS tag `DFTAG_SDG` and a unique reference number, as illustrated in Figure 5.1.

Figure 5.1 Three SDS Tag/refs



Suppose you wish to attach the following annotation to the second SDS: "Data from black hole experiment 8/18/87." This text will be stored in a description annotation data object. The data element will include the tag/ref, `DFTAG_NDG/4`, and the annotation itself. Figure 5.2 illustrates the annotation data object.

Figure 5.2 Sample Annotation Data Object



Getting Reference Numbers for Object Annotations

To use annotation routines, you need to know the tags and reference numbers of the objects you wish to annotate.

The following routines return the most recent reference number used in either reading or writing the specified type of data object:

`DFSDlastref`	SDS data objects
`DFR8lastref`	RIS data objects
`DFPlastref`	Palettes
`DFANlastref`	Annotations

Reference numbers for other objects can be obtained with the routine `Hfindnextref`, a general purpose HDF routine that searches an HDF file sequentially for reference numbers associated with a given tag.

These routines are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Chapter 6 Tag Specifications

Chapter Overview

This chapter addresses issues related to HDF tags and the data they represent. The first section provides general information about tags and their interpretation. The remainder of the chapter contains a complete list of tags supported by NCSA HDF Version 3.3 and detailed tag specifications.

The HDF Tag Space

As discussed in Chapter 1, "The Basic Structure of HDF Files," 16 bits are allotted for an HDF tag number. This provides for 65535 possible tags, ranging from 1 to 65535; zero (0) is not used. This tag space is divided into three ranges:

1	–	32767	Reserved for NCSA-supported tags
32768	–	64999	Set aside as user-definable tags
65000	–	65535	Reserved for expansion of the format

No restrictions are placed on the user-definable tags. Note that tags from this range are not expected to be unique across user-developed HDF applications.

The rest of this chapter is devoted to the NCSA-supported tags in the range 1 to 32767.

Extended Tags and Alternate Physical Storage Methods

Prior to HDF Version 3.2, each data element had to be stored in one contiguous block in the basic HDF file. Version 3.2 introduced *extended tags*, a mechanism supporting alternate physical data element storage structures. All NCSA-supported tags with variable-sized data elements can take advantage of the extended tag features.

Extended Tag Implementation

Extended tags are automatically recognized by current versions of the HDF library and interpreted according to a description record. The description record, a complete data element, identifies the type of extended element and provides the relevant parameters for data retrieval.

Extended tags currently support two styles of alternate physical storage:

Linked block elements are stored in several non-contiguous blocks within the basic HDF file.

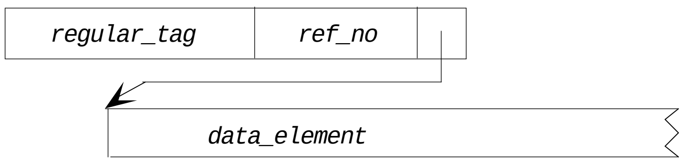
External elements are stored in a separate file, external to the basic HDF file.

Every NCSA-supported tag is represented in HDF libraries and files by a tag number. NCSA-supported tags that take advantage of alternative physical storage features have an alternative tag number, called an *extended tag number*, that appears instead of the original tag number when an alternative physical storage method is in use.

When NCSA determines that an extended tag should be defined for a given tag, the extended tag number is determined by performing an arithmetic OR with the original tag number and the hexadecimal number 0x4000. For example, the tag `DFTAG_RI` points to a data element containing a raster image. If the data element is stored contiguously in the same HDF file, the DD contains the tag number `302`; if the data element is stored either in linked blocks or in an external file, the DD contains the extended tag number `16384`.

If a data object uses a regular tag number, its storage structure will be exactly as described in the "Tag Specifications" section of this chapter. Figure 6.1 illustrates this general structure with the DD pointing directly to a single, contiguous data block.

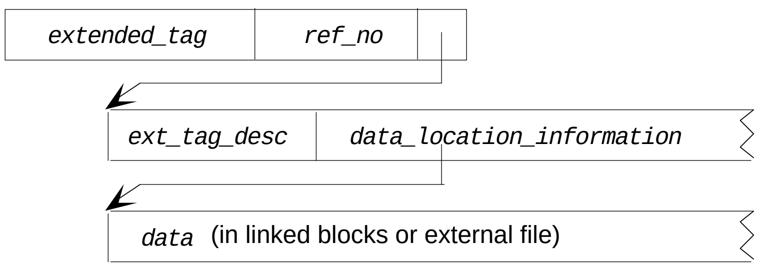
Figure 6.1 Regular Data Object



regular_tag	Tag number
ref_no	Reference number
data_element	The data element

If a data object uses an extended tag, the storage structure will appear generally as illustrated in Figure 6.2. The DD will point to an extended tag description record which in turn will point to the data.

Figure 6.2 Data Object with Extended Tag



extended_tag	Extended tag number
ref_no	Reference number
ext_tag_desc	A 32-bit constant defined in `Hdfi.h` that identifies the type of alternative storage involved. Current definitions include `EXT_LINKED` for linked block elements or `EXT_EXTERN` for external elements.

data_location_information
Information identifying and describing the linked blocks or external file

data — The data, stored either in linked blocks or in an external file

Since the HDF tools were modified for HDF Version 3.2 to handle extended tags automatically, the only thing the user ever has to do is specify the use of either the linked blocks mechanism or an external file. Once that has been specified, the user can forget about extended tags entirely; the HDF library will manage everything correctly.

There is only one circumstance under which an HDF user will need to be concerned with the difference between regular tag numbers and extended tag numbers. If a user bypasses the regular HDF interface to examine a raw HDF file, that user will have to know the extended tag numbers, their significance, and the alternative storage structures.

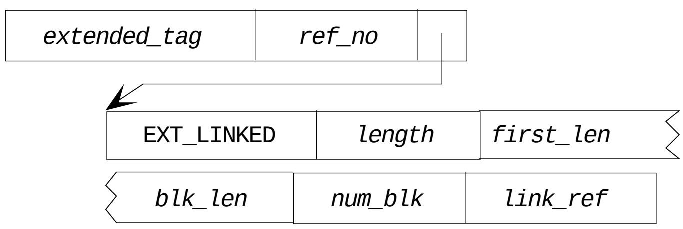
Linked Block Elements

As mentioned above, data elements had to be stored as single contiguous blocks within the basic HDF file prior to HDF Version 3.2. This meant that if a data element grew larger than the allotted space, the file had to be erased from its current location and rewritten at the end of the file.

Linked blocks provide a convenient means of addressing this problem by linking new data blocks to a pre-existing data element. Linked block elements consist of a series of data blocks chained together in a linked list (similar to the DD list). The data blocks must be of uniform size, except for the first block, which is considered a special case.

The linked block data element is a description record beginning with the constant `EXT_LINKED`, which identifies the linked block storage method. The rest of the record describes the organization of the data element stored as linked blocks. Figure 6.3 illustrates a linked block description record.

Figure 6.3 Linked Block Description Record



extended_tag — The extended tag counterpart of any NCSA standard tag (16-bit integer)

ref_no — Reference number (16-bit integer)

`EXT_LINKED` — Constant identifying this as a linked block description record (32-bit integer)

length — Length of entire element (32-bit integer)

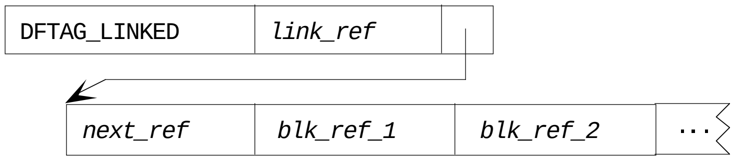
first_len — Length of the first data block (32-bit integer)

blk_len — Length of successive data blocks (32-bit integer)

num_blk — Number of blocks per block table (32-bit integer) *link_ref* Reference number of first block table (16-bit integer)

The *link_ref* field of the description record gives the reference number of the first linked block table for the element. This table is identified by the tag/ref DFTAG_LINKED/*link_ref* and contains *num_blk* entries. There may be any number of linked block tables chained together to describe a linked block element. Figure 6.4 illustrates a linked block table.

Figure 6.4 A Linked Block Table



link_ref Reference number for this table (16-bit integer)

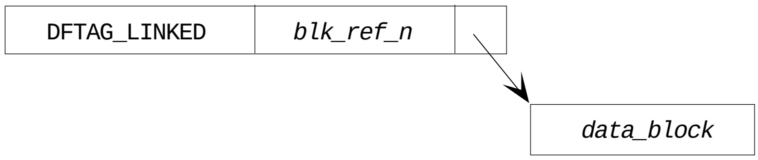
next_ref Reference number for next table (16-bit integer)

blk_ref_n Reference number for data block (16-bit integer)

The *next_ref* field contains the reference number of the next linked block table. A value of zero (0) in this field indicates that there are no additional linked block tables associated with this element.

The *blk_ref_n* fields of each linked block table contain reference numbers for the individual data blocks that make up the data portion of the linked block element. These data blocks are identified by the tag/ref DFTAG_LINKED/*blk_ref_n* as illustrated in Figure 6.5. Although it may seem ambiguous to use the same tag to refer to two different objects, this ambiguity is resolved by the context in which the tags appear.

Figure 6.5 A Data Block



blk_ref_n Reference number for this data block (16-bit integer)

data_block Block of actual data (size specified by *first_len* or *blk_len* in the description record)

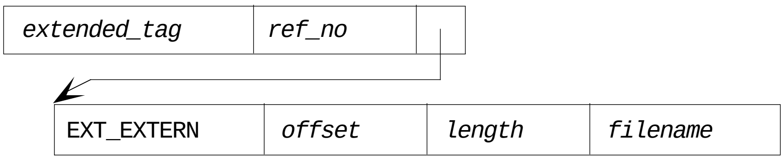
Linked block elements can be created using the function HLcreate(), which is discussed in Chapter 3, "The HDF General Purpose Interface."

External Elements

External elements allow the data portion of an HDF element to reside in a separate file. The potential of external data elements is largely unexplored in the HDF context, although other file formats (most notably the Common Data Format, CDF, from NASA) have used external data elements to great advantage.

Because there has been little discussion of external elements within the HDF user community, the structure of these elements is still not completely defined. Figure 6.6 shows a diagram of the suggested structure for an external element.

Figure 6.6 External Element Description Record



extended_tag	The extended tag counterpart of any NCSA standard tag (16-bit integer)
ref_no	Reference number (16-bit integer)
EXT_EXTERN	Constant identifying this as an external element description record (16-bit integer)
offset	Location of the data within the external file (32-bit integer)
length	Length in bytes of the data in the external file (32-bit integer)
filename	Non-null terminated ASCII string naming the external file (any length)

An external element description record begins with the constant `EXT_EXTERN`, which identifies the data object as having an externally stored data element. The rest of the description record consists of the specific information required to retrieve the data.

External elements can be created using the function `HXcreate()`, which is discussed in Chapter 3, “The HDF General Purpose Interface.”

Tag Specifications

The following pages contain the specifications of all the NCSA-supported tags in HDF Version 3.3. Each entry contains the following information:

- The tag (in capital letters in the left margin)
- The full name of the tag (on the first line to the right)
- The type and, where possible, the amount of data in the corresponding data element (on the second line to the right)

  When the data element is a variable-sized data structure—such as text, a string, or a variable-sized array—the amount of data cannot be specified exactly. Where possible, a formula is provided to estimate the amount of data. The string `? bytes` appears when neither the size nor the structure of the data element can be specified.
- The tag number in decimal/(hexadecimal) (on the third line to the right)
- A diagram illustrating the structure of the tag and its associated data

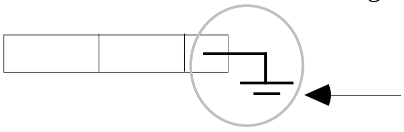
  Since all DDs that point to a data element contain data length and data offset fields, these fields are not included in the illustrations.
- A full specification of the tag, including a description of the data element and a discussion of its intended use.

Tags are roughly grouped according to the roles they play:
- Utility tags
- Annotation tags
- Compression tags
- Raster Image tags
- Composite image tags
- Vector image tags
- Scientific data set tags
- Vset tags
- Obsolete tags

These groupings imply a general context for the use of each tag; they are not meant to restrict their use.

Please note the subsection "Obsolete Tags." These tags have fallen out of use with the continuing development of HDF. They are still recognized by the HDF library, but users should not write new objects using them; they may eventually be dropped from the HDF specification.

In the following discussion, the ground symbol indicates that the DD for this tag includes no pointer to a data element. I.e., there is never a data element associated with the tag.

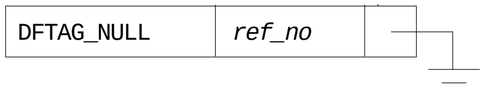


This symbol indicates that there is no data element associated with the tag.

Utility Tags

`DFTAG_NULL`

No data
0 bytes
1 (0x0001)

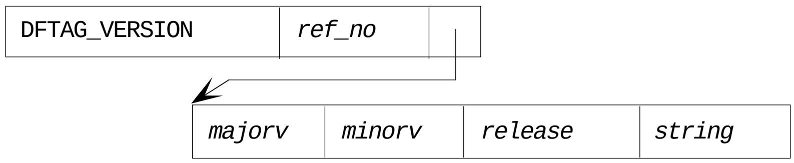


ref_no Reference number (16-bit integer; always 0)

This tag is used for place holding and to fill empty portions of the data description block. The length and offset fields (not shown) of a `DFTAG_NULL` DD must be zero (`0`).

`DFTAG_VERSION`

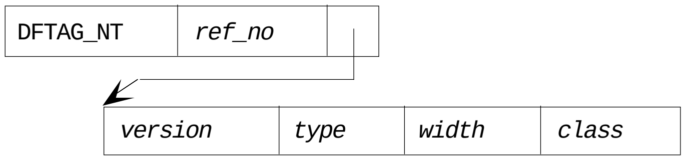
Library version number
12 bytes plus the length of a string
30 (0x001E)



ref_no Reference number (16-bit integer)
majorv Major version number (32-bit integer)
minorv Minor version number (32-bit integer)
release Release number (32-bit integer)
string Non-null terminated ASCII string (any length)

The data portion of this tag contains the complete version number and a descriptive string for the latest version of the HDF library to write to the file.

`DFTAG_NT` Number type
4 bytes
106 (0x006A)



ref_no	Reference number (16-bit integer)
version	Version number of NT information (8-bit integer)
type	Unsigned integer, signed integer, unsigned character, character, floating point, double precision floating point (8-bit code)
width	Number of bits, all of which are assumed to be significant (8-bit code)
class	A generic value, with different interpretations depending on type: floating point, integer, or character (8-bit code)

Several values that may be used for each of the three types in the field CLASS are listed in Table 6.1. This is not an exhaustive list.

Table 6.1 Number Type Values

Type	Mnemonic	Value
Floating point	DFNTF_NONE	0
	DFNTF_IEEE	1
	DFNTF_VAX	2
	DFNTF_CRAY	3
	DFNTF_PC	4
	DFNTF_CONVEX	5
Integer	DFNTI_MBO	1
	DFNTI_IBO	2
	DFNTI_VBO	4
Character	DFNTC_ASCII	1
	DFNTC_EBCDOC	2
	DFNTC_BYTE	0

The number type flag is used by any other element in the file to indicate specifically what a numeric value looks like. Other tag types should contain a reference number pointer to an `DFTAG_NT` instead of containing their own number type definitions.

The version field allows expansion of the number type information, in case some future number types cannot be described using the fields currently defined. Successive versions of the `DFTAG_NT` may be substantially different from the current definition, but backward compatibility will be maintained. The current `DFTAG_NT` version number is 1.

DFTAG_MT

Machine type
0 bytes
107 (0x006B)

DFTAG_MT	*double*	*float*	*int*	*char*	

double Specifies method of encoding double precision floating point (4-bit code)

float Specifies method of encoding single precision floating point (4-bit code)

int Specifies method of encoding integers (4-bit code)

char Specifies method of encoding characters (4-bit code)

DFTAG_MT specifies that all unconstrained or partially constrained values in this HDF file are of the default type for that hardware. When DFTAG_MT is set to VAX, for example, all integers will be assumed to be in VAX byte order unless specifically defined otherwise with a DFTAG_NT tag. Note that all of the headers and many tags, the whole raster image set for example, are defined with bit-wise precision and will not be overridden by the DFTAG_MT setting.

For DFTAG_MT, the reference field itself is the encoding of the DFTAG_MT information. The reference field is 16 bits, taken as four groups of four bits, specifying the types for double-precision floating point, floating point, integer, and character respectively. This allows 16 generic specifications for each type.

To the user, these will be defined constants in the header file hdf.h, specifying the proper descriptive numbers for Sun, VAX, Cray, Convex, and other computer systems. If there is no DFTAG_MT in a file, the application may assume that the data in the file has been written on the local machine; any portability problems must be addressed by the user. For this reason, we recommend that all HDF files contain a DFTAG_MT for maximum portability.

Currently available data encodings are listed in Table 6.2.

Table 6.2 Available Machine Types

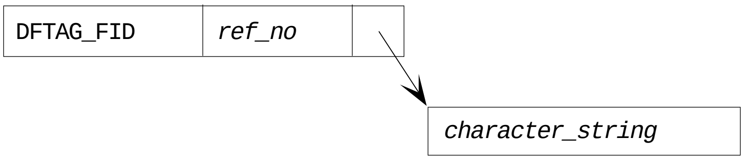
Type	Available Encodings
Double precision floating point	IEEE64 VAX64 CRAY128
Floating point	IEEE32 VAX32 CRAY64
Integers	VAX32 Intel16 Intel32 Motorola32 CRAY64
Characters	ASCII EBCDIC

New encodings can be added for each data type as the need arises.

Annotation Tags

`DFTAG_FID`

File identifier
String
100 (0x0064)



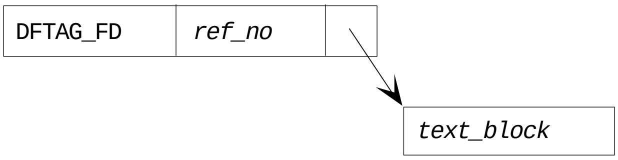
ref_no Reference number (16-bit integer)

character_string Non-null terminated ASCII text (any length)

This tag points to a string which the user wants to associate with this file. The string is not null terminated. The string is intended to be a user-supplied title for the file.

`DFTAG_FD`

File description
Text
101 (0x0065)



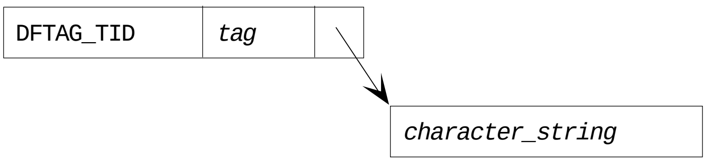
ref_no Reference number (16-bit integer)

text_block Non-null terminated ASCII text (any length)

This tag points to a block of text describing the overall file contents. The text can be any length. The block is not null terminated. The text is intended to be user-supplied comments about the file.

`DFTAG_TID`

Tag identifier
String
102 (0x0066)



tag Tag number to which this tag refers (16-bit integer)

character_string
Non-null terminated ASCII text (any length)

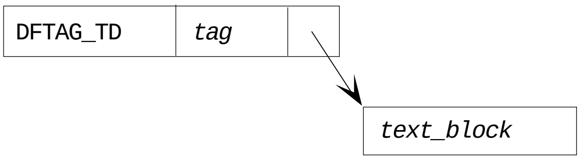
The data for this tag is a string that identifies the functionality of the tag indicated in the space normally used for the reference number. For example, the tag identifier for `DFTAG_TID` might point to data that reads "tag identifier."

Many tags are identified in the HDF specification, so it is usually unnecessary to include their identifiers in the HDF file. But with user-defined tags or special-purpose tags, the only way for a human reader to diagnose what kind of data is stored in a file is to read tag identifiers. Use tag descriptions to define even more detail about your user-defined tags.

Note that with this tag you may make use of the user-defined tags to check for consistency. Although two persons may use the same user-defined tag, they probably will not use the same tag identifier.

`DFTAG_TD`

Tag description
Text
103 (0x0067)



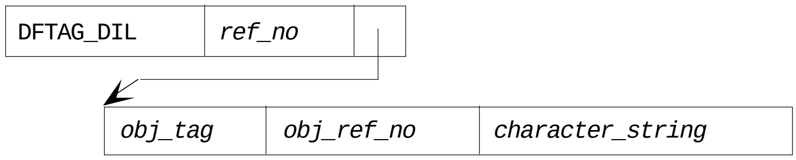
tag	Tag number to which this tag refers (16-bit integer)
text_block	Non-null terminated ASCII text (any length)

The data for this tag is a text block which describes in relative detail the functionality and format of the tag which is indicated in the space normally occupied by the reference number. This tag is intended to be used with user-defined tags and provides a medium for users to exchange files that include human-readable descriptions of the data.

It is important to provide everything that a programmer might need to know to read the data from your user-defined tag. At the minimum, you should specify everything you would need to know in order to retrieve your data at a later date if the original program were lost.

`DFTAG_DIL`

Data identifier label
String
104 (0x0068)



ref_no Reference number (16-bit integer)

obj_tag Tag number of the data to which this label applies (16-bit integer)

obj_ref_no Reference number of the data object to which this label applies (16-bit integer)

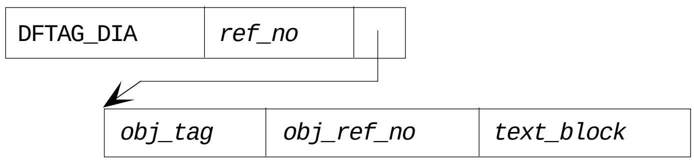
character_string Non-null terminated ASCII text (any length)

The `DFTAG_DIL` data object consists of a tag/ref followed by a string. The string serves as a label for the data identified by the tag/ref.

By including `DFTAG_DIL` tags, you can give a data object a label for future reference. For example, `DFTAG_DIL` can be used to assign titles to images.

`DFTAG_DIA`

Data identifier annotation
Text
105 (0x0069)



ref_no Reference number (16-bit integer)

obj_tag Tag number of the data to which this annotation applies (16-bit integer)

obj_ref_no Reference number of the data object to which this annotation applies (16-bit integer)

text_block Non-null terminated ASCII text (any length)

The `DFTAG_DIA` data object consists of a tag/ref followed by a text block. The text block serves as an annotation of the data identified by the tag/ref.

With a `DFTAG_DIA` tag, any data object can have a lengthy, user-written description. This can be used to include comments about images, data sets, source code, and so forth.

Compression Tags

`DFTAG_RLE`

Run length encoded data
0 bytes
11 (0x000B)

DFTAG_RLE	*ref_no*	

ref_no Reference number (16-bit integer)

This tag is used in the `DFTAG_ID` compression field and in other places to indicate that an image or section of data is encoded with a run-length encoding scheme. The RLE method used is byte-wise. Each run is preceded by a count byte. The low seven bits of the count byte indicate the number of bytes (n). The high bit of the count byte indicates whether the next byte should be replicated *n* times (high bit = 1), or whether the next *n* bytes should be included as is (high bit = 0).

See also: `DFTAG_ID` in "Raster Image Tags"
`DFTAG_NDG` in "Scientific Data Set Tags"

`DFTAG_IMC`

IMCOMP compressed data
0 bytes
12 (0x000C)

DFTAG_IMC	*ref_no*	

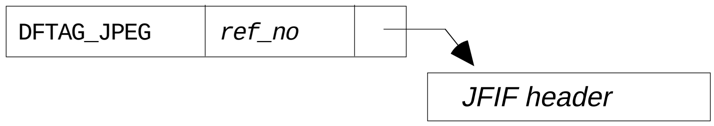
ref_no Reference number (16-bit integer)

This tag is used in the `DFTAG_ID` compression field and in other places to indicate that an image or section of data is encoded with an IMCOMP encoding scheme. This scheme is a 4:1 aerial averaging method which is easy to decompress. It counts color frequencies in 4x4 squares to optimize color sampling.

See also: `DFTAG_ID` in "Raster Image Tags"
`DFTAG_NDG` in "Scientific Data Set Tags"

`DFTAG_JPEG`

24-bit JPEG compression information
? bytes
13 (0x000D)

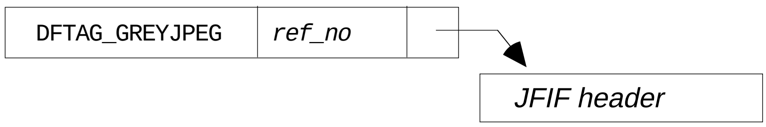


ref_no Reference number (16-bit integer)

This tag points to header information for 24-bit JPEG compressed images. The data in this tag is identical to the header data stored in a JFIF (JPEG File Interchange Format) file up to the start-of-frame parameter. The start-of-frame parameter and all further data for the JPEG image is stored in the associated `DFTAG_CI` data element which is the companion to the `DFTAG_JPEG` element. (See the document *JPEG File Interchange Format*[*] for a detailed description of the file format.)

`DFTAG_GREYJPEG`

8-bit JPEG compression information
? bytes
14 (0x000E)



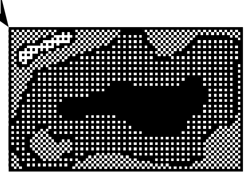
ref_no Reference number (16-bit integer)

This tag points to header information for 8-bit JPEG compressed images. The data in this tag is identical to the header data stored in a JFIF (JPEG File Interchange Format) file up to the start-of-frame parameter (see the JFIF format document for further details). The start-of-frame parameter and all further data for the JPEG image is stored in the associated `DFTAG_CI` data element which is the companion to the `DFTAG_JPEG` element.

[*] The document *JPEG File Interchange Format* has not been published in a regular periodical. An electronic copy is available as a Postscript file from NCSA's FTP server `ftp.ncsa.uiuc.edu` in the same directory as this document, *NCSA HDF Specification and Developer's Guide*. Printed copies are available from C-Cube Microsystems, 1778 McCarthy Boulevard, Milpitas, CA 95035 (phone: 408-944-6300. Fax: 408-944-6314. Current email contact: `eric@c3.pla.ca.us`).

`DFTAG_CI` Compressed raster image
? bytes
303 (0x012F

DFTAG_CI	*ref_no*	



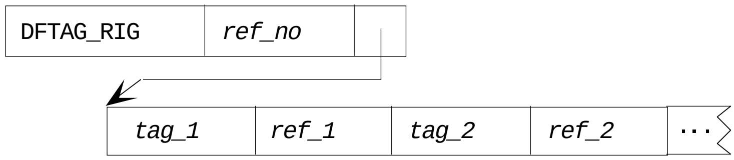
ref_no Reference number (16-bit integer)

This tag points to a stream of bytes that make up a compressed image. The type of compression, together with any necessary parameters, are stored as a separate data object. For example, if `DFTAG_JPEG` is contained in the same raster image group, the stream of bytes contains the sratt-of-frame parameter and all further data for the JPEG image. Other parameters are stored in the `DFTAG_JPEG` object.

Raster Image Tags

DFTAG_RIG

Raster image group
*n**4 bytes (where *n* is the number of data objects in the group)
306 (0x0132)



ref_no Reference number (16-bit integer)

tag_n Tag number for n^{th} member of the group (16-bit integer)

ref_n Reference number for n^{th} member of the group (16-bit integer)

The RIG data element contains the tag/refs of all the data objects required to display a raster image correctly. Application programs that deal with RIGs should read all the elements of a RIG and process those identifiers which it can display correctly. Even if the application cannot process *all* of the objects, the objects that it can process will be usable.

Table 6.3 lists the tags that may appear in an RIG.

Table 6.3 Available RIG Tags

Tag	Description
DFTAG_ID	Image dimension record
DFTAG_RI	Raster image
DFTAG_XYP	X-Y position
DFTAG_LD	LUT dimension
DFTAG_LUT	Color lookup table
DFTAG_MD	Matte channel dimension
DFTAG_MA	Matte channel
DFTAG_CCN	Color correction
DFTAG_CFM	Color format
DFTAG_AR	Aspect ratio

Example

DFTAG_ID, DFTAG_RI, DFTAG_LD, DFTAG_LUT
Assume that an image dimension record, a raster image, an LUT dimension record, and an LUT are all required to display a particular raster image correctly. These data objects can be associated in an RIG so that an application can read the image dimensions then the image. It will then read the lookup table and display the image.

DFTAG_ID
DFTAG_LD
DFTAG_MD

DFTAG_ID	DFTAG_LD	DFTAG_MD
Image dimension	LUT dimension	Matte dimension
20 bytes	20 bytes	20 bytes
300 (0x012C)	307 (0x0133)	308 (0x0134)

DFTAG_ID	*ref_no*	

x_dim	*y_dim*	DFTAG_NT	*NT_ref*
elements	*interlace*	*comp_tag*	*comp_ref*

ref_no Reference number (16-bit integer)

x_dim Length of x (horizontal) dimension (32-bit integer)

y_dim Length of y (vertical) dimension (32-bit integer)

NT_ref Reference number for number type information

elements Number of elements that make up one entry (16-bit integer)

interlace Type of interlacing used (16-bit integer)
`0` The components of each pixel are together.
`1` Color elements are grouped by scan lines.
`2` Color elements are grouped by planes.

comp_tag Tag which tells the type of compression used and any associated parameters (16-bit integer)

comp_ref Reference number of compression tag (16-bit integer)

These three dimension records have exactly the same format; they specify the dimensions of the 2-dimensional arrays after which they are named and provide information regarding other attributes of the data in the array:

`DFTAG_ID` specifies the dimensions of a `DFTAG_RI.`
`DFTAG_LD` specifies the dimensions of a `DFTAG_LUT`.
`DFTAG_MD` specifies the dimensions of a `DFTAG_MA`.

Other attributes described in the image dimension record include the number type of the elements, the number of elements per pixel, the interlace scheme used, and the compression scheme used (if any).

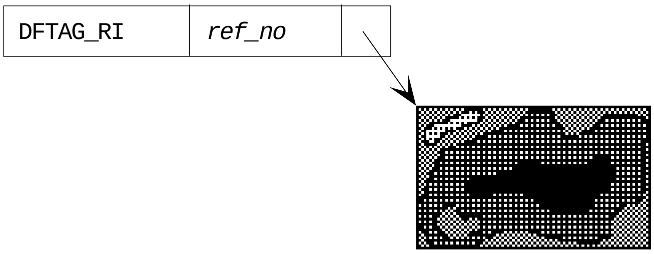
For example, a 512x256 row-wise 24-bit raster image with each pixel stored as RGB bytes would have the following values:

x_dim	512
y_dim	256
NT_ref	UINT8
elements	3 (3 elements per pixel: e.g., R, G, and B)
interlace	0 (RGB values not separated)
comp_tag	0 (no compression is used)

The diagram above illustrates the tag `DFTAG_ID`. The `DFTAG_LD` and `DFTAG_MD` diagrams would be identical except for the tag name in the fist cell, whch would be `DFTAG_LD` and `DFTAG_MD`, respectively.

`DFTAG_RI`

Raster image
*xdim*ydim*elements*NTsize* bytes (*xdim*, *ydim*, *elements*, and *NTsize* are specified in the corresponding `DFTAG_ID`)
302 (0x012E)

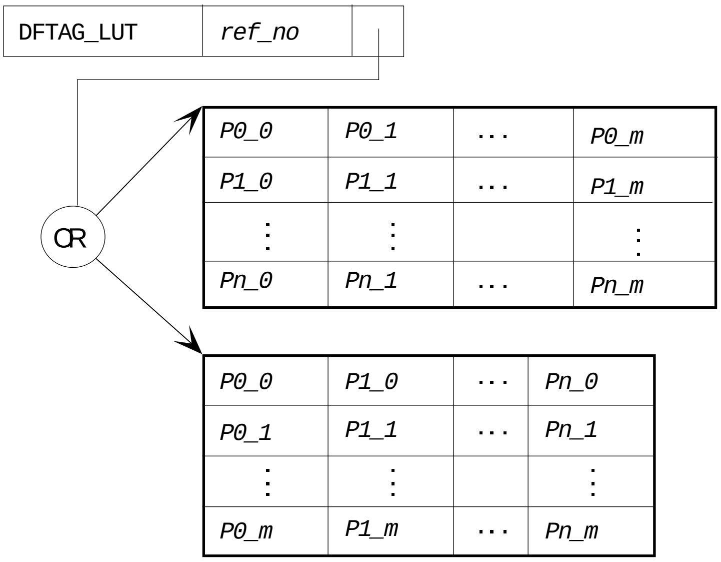


`ref_no` Reference number (16-bit integer)

This tag points to raster image data. It is stored in row-major order and must be interpreted as specified by *`interlace`* in the related `DFTAG_ID`.

`DFTAG_LUT`

Lookup table
*xdim*ydim*elements*NTsize* bytes (*xdim*, *ydim*, *elements*, and *NTsize* are specified in the corresponding `DFTAG_ID`)
301 (0x012D)



`ref_no` Reference number (16-bit integer)

`Pn_m` m^{th} value of parameter n (size is specified by the `DFTAG_NT` in the corresponding `DFTAG_LD`)

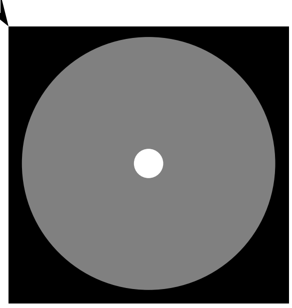
The `DFTAG_LUT`, sometimes called a palette, is used to assign colors to data values. When a raster image consists of data values which are going to be interpreted through an LUT capability, the `DFTAG_LUT` should be loaded along with the image.

The most common lookup table is the RGB lookup table which will have X dimension = 256 and Y dimension = 1 with three elements per entry, one each for red, green, and blue. The interlace will be either 0, where the LUT values are given RGB, RGB, RGB, ..., or 1, where the LUT values are given as 256 reds, 256 greens, 256 blues.

`DFTAG_MA`

Matte channel
*xdim***ydim***elements***NTsize* bytes (*xdim*, *ydim*, *elements*, and *NTsize* are specified in the corresponding `DFTAG_ID`)
309 (0x0135)

DFTAG_MA	*ref_no*	

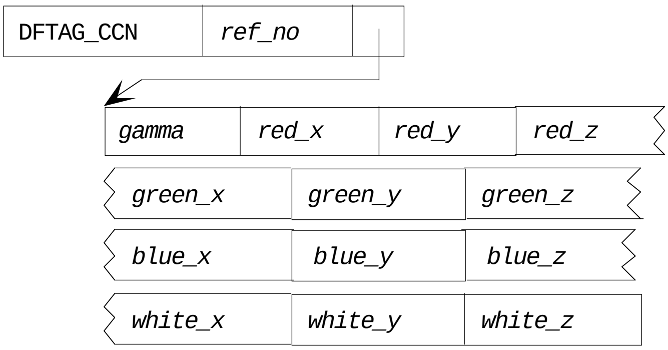


ref_no Reference number (16-bit integer)

The `DFTAG_MA` data object contains transparency data which can be used to facilitate the overlaying of images. The data consists of a 2-dimensional array of unsigned 8-bit integers ranging from 0 to 255. Each point in a `DFTAG_MA` indicates the transparency of the corresponding point in a raster image of the same dimensions. A value of 0 indicates that the data at that point is to be considered totally transparent, while a value of 255 indicates that the data at that point is totally opaque. It is assumed that a linear scale applies to the transparency values, but users may opt to interpret the data in any way they wish.

`DFTAG_CCN`

Color correction
52 bytes (usually)
310 (0x0136)



ref_no Reference number (16-bit integer)

gamma Gamma parameter (32-bit IEEE floating point)

red_x, *red_y*, and *red_z*
Red x, y, and z correction factors (32-bit IEEE floating point)

green_x, *green_y*, and *green_z*
Green x, y, and z correction factors (32-bit IEEE floating point)

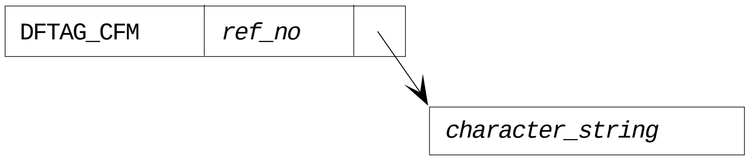
blue_x, *blue_y*, and *blue_z*
Blue x, y, and z correction factors (32-bit IEEE floating point)

white_x, *white_y*, and *white_z*
White x, y, and z correction factors (32-bit IEEE floating point)

Color correction specifies the Gamma correction for the image and color primaries for the generation of the image.

`DFTAG_CFM`

Color format
String
311 (0x0137)



ref_no	Reference number (16-bit integer)
character_string	Non-null terminated ASCII string (any length)

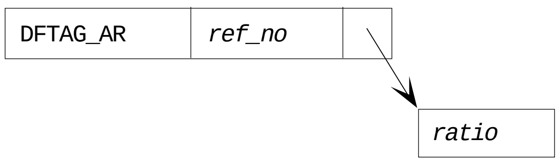
The color format data element contains a string of uppercase characters that indicates how each element of each pixel in a raster image is to be interpreted. Table 6.4 lists the available color format strings.

Table 6.4 Color Format String Values

String	Description
VALUE	Pseudo-color, or just a value associated with the pixel
RGB	Red, green, blue model
XYZ	Color-space model
HSV	Hue, saturation, value model
HSI	Hue, saturation, intensity
SPECTRAL	Spectral sampling method

`DFTAG_AR`

Aspect ratio
4 bytes
312 (0x0138)



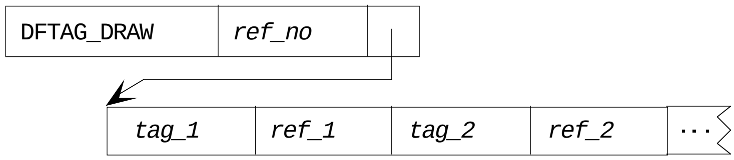
ref_no	Reference number (16-bit integer)
ratio	Ratio of width to height (32-bit IEEE float)

The data for this tag is the visual aspect ratio for this image. The image should be visually correct if displayed on a screen with this aspect ratio. The data consists of one floating-point number which represents width divided by height. An aspect ratio of 1.0 indicates a display with perfectly square pixels; 1.33 is a standard aspect ratio used by many monitors.

Composite Image Tags

`DFTAG_DRAW`

Draw
*n**4 bytes (where *n* is the number of data objects that make up the composite image)
400 (0x0190)



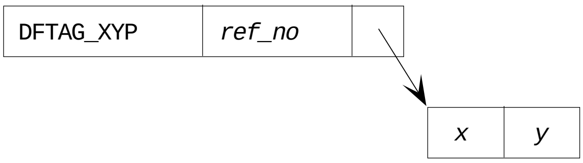
ref_no	Reference number (16-bit integer)
tag_n	Tag number of the n^{th} member of the draw list (16-bit integer)
ref_n	Reference number of the n^{th} member of the draw list (16-bit integer)

The `DFTAG_DRAW` data element consists of a list of tag/refs that define a composite image. The data objects indicated should be displayed in order. This can include several RIGs which are to be displayed simultaneously. It can also include vector overlays, like `DFTAG_T14`, which are to be placed on top of an RIG.

Some of the elements in a `DFTAG_DRAW` list may be instructions about how images are to be composited (XOR, source put, anti-aliasing, etc.). These are defined as individual tags.

`DFTAG_XYP`

XY position
8 bytes
500 (0x01F4)



ref_no	Reference number (16-bit integer)
x	X-coordinate (32-bit integer)
y	Y-coordinate (32-bit integer)

`DFTAG_XYP` is used in composites and other groups to indicate an XY position on the screen. For this, (0,0) is the lower left corner of the print area. X is the number of pixels to the right along the horizontal axis and Y is the number of pixels up on the vertical axis. The X and Y coordinates are two 32-bit integers.

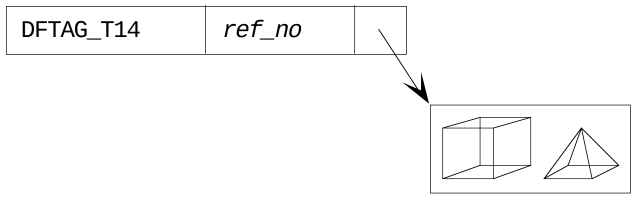
For example, if `DFTAG_XYP` is present in a `DFTAG_RIG`, the `DFTAG_XYP` specifies the position of the lower left corner of the raster image on the screen.

See also: `DFTAG_DRAW` in this section

Vector Image Tags

`DFTAG_T14`

Tektronix 4014
? bytes
602 (0x25A)

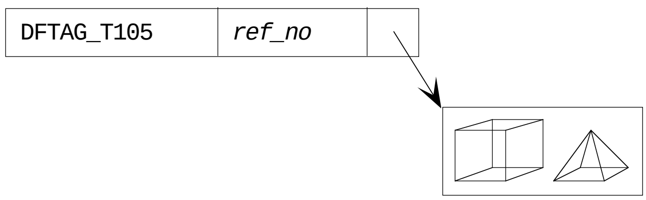


ref_no Reference number (16-bit integer)

This tag points to a Tektronix 4014 data stream. The bytes in the data field, when read and sent to a Tektronix 4014 terminal, will display a vector image. Only the lower seven bits of each byte are significant. There are no record markings or non-Tektronix codes in the data.

`DFTAG_T105`

Tektronix 4105
? bytes
603 (0x25B)

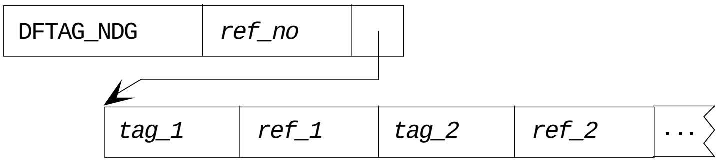


ref_no Reference number (16-bit integer)

This tag points to a Tektronix 4105 data stream. The bytes in the data field, when read and sent to a Tektronix 4105 terminal, will be displayed as a vector image. Only the lower seven bits of each byte are significant. Some terminal emulators will not correctly interpret every feature of the Tektronix 4105 terminal, so you may wish to use only a subset of the available Tektronix 4105 vector commands.

Scientific Data Set Tags

`DFTAG_NDG`	Numeric data group *n**4 bytes (where *n* is the number of data objects in the group.) 720 (0x02D0)



ref_no	Reference number (16-bit integer)
tag_n	Tag number of n^{th} member of the group (16-bit integer)
ref_n	Reference number of n^{th} member of the group (16-bit integer)

The NDG data contains a list of tag/refs that define a scientific data set. `DFTAG_NDG` supersedes the old `DFTAG_SDG`, which became obsolete upon the release on HDF Version 3.2. A more complete explanation of the relationship between `DFTAG_NDG` and `DFTAG_SDG` can be found in Chapter 4, “Sets and Groups.”

All of the members of an NDG provide information for correctly interpreting and displaying the data. Application programs that deal with NDGs should read all of the elements of a NDG and process those data objects which it can use. Even if an application cannot process all of the objects, the objects that it can understand will be usable.

Table 6.5 lists the tags that may appear in an NDG.

Table 6.5 Available NDG Tags

Tag	Description
DFTAG_SDD	Scientific data dimension record (rank and dimensions)
DFTAG_SD	Scientific data
DFTAG_SDS	Scales
DFTAG_SDL	Labels
DFTAG_SDU	Units
DFTAG_SDF	Formats
DFTAG_SDM	Maximum and minimum values
DFTAG_SDC	Coordinate system
DFTAG_CAL	Calibration information
DFTAG_FV	Fill value
DFTAG_LUT	Color lookup table
DFTAG_LD	Lookup table dimension record
DFTAG_SDLNK	Link to old-style `DFTAG_SDG`

Example

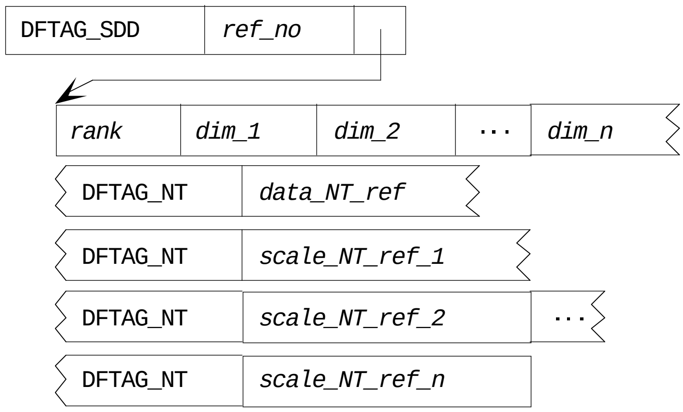
`DFTAG_SDD`, `DFTAG_SD`, `DFTAG_SDM`

Suppose that an NDG contains a dimension record, scientific data, and the maximum and minimum values of the data. These data objects can be associated in an NDG so that an application can read the rank and dimensions from the dimension record and then read the data array. If the application needs maximum and minimum values, it will read them as well.

See also: Chapter 4, “Sets and Groups”

`DFTAG_SDD`

Scientific data dimension record
6 + 8**rank* bytes
701 (0x02BD)



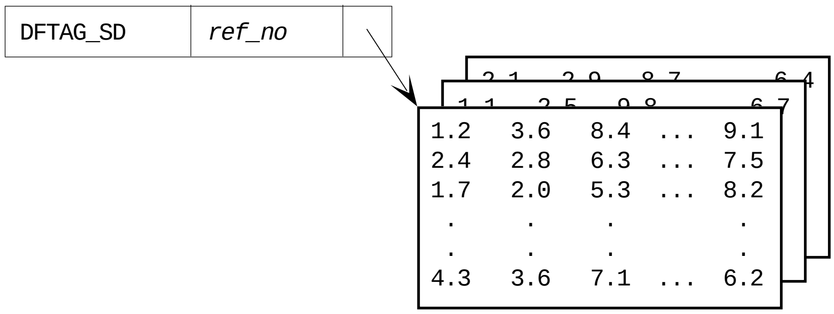
ref_no	Reference number (16-bit integer)
rank	Number of dimensions (16-bit integer)
dim_n	Number of values along the n^{th} dimension (32-bit integer)
data_NT_ref	Reference number of `DFTAG_NT` for data (16-bit integer)
scale_NT_ref_n	Reference number for `DFTAG_NT` for the scale for the n^{th} dimension (16-bit integer)

This record defines the rank and dimensions of the array in the scientific data set. For example, a `DFTAG_SDD` for a 500x600x3 array of floating-point numbers would have the following values and components.

Rank: 3
Dimensions: 500, 600, and 3.
One data NT
Three scale NTs

`DFTAG_SD`

Scientific data
*NTsize*x*y*z*...* bytes (where NTsize is the size of the data NT specified in the corresponding `DFTAG_SDD` and *x*, *y*, *z*, etc. are the dimension sizes)
702 (0x02BE)

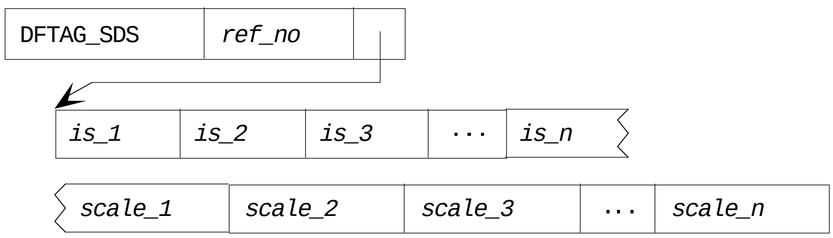


`ref_no` Reference number (16-bit integer)

This tag points to an array of scientific data. The type of the data may be specified by an `DFTAG_NT` included with the SDG. If there is no `DFTAG_NT`, the type of the data is floating-point in standard IEEE 32-bit format. The rank and dimensions must be stored as specified in the corresponding `DFTAG_SDD`. The diagram above shows a 3-dimensional data array.

`DFTAG_SDS`

Scientific data scales
*rank + NTsize0*x + NTsize1*y +NTsize2*z +...* bytes (where *rank* is the number of dimensions, *x*, *y*, *z*, etc. are the dimension sizes, and *NTsize#* are the sizes of each scale NT from the corresponding `DFTAG_SDD`)
703 (0x02BF)



`ref_no` Reference number (16-bit integer)

`is_n` A flag indicating whether a scale exists for the n^{th} dimension (8-bit integer; `0` or `1`)

`scale_n` List of scale values for the n^{th} dimension (type specified in corresponding `DFTAG_SDD`)

This tag points to the scales for the data set. The first *n* bytes indicate whether there is a scale for the corresponding dimension (`1` = yes, `0` = no). This is followed by the scale values for each dimension. The scale consists of a simple series of values where the number of values and their types are specified in the corresponding `DFTAG_SDD`.

`DFTAG_SDL`

Scientific data labels
? bytes
704 (0x02C0)

DFTAG_SDL	*ref_no*	

label_1	*label_2*	...	*label_n*

ref_no Reference number (16-bit integer)

label_n Null terminated ASCII string (any length)

This tag points to a list of labels for the data in each dimension of the data set. Each label is a string terminated by a null byte (0).

`DFTAG_SDU`

Scientific data units
? bytes
705 (0x02C1)

DFTAG_SDU	*ref_no*	

unit_1	*unit_2*	...	*unit_n*

ref_no Reference number (16-bit integer)

unit_n Null terminated ASCII string (any length)

This tag points to a list of strings specifying the units for the data and each dimension of the data set. Each unit's string is terminated by a null byte (0).

`DFTAG_SDF`

Scientific data format
? bytes
706 (0x02C2)

DFTAG_SDF	*ref_no*	

format_1	*format_2*	...	*format_n*

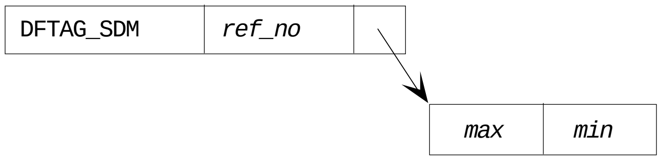
ref_no Reference number (16-bit integer)

format_n Null terminated ASCII string (any length)

This tag points to a list of strings specifying an output format for the data and each dimension of the data set. Each format string is terminated by a null byte (0).

`DFTAG_SDM`

Scientific data max/min
8 bytes
707 (0x02C3)

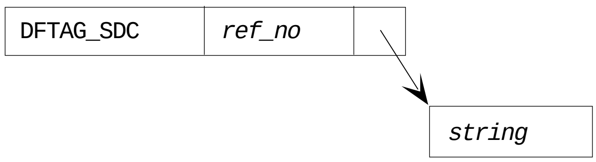


ref_no	Reference number (16-bit integer)
max	Maximum value (type is specified by the data NT in the corresponding `DFTAG_SDD`)
min	Minimum value (type is specified by the data NT in the corresponding `DFTAG_SDD`)

This record contains the maximum and minimum data values in the data set. The type of *max* and *min* are specified by the data NT of the corresponding `DFTAG_SDD`.

`DFTAG_SDC`

Scientific data coordinates
? bytes
708 (0x02C4)

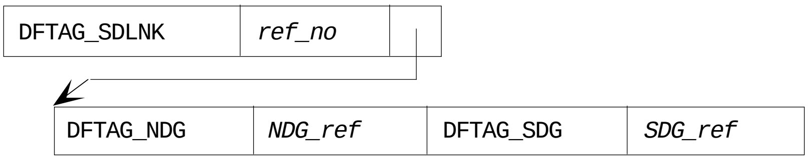


ref_no	Reference number (16-bit integer)
string	Null terminated ASCII string (any length)

This tag points to a string specifying the coordinate system for the data set. The string is terminated by a null byte.

`DFTAG_SDLNK`

Scientific data set link
8 bytes
710 (0x02C6)



ref_no	Reference number (16-bit integer)
`DFTAG_NDG`	NDG tag (16-bit integer)
NDG_ref	NDG reference number (16-bit integer)
`DFTAG_SDG`	SDG tag (16-bit integer)
SDG_ref	SDG reference number (16-bit integer)

The purpose of this tag is to link together an old-style `DFTAG_SDG` and a `DFTAG_NDG` in cases where the NDG contains 32-bit floating point data and is, therefore, equivalent to an old SDG.

See also: Chapter 4, “Sets and Groups”

`DFTAG_CAL`

Calibration information
36 bytes
731 (0x02DB)

DFTAG_CAL	*ref_no*	

cal	*cal_err*	*off*	*off_err*	*data_type*

ref_no	Reference number (16-bit integer)
cal	Calibration factor (64-bit IEEE float)
cal_err	Error in calibration factor (64-bit IEEE float)
off	Calibration offset (64-bit IEEE float)
off_err	Error in calibration offset (64-bit IEEE float)
data_type	Constant representing the effective data type of the calibrated data (32-bit integer)

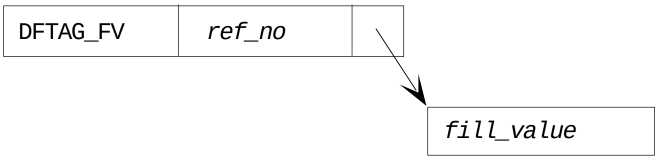
This tag points to a calibration record for the associated `DFTAG_SD`. The data can be calibrated by first multiplying by the *cal* factor, then adding the *off* value. Also included in the record are errors for the calibration factor and offset and a constant indicating the effective data type of the calibrated data. Table 6.6 lists the available *data_type* values.

Table 6.6 Available Calibrated Data Types

Data Type	Description
DFTNT_INT8	Signed 8-bit integer
DFTNT_UINT8	Unsigned 8-bit integer
DFTNT_INT16	Signed 16-bit integer
DFTNT_UINT16	Unsigned 16-bit integer
DFTNT_INT32	Signed 32-bit integer
DFTNT_UINT32	Unsigned 32-bit integer
DFTNT_FLOAT32	32-bit floating point
DFTNT_FLOAT64	64-bit floating point

DFTAG_FV

Fill value
? bytes (size determined by size of data NT in corresponding DFTAG_SDD)
732 (0x02DC)



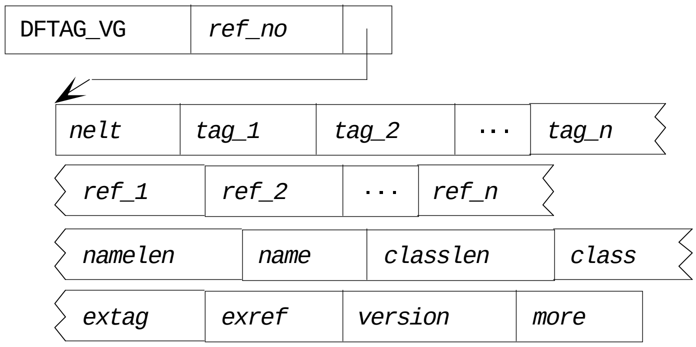
ref_no	Reference number (16-bit integer)
fill_value	Value representing unset data in the corresponding DFTAG_SD (size determined by size of data NT in corresponding DFTAG_SDD)

This tag points to a value which has been used to indicate unset values in the associated DFTAG_SD. The number type of the value (and, therefore, its size) is given in the corresponding DFTAG_SDD.

Vset Tags

`DFTAG_VG`

Vgroup
14 + 4**nelt* + *namelen* + *classlen* bytes
1965 (0x07AD)



ref_no	Reference number (16-bit integer)
nelt	Number of elements in the Vgroup (16-bit integer)
tag_n	Tag of the n^{th} member of the Vgroup (16-bit integer)
ref_n	Reference number of the n^{th} member of the Vgroup (16-bit integer)
namelen	Length of the name field (16-bit integer)
name	Non-null terminated ASCII string (length given by *namelen*)
classlen	Length of the class field (16-bit integer)
class	Non-null terminated ASCII string (length given by *classlen*)
extag	Extension tag (16-bit integer)
exref	Extension reference number (16-bit integer)
version	Version number of `DFTAG_VG` information (16-bit integer)
more	Unused (2 zero bytes)

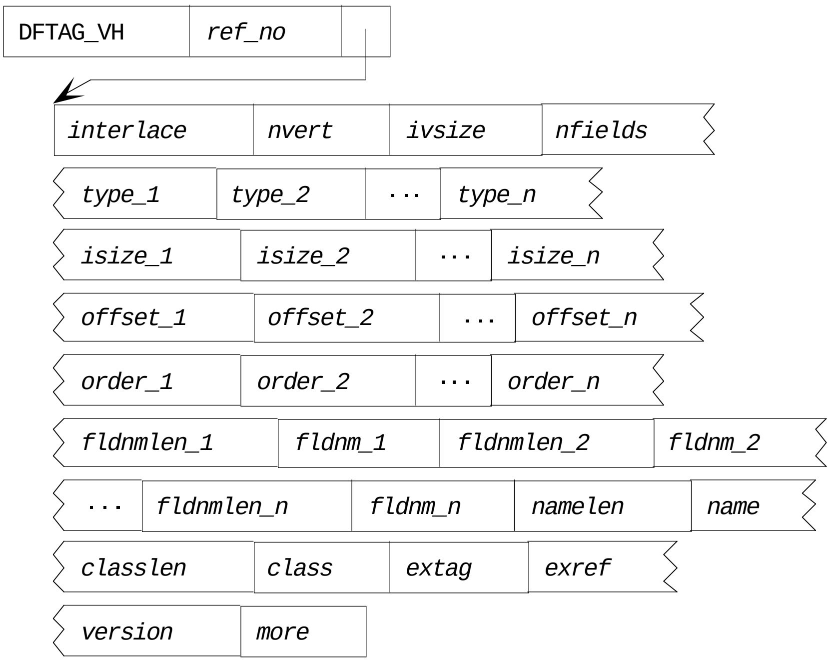
`DFTAG_VG` provides a general-purpose grouping structure which can be used to impose a hierarchical structure on the tags in the group. Any HDF tag may be incorporated into a Vgroup, including other `DFTAG_VG` tags.

See also: “Vsets, Vdatas, and Vgroups” in Chapter 4, “Sets and Groups”
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User’s Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

`DFTAG_VH` Vdata description

$22 + 10*\mathit{nfields} + \sum \mathit{fldnmlen}\ n + \mathit{namelen} + \mathit{classlen}$ bytes

1962 (0x07AA)



ref_no	Reference number (16-bit integer)
interlace	Constant indicating interlace scheme used (16-bit integer)
nvert	Number of entries in Vdata (32-bit integer)
ivsize	Size of one Vdata entry (16-bit integer)
nfields	Number of fields per entry in the Vdata (16-bit integer)
type_n	Constant indicating the data type of the n^{th} field of the Vdata (16-bit integer)
isize_n	Size in bytes of the n^{th} field of the Vdata (16-bit integer)
offset_n	Offset of the n^{th} field within the Vdata (16-bit integer)
order_n	Order of the n^{th} field of the Vdata (16-bit integer)
fldnmlen_n	Length of the n^{th} field name string (16-bit integer)
fldnm_n	Non-null terminated ASCII string (length given by corresponding *fldnmlen_n*)
namelen	Length of the name field (16-bit integer)
name	Non-null terminated ASCII string (length given by *namelen*)
classlen	Length of the class field (16-bit integer)
class	Non-null terminated ASCII string (length given by *classlen*)
extag	Extension tag (16-bit integer)
exref	Extension reference number (16-bit integer)

version	Version number of DFTAG_VH information (16-bit integer)
more	Unused (2 zero bytes)

DFTAG_VH provides all the information necessary to process a DFTAG_VS.

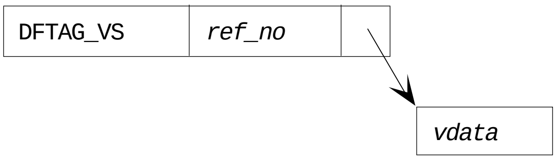
See also: DFTAG_VS (this section)
"Vsets, Vdatas, and Vgroups" in Chapter 4, "Sets and Groups"
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User's Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

DFTAG_VS

Vdata

$$nvert * \sum_{n=1}^{nfields} (isize\_n * order\_n)$$ bytes, where *nvert*, *isize_n*, and *order_n* are specified in the corresponding DFTAG_VH

1963 (0x07AB)



ref_no	Reference number (16-bit integer)
vdata	Data block interpreted according to the corresponding DFTAG_VH ($nvert * \sum_{n=1}^{nfields} (isize\_n * order\_n$ bytes, where *nvert*, *isize_n*, and *order_n* are specified in the corresponding DFTAG_VH)

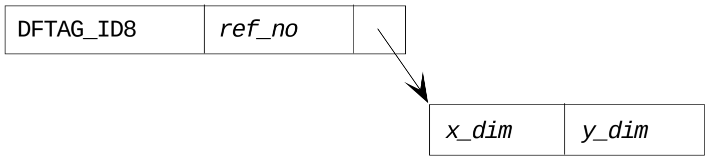
DFTAG_VS contains a block of data which is to be interpreted according to the information in the corresponding DFTAG_VH.

See also: DFTAG_VH (this section)
"Vsets, Vdatas, and Vgroups" in Chapter 4, "Sets and Groups"
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User's Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

Obsolete Tags

`DFTAG_ID8`

Image dimension-8
4 bytes
200 (0x00C8)



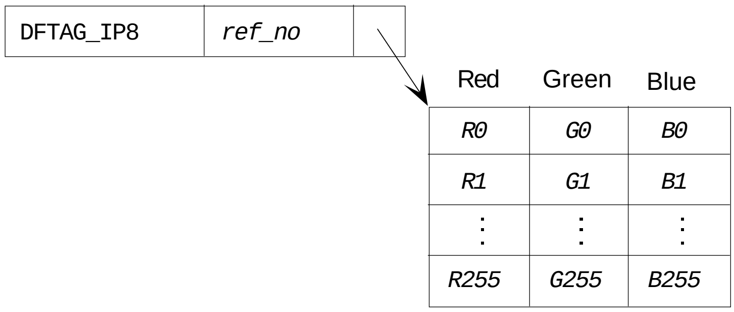
ref_no	Reference number (16-bit integer)
x_dim	Length of x dimension (16-bit integer)
y_dim	Length of y dimension (16-bit integer)

The data for this tag consists of two 16-bit integers representing the width and height of an 8-bit raster image in bytes.

This tag has been superseded by `DFTAG_ID`.

`DFTAG_IP8`

Image palette-8
768 bytes
201 (0x00C9)



ref_no	Reference number (16-bit integer)
Table entries	256 triples of 8-bit integers

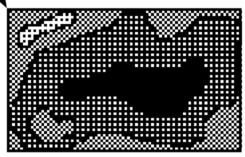
The data for this tag can be thought of as a table of 256 entries, each containing one value for red, green, and blue. The first triple is palette entry 0 and the last is palette entry 255.

This tag has been superseded by `DFTAG_LUT`.

DFTAG_RI8

Raster image-8
*xdim*ydim* bytes (where *xdim* and *ydim* are the dimensions specified in the corresponding DFTAG_ID8)
202 (0x00CA)

DFTAG_RI8	*ref_no*	



ref_no	Reference number (16-bit integer)
Image data	2-dimensional array of 8-bit integers

The data for this tag is a row-wise representation of the elementary 8-bit image data. The data is stored width-first (i.e., row-wise) and is 8 bits per pixel. The first byte of data represents the pixel in the upper-left hand corner of the image.

This tag has been superseded by DFTAG_RI.

DFTAG_CI8

Compressed image-8
? bytes
203 (0x00CB)

DFTAG_CI8	*ref_no*	

compressed_image

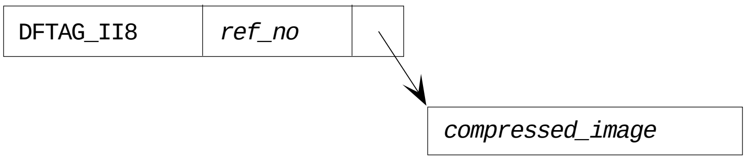
ref_no	Reference number (16-bit integer)
compressed_image	Series of run-length encoded bytes

The data for this tag is a row-wise representation of the elementary 8-bit image data. Each row is compressed using the following run-length encoding where *n* is the lower seven bits of the byte. The high bit indicates whether the following *n* bytes will be reproduced exactly (high bit = 0) or whether the following byte will be reproduced *n* times (high bit = 1). Since DFTAG_CI8 and DFTAG_RI8 are basically interchangeable, it is suggested that you not have a DFTAG_CI8 and a DFTAG_RI8 with the same reference number.

This tag has been superseded by DFTAG_RLE.

`DFTAG_II8`

IMCOMP image-8
? bytes
204 (0x00CC)



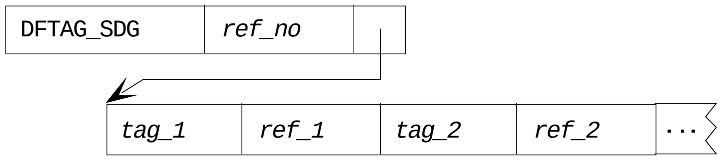
ref_no	Reference number (16-bit integer)
compressed_image	Compressed image data

The data for this tag is a 4:1 compressed 8-bit image, using the IMCOMP compression scheme.

This tag has been superseded by `DFTAG_IMC`.

DFTAG_SDG

Scientific data group
*n**4 bytes (where *n* is the number of data objects in the group)
700 (0x02BC)



ref_no Reference number (16-bit integer)

tag_n Tag number of n^{th} member of the group (16-bit integer)

ref_n Reference number of n^{th} member of the group (16-bit integer)

The SDG data element contains a list of tag/refs that define a scientific data set. All of the members of the group provide information required to correctly interpret and display the data. Application programs that deal with SDGs should read all of the elements of an SDG and process those which it can use. Even if an application cannot process all of the objects, the objects that it can understand will be usable.

Table 6.7 lists the tags that may appear in an SDG.

Table 6.7 Available SDG Tags

Tag	Description
DFTAG_SDD	Scientific data dimension record (rank and dimensions)
DFTAG_SD	Scientific data
DFTAG_SDS	Scales
DFTAG_SDL	Labels
DFTAG_SDU	Units
DFTAG_SDF	Formats
DFTAG_SDM	Maximum and minimum values
DFTAG_SDC	Coordinate system
DFTAG_SDT	Transposition (obsolete)
DFTAG_SDLNK	Link to new DFTAG_NDG

Example
DFTAG_SDD, DFTAG_SD, DFTAG_SDM
Assume that a dimension record, scientific data, and the maximum and minimum values of the data are required to read and interpret a particular data set. These data objects can be associated in an SDG so that an application can read the rank and dimensions from the dimension record and then read the data array. If the application needs the maximum and minimum values, it will read them as well.

This tag has been superseded by DFTAG_NDG.

See also: Chapter 4, “Sets and Groups”

`DFTAG_SDT`

Scientific data transpose
0 bytes
709 (0x02C5)

DFTAG_SDT	*ref_no*	

ref_no Reference number (16-bit integer)

The presence of this tag in a group indicates that the data pointed to by the corresponding `DFTAG_SD` is in column-major order, instead of the default row-major order. No data is associated with this tag.

This tag is no longer written by the HDF library. When it is encountered in an old file, it is interpreted as originally intended.

Chapter 7 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 7.1.

Table 7.1 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], 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.

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.

Organization of Source Files

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

- Header files
- Source code files
- A makefile

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`, `hdfi.h`[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 general purpose 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.

---

2 In earlier implementations of HDF, these files were called `df.h` and `dfi.h`. Starting with HDF Version 3.2, the general purpose layer of HDF was completely rewritten and all routine names were changed from `df*` to `hdf*`.

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:

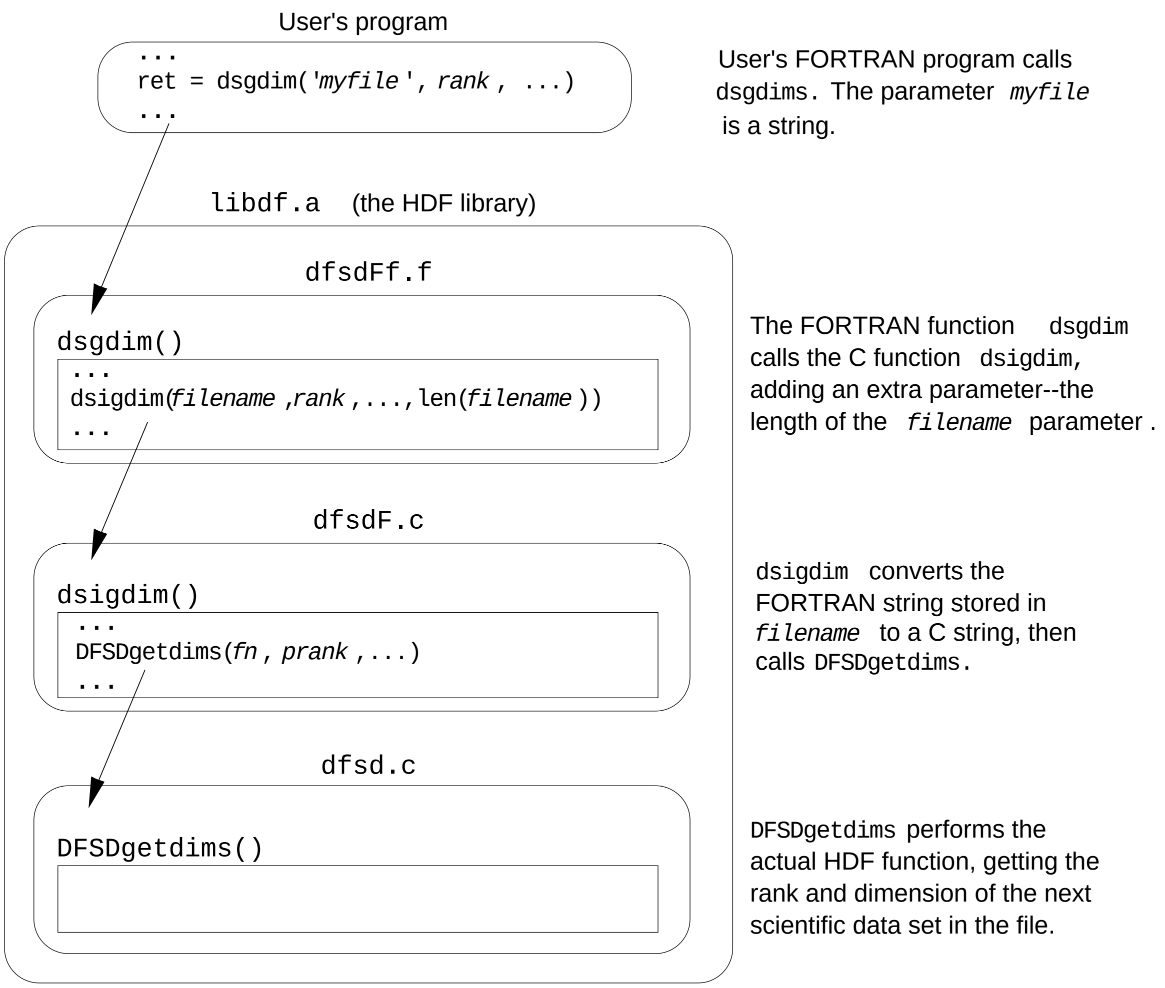
- UNICOS FORTRAN stores strings in a structure called `_fcd` (FORTRAN character descriptor). `_fcdtocp` is a built-in UNICOS function that returns the string’s address. (Since UNICOS provides this function, HDF omits the corresponding macro definition on UNICOS systems.)
- VMS FORTRAN uses a string descriptor 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 results in the following sequences of actions:

1. 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.
2. The FORTRAN filter then calls a C routine, to which it passes all parameters from the initial call the string's length.
3. 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.
4. This C routine then calls the HDF C routine that performs the actual work.

This process is illustrated in Figure 7.1

Figure 7.1. 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]

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.

3 `__STDC__` 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.

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, that 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`.

Appendix A Tags and Extended Tag Labels

The tables in this appendix lists all of the NCSA-supported HDF tags and the labels used to identify extended tags.

Tags

Table A.1 lists all the NCSA-supported HDF tags with the following information:

Tag	The tag itself
Tag number	The regular tag number in decimal (top) and hexadecimal (bottom)
Extended tag number	The extended tag number used with linked blocks and external data elements in decimal and (hexadecimal)
Full name	The tag name, a descriptive English phrase
Section	The section of Chapter 6, “Tag Specifications,” in which the tag is discussed

Table A.1 NCSA-supported HDF Tags

Tag	Number	Extended Number	Full Name	Section
DFTAG_AR	312 0x0138		Aspect ratio	Raster Image Tags
DFTAG_CAL	731 0x02DB		Calibration information	Scientific Data Set Tags
DFTAG_CCN	310 0x0136		Color correction	Raster Image Tags
DFTAG_CFM	311 0x0137		Color format	Raster Image Tags
DFTAG_CI8	203 0x00CB		Compressed image-8	Obsolete Tags
DFTAG_DIA	105 0x0069		Data identifier annotation	Annotation Tags

Table A.1 NCSA-supported HDF Tags *(Continued)*

Tag	Number	Extended Number	Full Name	Section
DFTAG_DIL	104 0x0068		Data identifier label	Annotation Tags
DFTAG_DRAW	400 0x0190		Draw	Composite Image Tags
DFTAG_FD	101 0x0065		File description	Annotation Tags
DFTAG_FID	100 0x0064		File identifier	Annotation Tags
DFTAG_FV	732 0x02DC		Fill value	Scientific Data Set Tags
DFTAG_GREYJPEG	14 0x000E		8-bit JPEG compression information	Compression Tags
DFTAG_ID	300 0x012C		Image dimension	Raster Image Tags
DFTAG_ID8	200 0x00C8		Image dimension-8	Obsolete Tags
DFTAG_II8	204 0x00CC		IMCOMP image-8	Obsolete Tags
DFTAG_IMC	12 0x000C		IMCOMP compressed data	Compression Tags
DFTAG_IP8	201 0x00C9		Image palette-8	Obsolete Tags
DFTAG_JPEG	13 0x000D		24-bit JPEG compression information	Compression Tags
DFTAG_LD	307 0x0133		LUT dimension	Raster Image Tags
DFTAG_LUT	301 0x012D		Lookup table	Raster Image Tags
DFTAG_MA	309 0x0135		Matte channel	Raster Image Tags
DFTAG_MD	308 0x0134		Matte channel dimension	Raster Image Tags
DFTAG_MT	107 0x006B		Machine type	Utility Tags
DFTAG_NDG	720 0x02D0		Numeric data group	Scientific Data Set Tags
DFTAG_NT	106 0x006A		Number type	Utility Tags
DFTAG_NULL	1 0x0001		No data	Utility Tags
DFTAG_RI	302 0x012E	16686 0x412E	Raster image	Raster Image Tags
DFTAG_RI8	202 0x00CA		Raster image-8	Obsolete Tags

Table A.1 **NCSA-supported HDF Tags** *(Continued)*

Tag	Number	Extended Number	Full Name	Section
DFTAG_RIG	306 0x0132		Raster image group	Raster Image Tags
DFTAG_RLE	11 0x000B		Run length encoded data	Compression Tags
DFTAG_SD	702 0x02BE	17086 0x42BE	Scientific data	Scientific Data Set Tags
DFTAG_SDC	708 0x02C4		Scientific data coordinates	Scientific Data Set Tags
DFTAG_SDD	701 0x02BD		Scientific data dimension record	Scientific Data Set Tags
DFTAG_SDF	706 0x02C2		Scientific data format	Scientific Data Set Tags
DFTAG_SDG	700 0x02BC		Scientific data group	Obsolete Tags
DFTAG_SDL	704 0x02C0		Scientific data labels	Scientific Data Set Tags
DFTAG_SDLNK	710 0x02C6		Scientific data set link	Scientific Data Set Tags
DFTAG_SDM	707 0x02C3		Scientific data max/min	Scientific Data Set Tags
DFTAG_SDS	703 0x02BF		Scientific data scales	Scientific Data Set Tags
DFTAG_SDT	709 0x02C5		Scientific data transpose	Obsolete Tags
DFTAG_SDU	705 0x02C1		Scientific data units	Scientific Data Set Tags
DFTAG_T105	603 0x25B		Tektronix 4105	Vector Image Tags
DFTAG_T14	602 0x25A		Tektronix 4014	Vector Image Tags
DFTAG_TD	103 0x0067		Tag description	Annotation Tags
DFTAG_TID	102 0x0066		Tag identifier	Annotation Tags
DFTAG_VERSION	30 0x001E		Library version number	Utility Tags
DFTAG_VG	1965 0x07AD		Vgroup	Vset Tags
DFTAG_VH	1962 0x07AA		Vdata description	Vset Tags
DFTAG_VS	1963 0x07AB	18347 0x47AB	Vdata	Vset Tags
DFTAG_XYP	500 0x01F4		X-Y position	Composite Image Tags

Extended Tag Labels

Table A.2 lists labels used to identify HDF extended tags. The table includes the following information:

Extended tag label
: The label, which appears as the first element of the extended tag description record

Physical storage method
: The alternative storage method indicated by the label

Table A.2 Extended Tag Labels

Extended Tag Label	Physical Storage Method
`EXT_EXTERN`	External file element
`EXT_LINKED`	Linked block element