Introducing the Getting Started with HDF Manual

I About this Manual

Getting Started with HDF is designed to serve as a basic introduction to the data models, interfaces, and utilities supported by HDF. It contains fundamental descriptions, definitions, and simple examples that demonstrate how to begin using the HDF library and its utilities. To use this manual most effectively:

See:		For:
Chapter 1:	Introducing HDF	An answer to the question "what is HDF" and an outline of what to expect from HDF as a file format.
Chapter 2:	The HDF Data Models	A brief introduction to each of the data models currently supported by HDF.
Chapter 3:	The Bare Necessities for Reading and Writing Data	Instructions and examples demonstrating how to perform simple read and write operations using each application interface.
Chapter 4:	HDF Utilities and Software Applications	A list of utilities and tools available for working with HDF files.

II Conventions

The only convention used in this manual is that of the typeface. As a result, you will find the following three fonts used throughout these pages:

1. `Courier` is used to identify all code examples.
2. **Times/Bold** is used to identify the name of a function call or utility.
3. ***Times/Bold/Italic*** is used to identify a word whose definition is included in the glossary.

III Using Other HDF Documentation

At this point in time, several HDF documents are actively undergoing review and reorganization. These documents include the *HDF Specification*, *NCSA HDF Calling Interfaces and Utilities Manual*, and the *NCSA HDF Vset Manual*. Under the planned restructuring, the *HDF Specification* will undergo little change, however, the *NCSA HDF Calling Interfaces and Utilities Manual* will be combined with the *NCSA HDF Vset Manual* to produce three new documents.

These new HDF documentation will include *Getting Started with HDF*, an *HDF User's Guide*, and an *HDF Reference Manual*. Of these documents, only *Getting Started with HDF* is complete. Until the user's guide and reference manual are available:

If you need to:	**See the:**
Obtain a current version of HDF	NCSA ftp server (ftp.ncsa.uiuc.edu).
Install HDF	Current *README* on the ftp server.
Get quick help on an HDF function call	HDF man pages.
Learn basic HDF concepts	*Getting Started with HDF.*
Consult the *HDF Specification*	*HDF Specification.*
Consult the *HDF User's Guide*	*NCSA HDF Calling Interfaces and Utilities.*
Consult the *HDF Reference Manual*	*NCSA HDF Calling Interfaces and Utilities.*

IV Trademark Acknowledgments

Alliant is a registered trademark and CONCENTRIX is a trademark of Alliant Computer Systems Corporation. Macintosh and Macintosh II are trademarks of Apple Computer Inc. UNIX is a registered trademark of AT&T. CRAY and UNICOS are registered trademarks and CRAY-2 and CFT77 are trademarks of Cray Research Inc. IBM PC is a registered trademark of International Business Machines Corporation. MS-DOS is a registered trademark of Microsoft Corporation. Sun is a registered trademark and Sun Workstation and Sun System 3 are trademarks of Sun Microsystems Inc.

V Copyright Restrictions

NCSA HDF source code is in the public domain. Specifically, we give to the public domain all rights for future licensing, resale, and publication. We ask, but do not require, that the following message be included in all derived works:

> **Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.**

VI Disclaimer

THE UNIVERSITY OF ILLINOIS GIVES NOWARRANTY, EXPRESS OR IMPLIED, FOR THE SOFTWARE AND/OR DOCUMENTATION PROVIDED, INCLUDING, WITHOUT LIMITATION, WARRANTY OF MERCHANTIBILITY AND WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE.

Getting Started with HDF

Draft, Version 3.2
May 1993

University of Illinois at Urbana-Champagin

Table of Contents

Chapter 1: Introducing HDF

1.1 HDF at a Glance 1

1.2 What Is HDF? 2

1.3 Why was HDF created? 3

1.4 Why Use HDF? 3

- HDF is a versatile file format 4
- HDF is an extensible file format 4
- HDF files are self-describing 5
- HDF is a flexible file format 5
- HDF is a portable file format 7
- HDF is a FORTRAN and C callable library 7
- HDF is available in the public domain 8

Chapter 2: The HDF Data Models

2.1 Introducing the Data Models Supported by HDF 10

2.2 The 8-bit Raster Model 10

2.3 The Palette Model 12

2.4 The 24-bit Raster Model 13

2.5 The Scientific Data Model 14

2.6 The Annotation Model 15

2.7 The Vdata Model 16

Chapter 3: The Bare Necessities for Reading and Writing Data

3.1 Introducing the Application Interfaces Supported by HDF 18

3.2 HDF and the 8-bit Raster Interface 19
- Writing an 8-bit Raster Image to an HDF File 19
- Reading an 8-bit Raster Image from an HDF File 21

3.3 HDF and the Palette Interface 24
- Writing a Palette to an HDF File 24
- Reading a Palette from an HDF File 26

3.4 HDF and the 24-bit Raster Image Interface 27
- Writing a 24-bit Image to an HDF File 27
- Reading a 24-bit Raster Image from an HDF File 30

3.5 HDF and the Scientific Data Interface 33
- Writing Scientific Data to an HDF File 33
- Reading Scientific Data from an HDF File 36

3.6 HDF and the Annotation Interface 38
- Writing a File Description 38
- Reading a File Description 40

Chapter 4: HDF Utilities and Software Applications

4.1 Introducing the HDF Utilities 44

4.2 The HDF Inquiry Utilities 45
- hdfls 45
- hdfed 46
- hdfseq/hdfrseq 46

4.3 The Raw Data to HDF Conversion Utilities 46
- r8tohdf 47
- paltohdf 47
- r24hdf8 47
- fp2hdf 48

4.4 The HDF to Raw Data Conversion Utilities 48
- hdftor8 49
- hdftopal 49

4.5 The HDF to HDF Conversion Utilities 49
- ristosds 50
- hdf24to8 50
- fp2hdf 50

4.6 The HDF Compression Utilities 50
- hdfpack 51
- hdfcomp 51

4.7 Introducing Applications that Support HDF 51

Chapter 1

Introducing HDF

1.0 Chapter Overview

Welcome to HDF! This chapter will introduce you the very simple, yet fundamental, HDF concepts. It begins by introducing you to HDF with a brief history of its development, follows with a description of HDF and its various layers, and concludes with a discussion outlining some of the benefits of HDF.

If you are already familiar with HDF and how it works, you can skip ahead to Chapter 2 to begin exploring the different HDF data models. If you've worked with HDF in the past but some of the basic concepts are still a bit fuzzy, this chapter will serve as a good review. If are you new to HDF, this chapter will introduce you to the basic capabilities of HDF.

1.1 HDF at a Glance

The HDF project began in 1988 at the National Center for Supercomputing Applications. The HDF development effort originated in response to a need for storing different types of scientific data generated on different kinds of computers. Its primary goal; to help NCSA scientists share their data regardless of its origin.

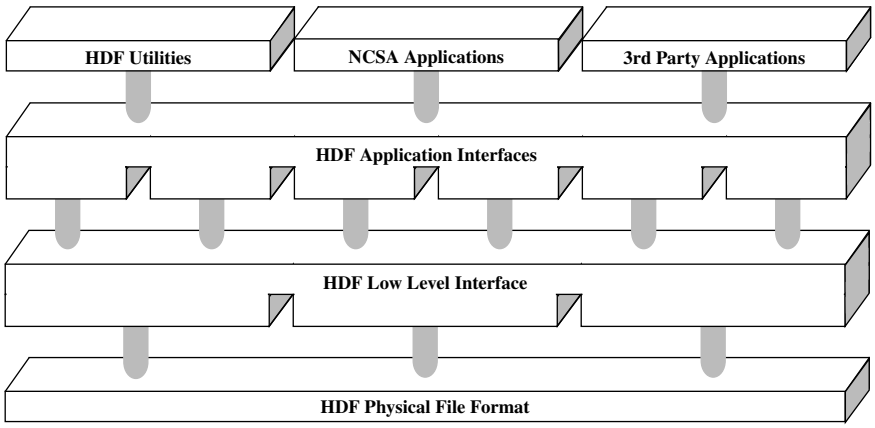
Because HDF is a flexible file format, its use quickly spread beyond NCSA. Commercial vendors, scientists from national laboratories, and many universities began using HDF early in its development cycle. Due to their influence, HDF quickly broadened its scope to include the functionality offered by more popular file formats. As a result, HDF is now used in various computing environments for many different applications.

Although it is gaining popularity, HDF is still only one of many standard file formats available in the scientific community. Achieving interoperability among different standards is still a particularly difficult problem. At first, HDF provided simple utilities for converting files between HDF and other formats. Today however, in addition to adding other features, the HDF development team is exploring new methods for achieving a higher level of interoperability between HDF and other file formats.

1.2 What Is HDF?

Although this question seems simple, it is difficult to answer because it can be addressed in a number of different ways. At its lowest level, HDF is a physical file format. At its highest level, HDF is a collection of utilities and large-scale applications. In between, HDF is a library that provides both a high-level application and a low-level data interface (See FIGURE 1.2a). In practical terms, HDF can be thought of as three interface layers built upon a physical file format.

FIGURE 1.2a **The HDF physical file format supports three levels of interaction.**



Like other file formats, the HDF file format is a predefined method of storing different types of data. The benefit of the HDF file format is that you can read or write different types of data to the same physical file. How you create or access an HDF file is basically up to you, however, it is highly recommended that you use one of the three officially recognized interface layers to do so.

The first interface layer, or the ***low level interface*** is the layer of HDF you'll work with if you wish to work with HDF as a software developer. It contains the source code for things like file I/O, error handling, memory management and physical storage. It's essentially a software tool-kit for skilled programmers who wish to make HDF do something more than what is officially available. If you are a scientist or engineer, it is more likely that you'll be working through the HDF ***application interfaces***.

The HDF application interfaces include six independent modules that are specifically designed to simplify the process of storing and accessing one type of data. If you are primarily interested in reading and writing data to HDF files, you will spend most of your time working with the application interface. Although each interface module requires some programming, all the low level details can be ignored. In most cases, all you have to do is make the right function call at the right time, and the interface will take care of the rest.

At its highest level, HDF includes utilities, NCSA applications, and a variety of other third party applications. Although the HDF utilities are automatically included in the official ***HDF source distribution***, the HDF applications are only available as separate programs. Official NCSA applications and those applications contributed by HDF users are freely available on the NCSA ***ftp server***. In addition, several software vendors also support HDF. Check with your local distributor for more information on commercial packages.

1.3 Why was HDF created?

The HDF file format was created to help today's scientists work in complex computing environments. Unlike the scientific laboratories of yesterday where a single scientist used a single computer to generate and analyze data, today's labs are typically filled with different scientists using many different data sources. These data sources often include scientists at other labs, networked computers, and sophisticated electronic instruments capable of generating vast quantities of data. This growing dependance on multiple data sources introduced the need for a more flexible file format. HDF met this need by allowing scientists to share data files across different computers, access the same data files using different software applications, and store different types of data in the same file.

1.4 Why Use HDF?

Although the world of electronic storage offers many different data formats, there are several reasons why many computational scientists use HDF:

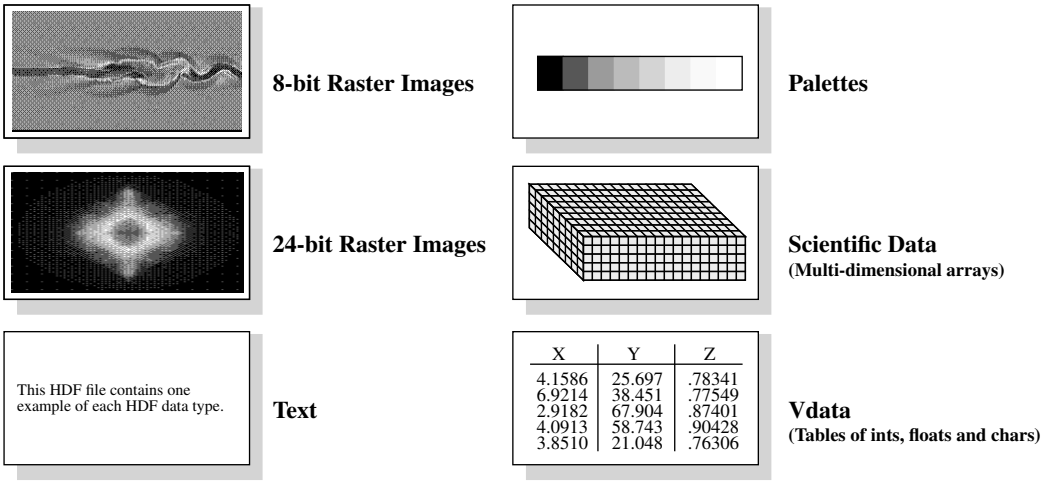
- ***HDF is a versatile file format.*** HDF supports six different data models. Each data model defines a specific type of data and provides a convenient interface for reading, writing, and organizing a unique set of data elements. Because each data model is modularly designed, you need only use the data model appropriate for your immediate needs.
- ***HDF is an extensible file format***. Although HDF currently supports six different data models, there will almost certainly be a need to incorporate new types of data in the near future. HDF is designed to easily accommodate new data models, regardless of whether they are added by the HDF development team or by HDF users.
- ***HDF is self-describing.*** Inside every HDF file there is a data index which identifies and describes the contents of the file. This makes HDF a convenient file format for sharing data because everything you (or the scientists you are working with) need to know about the data can be stored in the file along with the data. HDF eliminates the need for scribbled reminders describing how and what data is stored in a file.
- ***HDF is a flexible file format.*** With HDF, you can group sets of related data together and then access them as a group or as individual objects. You can even group existing sets together into larger sets and still access each subset individually. As you work with HDF, you will find that it can make storing and organizing data extremely convenient.
- ***HDF is a portable file format.*** HDF files are portable in the sense that they can be shared across platforms. For example, an HDF file created on one computer (a Cray supercomputer for example) can be used on another system (a Macintosh, perhaps) without modification.
- ***HDF is a portable library.*** Once the HDF library is compiled for your machine, its functions can be called from inside your FORTRAN or C programs. By using the HDF library, most of the programming necessary for reading or writing HDF files is already done for you.
- ***HDF is available in the public domain.***

HDF is a versatile file format.

If you are like most scientists, it is fairly likely that you will encounter many different types of data. This data often includes programs and their output, data collected from the real world, and data received from other scientists. In order to keep all this information organized, you'll need some way of storing and organizing different types of data. With HDF, you can store a variety of data types in the same file (See FIGURE 1.4a). Once the data is in a file, you can add meaningful annotations that describe the origin, or other characteristics of the data.

To demonstrate the versatility of HDF, consider an example. Assume an HDF file contains a program and its floating-point output. Suppose you wish to view the data as a series of images in an animation sequence. The first step toward an animation is to convert the floating point data into 8-bit integers. Once the data is converted, you must create and order the series of images. Finally, you must assign an appropriate color-look up table. Ordinarily the results of each step are stored in separate files, however, with HDF, the source code, its floating point output, the 8-bit images, and color look-up table can all be stored in the same file. As you can see, HDF is truly versatile in the sense that it can store any number (limited by available memory, of course) and variety of data types in the same file.

FIGURE 1.4a **HDF can stores several different types of data in the same file.**



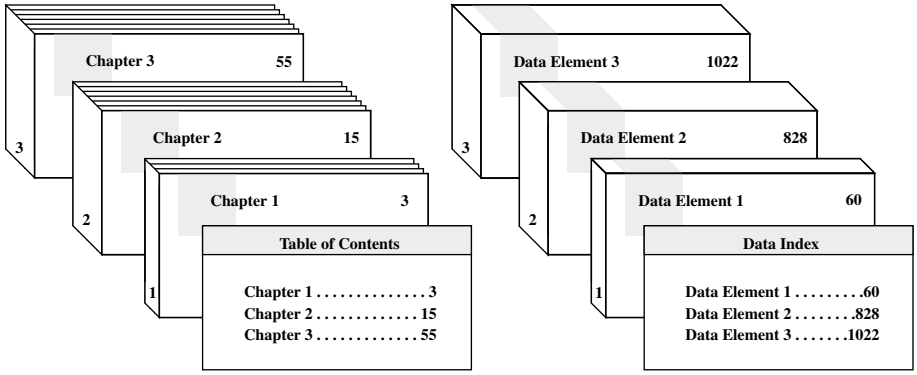
HDF is an extensible file format.

HDF is specifically designed to easily incorporate new data models. Even though there are already a great number of recognized data types, there will almost certainly be a need for new ones. The HDF development team is already considering the best way to create new data types for things like voice and video. If you need to work with a data type that HDF does not currently support, it is possible to create your own. However, only those people who have access to your code will recognize your new data type. You can learn more about creating new data types in the HDF Users Guide.

HDF files are self-describing.

HDF files are considered self-describing because inside every file is a list describing the contents of the file. Perhaps the best way to think of an HDF file is to think of a multi-chapter book with a table of contents (See FIGURE 1.4b). HDF files are “data books” in the sense that each chapter contains a different type of data or ***data element***. Just as a book uses a table of contents to list its chapters, an HDF file uses a “data index” to list its data elements. Consequently, the “table of contents” for an HDF file can be though of as a list of all the data elements contained in the file. Although keeping an active data index may seem a bit complicated, in most cases HDF will maintain control over how and where data elements are added. Your job is simply to identify what data elements you need to add to which files.

FIGURE 1.4b **The building blocks of an HDF file are similar to those found in a conventional book.**

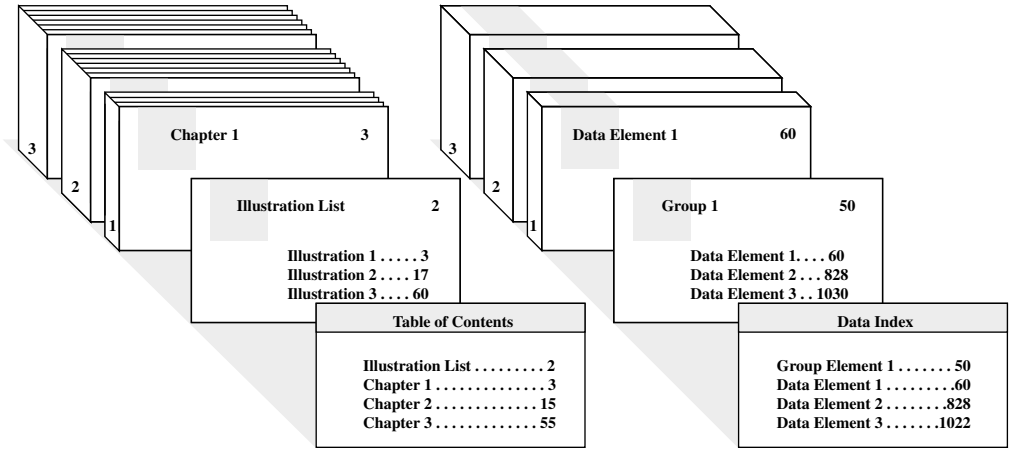


HDF is a flexible file format.

What makes HDF a flexible file format is its ability to access data elements regardless of how they are stored. With HDF, you can store data elements one-to-a-file, add multiple data elements to a file, or group related data elements together in the same file. The level of organization required for a particular collection of data is typically determined by the nature of the data, and your level of familiarity with HDF.

To better understand how data elements are grouped in HDF files, recall that every HDF file can be thought of as a multi-chapter book with a table of contents. The process of grouping data elements is similar to the process of adding a list of related items to a book. Say for example, you wish to add a list of illustrations to a book. First you must identify the illustrations you wish to include, then you must compile the list and insert it into the book. Finally, you must create an entry for the list in the table of contents. HDF files perform similar tasks when asked to group data elements. First the group members are identified, then they are compiled into a list and added to the file. Finally, an appropriate entry for the group is added to the data index (See FIGURE 1.4c on page 6).

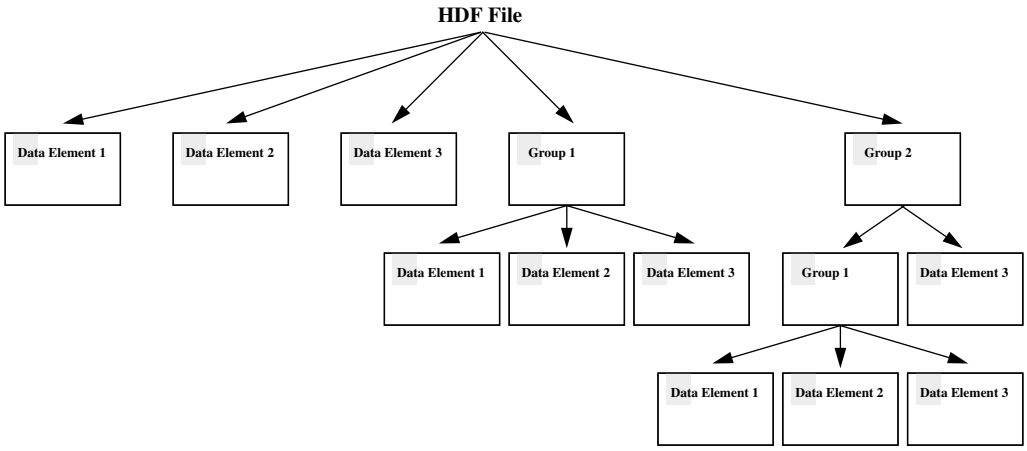
FIGURE 1.4c **Grouping a set of data elements in an HDF file is similar to adding a list to a book.**



Once a ***group element*** appears in the data index, HDF treats it just like an individual data element. In other words, groups can be added to other groups just as if they were single data elements. For example, grouping Data Element 3 with Group Element 1 will form a new group and add another group reference to the data index. The second group will not interfere with the first group or any of the other data elements contained in the file. It will simply add another group to the file.

By intelligently grouping data elements, you can create extremely complex or very simple organizational structures. Conceptually speaking, each data group represents a node in a data tree. Consider the two group elements described in the previous example. The first group contained three individual elements, and the second contained one group and one individual element. This particular arrangement translates into a three-tier data tree with each group branching into the next lowest tier (See FIGURE 1.4d).

FIGURE 1.4d **The organizational structure of an HDF file can be represented by a data tree..**



As you work with HDF, you will discover that HDF will most likely create the appropriate data structures for you. However, for more complex structures you may be called upon to create your own. You can learn more about creating data hierarchies in the HDF User's Guide.

HDF is a portable file format.

The official ***HDF distribution*** contains all the source code you will need to create an HDF library for your machine. The HDF distribution provides C code for a variety of different computers. At this point, check the list below to see if your machine is officially supported.

Convex (UNIX)	**IBM PC** (MS-DOS/Windows)	**NeXT** (NeXTStep)
Cray X-MP/2 (UNICOS)	**IBM RT** (AIX)	**Silicon Graphics** (Irix)
DecStation (Ultrix)	**Macintosh** (MacOS)	**Sun 3, Sun 386, Sparc** (SunOS)
HP 9000 (HPUX)	**IBM RS6000** (AIX)	**Vax** (VMS, Ultrix)

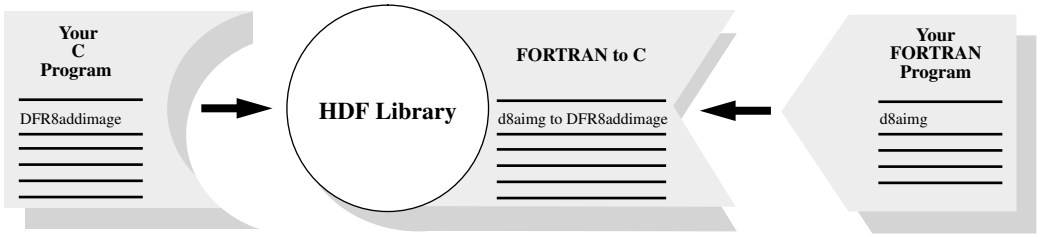
If your computer does not appear in this list, there is still some hope. NCSA may have ported HDF to your machine since the publication of this document. There is also a chance that a user has contributed code for your machine. Although contributed code is not included in the official release, it is available on the NCSA ftp server in the HDF/contrib directory. Check the current HDF README for more details.

HDF is a FORTRAN and C callable library.

The ***HDF library***, like other software libraries, provides object code for a collection of frequently used functions. Unlike other software libraries however, the HDF library is accessible from both C and FORTRAN programs. By using its own FORTRAN to C converter, the HDF library can support HDF function calls made from both C and FORTRAN programs.

Even though the HDF library is accessible to both C and FORTRAN programmers, some FORTRAN compilers, such as UNICOS FORTRAN (CFT77) on the CRAY-2 system, only accept function names that are eight or fewer characters. To avoid compiler errors, HDF provides two names for each function; one for use with the C library, and a shorter version for use with the FORTRAN library. For example, **d8aimg** is the FORTRAN equivalent for **DFR8addimage**. Both function calls will add an 8-bit raster image to an HDF file (See FIGURE 1.4e).

FIGURE 1.4e **HDF uses a function call converter to interface FORTRAN calls to its C library.**



Before you can use ***HDF function calls*** in your C or FORTRAN programs, the HDF library must be installed on the machine you plan to use. If at some point in the past HDF was compiled for your machine, all you need to do is check to see that it is current. To do so, you can ask your system administrator or check its ***version number*** and compare it with the current version available on the ftp server. If you do not have access to a machine with a usable HDF library, you (or your system administrator) will need to install it before you attempt to link your code to the HDF library. Complete installation instructions are available on the NCSA ftp server.

HDF is available in the public domain.

HDF source code is made available in the public domain. Specifically, NCSA gives to the public domain all rights for future licensing, resale, and publication. NCSA asks, but does not require, that the following message be included in all derived works:

> ***Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign, in collaboration with the Information Technology Institute of Singapore.***

If you want to see more software like HDF, please send us a letter (email or U.S. mail), telling us what you are doing with HDF. We'd like to know what you are working on, so please include an abstract or short description of your work. We'd also like to know how HDF has helped you, for example, by increasing your productivity or allowing you to do things you could not do before. A bibliography of your work would be extremely helpful.

We encourage you to cite the use of HDF in your publications. This is a new kind of shareware. You share your science and successes with us, and we can get resources to share more software like HDF with you. Please send your letters to:

U.S. mail:

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

Electronic mail:

softdev@ncsa.uiuc.edu
softdev@ncsavms.bitnet

Although we try hard to test our code, errors are an unavoidable part of software development. If you encounter difficulty with HDF and you are convinced that it is a bug, feel free to report it to us at the addresses listed above. If you are experiencing difficulty and you are not sure what is causing the problem, you can get help by contacting the HDF support staff at NCSA. Our support staff will try to respond as quickly as possible. If they are unable to respond immediately, please be patient.

Chapter 2

The HDF Data Models

2.0 Chapter Overview

The HDF library currently supports six different data models, where each data model represents a framework for accessing a different type of data and its associated information. In a sense, each data model can be thought of as a set of tools for customizing the contents an HDF file. Although there is some overlap among tool sets, in most cases each set of tools is limited to operating on data from one data model. As a result of HDF's "tool set modularity," you need only familiarize yourself with the data model specific to your needs.

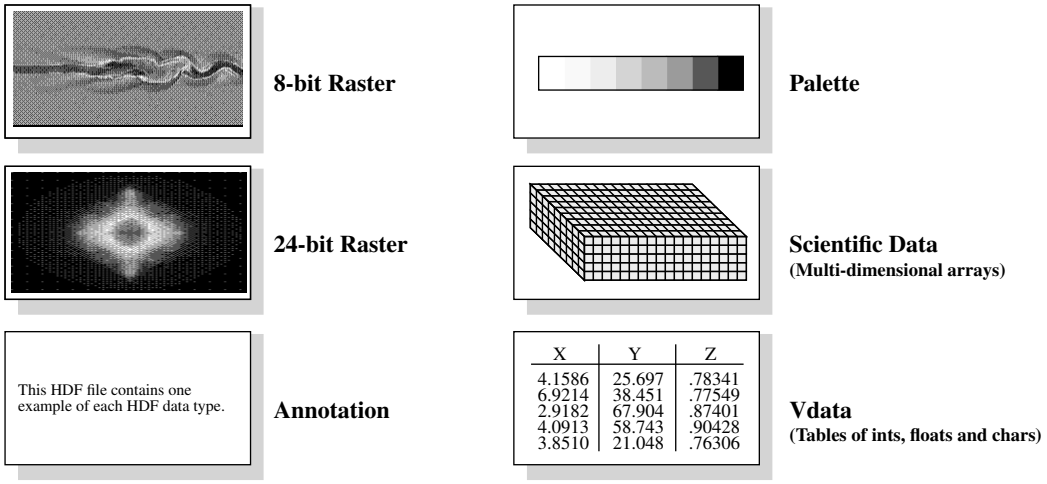
In Chapter 3, you will learn to think of each data model in the following way:

- **The *8-bit raster model*** stores and retrieves 8-bit raster images, their dimensions, and palettes.
- **The *palette model*** stores and retrieves 8-bit palettes outside the 8-bit raster model.
- **The *24-bit raster model*** stores and retrieves 24-bit images and their dimensions.
- **The *scientific data model*** stores and retrieves multi-dimensional arrays of integer or floating-point numbers, their dimensions, number type, and attributes.
- **The *annotation model*** stores and retrieves the text strings used to describe a file or any of the data elements it contains.
- **The *virtual data model*** stores and retrieves multi-variate data stored as records in a table.

2.1 Introducing the Data Models Supported by HDF

The HDF library currently supports six different data models (See FIGURE 2.1a). In practice, a ***data model*** is a framework or data structure used by HDF to store a particular type of data along with its associated information. For example, the Vdata model uses tables to store records containing fields of different number types.

FIGURE 2.1a **The HDF library supports six different data models.**



HDF data models are designed to support only those data elements which are applicable to the group as a whole. In other words, data models are limited to data elements which "make sense" in the context of the group. The 24-bit raster model, for example, will not support palettes or three-dimensional arrays because neither of them are necessary for 24-bit imaging operations.

Recognized groups of data elements are called ***HDF data sets***. All sets have a minimum number of required elements, and a maximum number of optional elements. A raster image set, for example, must contain both a 2-dimensional array of 8-bit numbers and a dimension record, however adding a palette is optional. In the case of 8-bit raster data, HDF limits each set to one 8-bit image array, one dimension record, and one palette. If a raster image set contains all three elements, the set is considered complete.

In the following sections you will learn more about each data model, the type of data it supports, and its acceptable data elements.

2.2 The 8-bit Raster Model

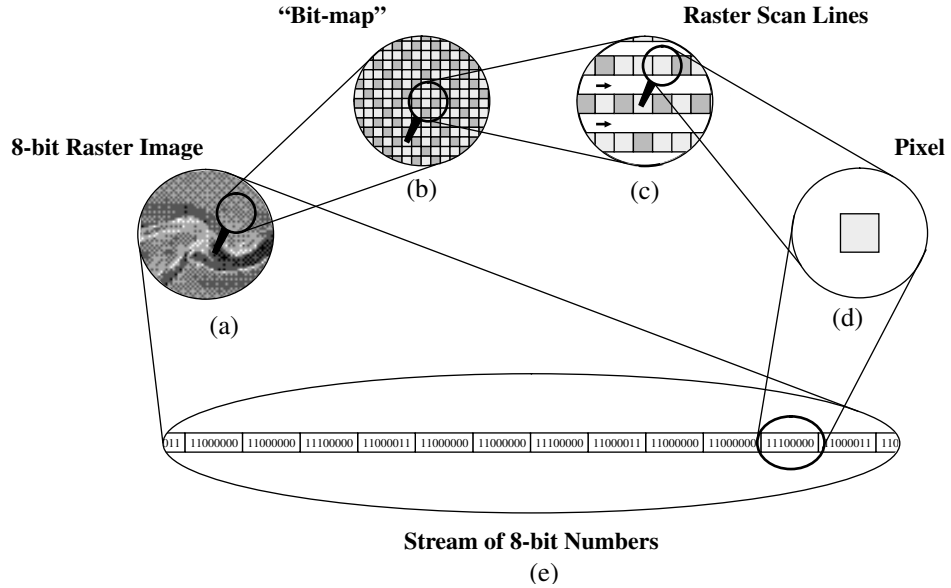
In the days when computers were limited to monochrome displays, determining the value of a pixel required interpreting one bit of information. If the bit was "on" the pixel was white, if the bit was "off" the pixel was black. This simple one-to-one mapping strategy gave rise to the term ***bit-map***. Today the term bit-map is often used to describes any image whose colors or grayscale intensities are stored as n-bits per pixel.

Today, the terms “bit-map” and “raster” are used almost interchangeably. However, the phrase ***raster image*** comes from the application of television technology to bit-map imaging. In a standard raster display, images are scanned out sequentially one line at a time from left to right, top to bottom. An entire raster image is thus formed by scanning a set of horizontal lines each as long as the image is wide.

Although televisions are limited to single image of a standard size, computer monitors are most often used to display multiple images of varying sizes. Consequently, the dimensions for each image must be stored with the image data. Once the dimensions are known however, raster images are easily condensed into a single stream of data. Recreating an image from a single data stream is then simply a matter of painting the appropriate number of pixels per line.

In HDF, an ***8-bit raster image*** is defined as any bit-mapped image whose rows are painted from left to right using pixels whose values are represented by a stream of 8-bit numbers (See FIGURE 2.2a). As a simple example of the mechanics behind 8-bit raster imaging, consider a stream of 8-bit numbers representing raster image data. To display the data as a bit-map, the color associated with the first number in the data stream must appear in the upper left corner of the image. The remainder of the first line is then painted from left to right using as many values from the data stream as necessary to complete the line. The remainder of the rows are similarly painted from left to right, top to bottom until every value in the data stream appears as one pixel in the image.

FIGURE 2.2a **An 8-bit raster image(a) is a bit-map(b) whose rows are painted from left to right(c) using pixels(d) whose values are stored as a single stream of 8-bit numbers(e)**.



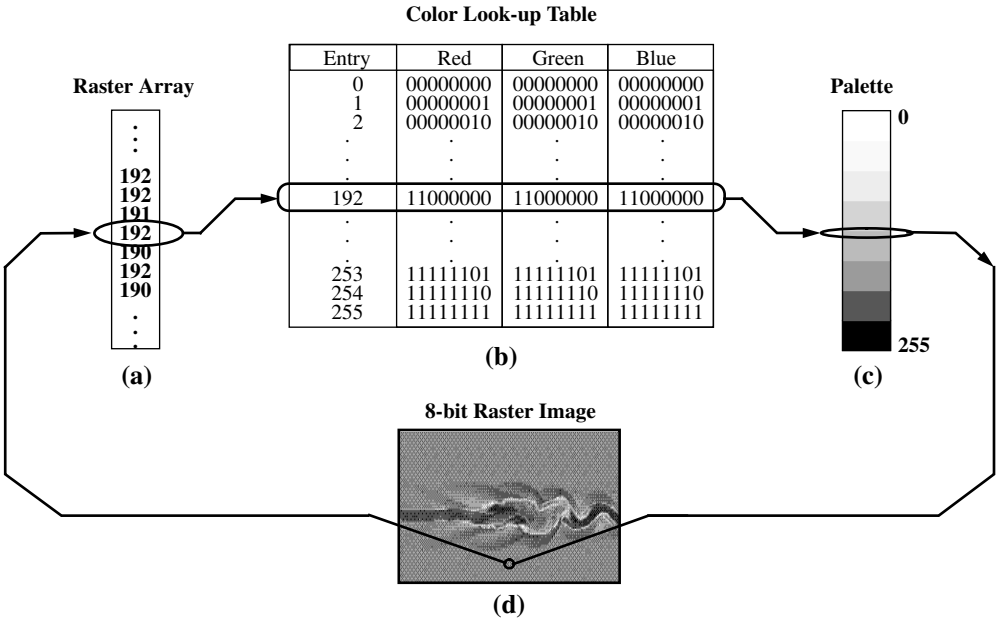
The HDF ***8-bit raster model*** provides a framework for storing 8-bit raster images along with their image dimensions and palettes. Any 8-bit raster image array accompanied by its dimension record is recognized as a ***8-bit raster image set***. A raster image set may also include a palette, however, palettes are optional and may be omitted. To learn more about raster image sets, consult the HDF User’s Guide.

2.3 The Palette Model

Palettes are the means by which color is applied to 8-bit raster image data. Recall from the previous section that an 8-bit raster image is stored as a single stream of data where each pixel value is represented by one 8-bit number. Because pixel values are stored as 8-bit numbers, they are limited to values between 0 and 255.

Every pixel value appearing in an image is assigned a color through a ***color look-up table***. In one sense, pixel values serve as an index for 256 entries stored in a color look-up table (See FIGURE 2.3a). On the other hand, pixel values also represent the color entries in a ***palette***. Whether you prefer to think of assigning colors via the numerical entries in a color look-up table or the color entries in a palette, both represent a means for assigning the same color to every pixel with the same pixel value. For example, every pixel assigned a pixel value of 192, will be painted using the values stored in table entry 192 or the color stored in palette entry 192. If the color entry or its numerical equivalent is changed, every pixel in the image with a value of 192 will also change.

FIGURE 2.3a **For every pixel value in a raster array(a) there is one entry in a color look-up table(b) that will determine its color in the palette(c), and thus its appearance in the resultant image (d).**



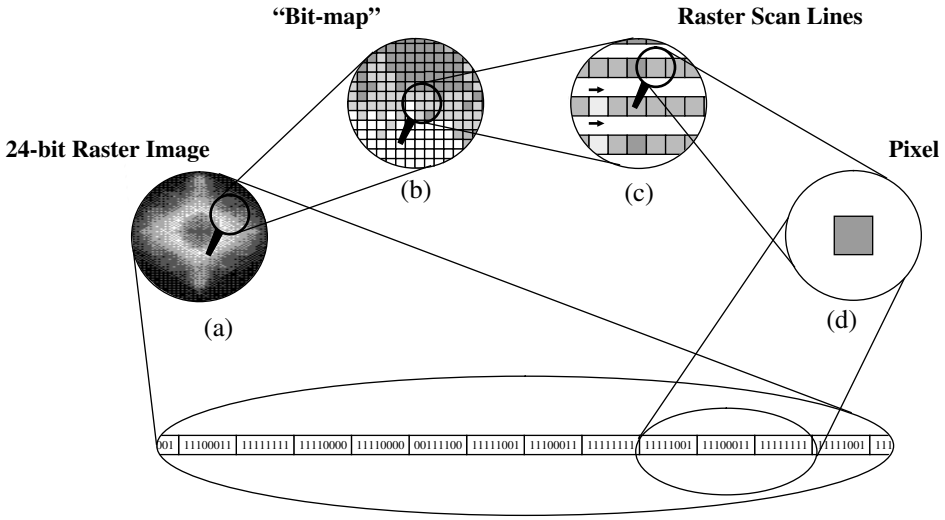
For every color appearing in an image, there are three 8-bit numbers in the color look-up table describing its appearance. Each 8-bit number or ***color component*** represents the amount of red, green, or blue used to create a particular color. Because each component is stored as an 8-bit number, the amount of red, green, or blue added to a color can range in value from 0 to 255. After a quick calculation, it's easy to see that there are 16.7 million (2^{24}) possible color combinations, but only 256 entries available in the color table. As a result, 256 colors out of a possible 16.7 million must be selected and assigned a position in the color look-up table. Selecting 256 colors for a color look-up table is also known as creating a palette.

In many ways, the ***palette model*** is the simplest data model in HDF. It provides a framework for storing color components as three 256-byte segments in a single 768-byte array. The palette model is most often used for working with multiple palettes, or palettes not specifically assigned to a raster image. To learn more about working with the palette model, consult the HDF User's Guide.

2.4 The 24-bit Raster Model

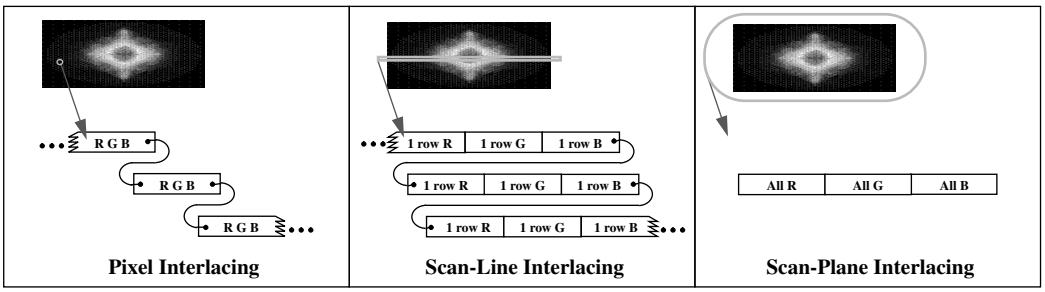
In terms of HDF, the definition of a 24-bit raster image is identical to that of an 8-bit raster image except that each pixel value is represented by three 8-bit numbers instead of one. More specifically, a ***24-bit raster image*** is any bit-mapped image whose rows are painted from left to right using three 8-bit numbers to represent each pixel value (See FIGURE 2.4a). Because each set of 8-bit numbers represents the RGB components for one pixel in the image, 24-bit images are not limited to the 256 color entries available in a color look-up table. The trade-off, however, is that 24-bit images occupy arrays three times larger than those used to store an equivalent set of 8-bit raster images.

FIGURE 2.4a **A 24-bit raster image(a) is a bit-map(b) whose rows are painted from left to right(c) using pixels(d) whose values are stored as three 8-bit numbers in a single stream of data(e).**



HDF supports three options for grouping the color components in 24-bit raster image data. These options consist of ***pixel interlacing***, ***scan-line interlacing***, and ***scan-plane interlacing*** (See FIGURE 2.4b). Storing the color components grouped by pixel, as in red-green-blue, red-green-blue, etc., is called pixel interlacing. Storing the color components by line, as in 1 row of red, 1 row of green, 1 row of blue, 1 row red, etc., is called scan-line interfacing. Finally, storing the color components grouped by color plane, as in the red components first, the green components second, and the blue components last, is called scan-plane interlacing. Unless otherwise specified, the HDF 24-bit raster model assumes that all 24-bit images are stored using pixel interlacing.

FIGURE 2.4b **HDF supports three different RGB interlacing schemes for 24-bit raster images.**

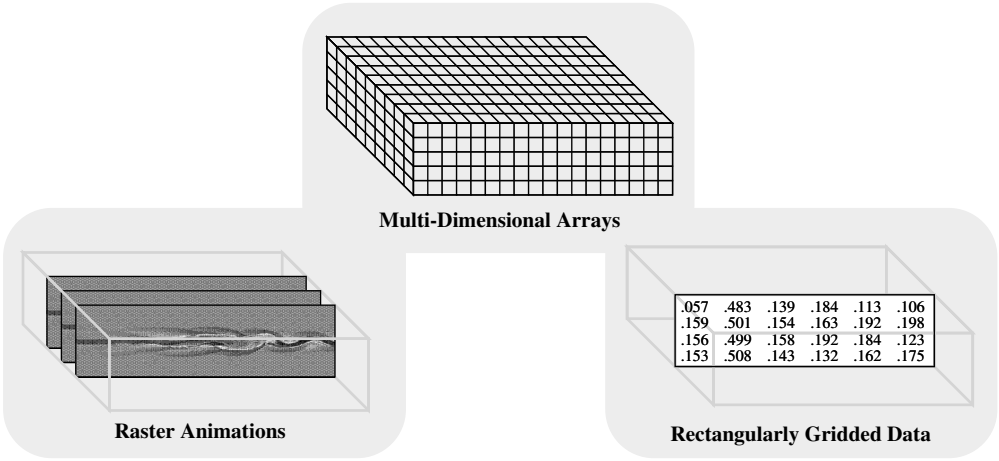


The ***24-bit raster model*** provides a framework for storing 24-bit raster images along with their image dimensions. Any 24-bit raster image accompanied by its image dimensions is known as a ***24-bit raster image set***. To learn more about 24-bit raster image sets, consult the HDF User's Guide.

2.5 The Scientific Data Model

Although any data contributing to a scientific investigation can be considered "scientific," HDF uses the phrase scientific data to refer to any data not specifically designed to be viewed as a raster image. Of course this does not preclude viewing scientific data as a raster image, it simply means that scientific data may be stored or viewed as something other than a two-dimensional array of 8, or 24-bit integers (See FIGURE 2.5a). In fact, HDF defines scientific data as any multi-dimensional array containing 8, 16, or 32-bit integers or 32- or 64-bit floating point numbers.

FIGURE 2.5a **Scientific data can be viewed in a variety of different ways.**



In addition to providing a framework for storing data of arbitrary dimensions and number type, the ***scientific data model*** also provides support for six individual ***attribute records***. Each attribute record is optional and describes one particular facet of the environment from which the scientific data was taken. The six predefined attribute records are described as follows:

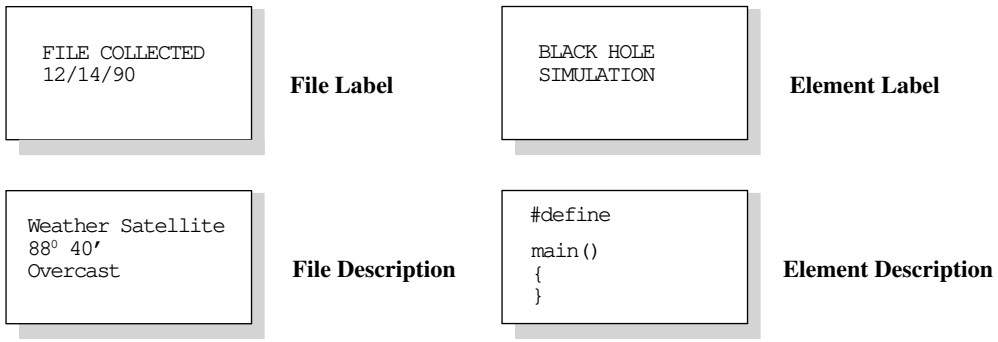
- **Coordinate System**: Identifies which coordinate system to use when displaying the data.
- **Formatting**: Specifies the format to use when displaying values for the data and its attributes.
- **Label**s: Contains a name for each independent variable and/or the data.
- **Max/Min Value**: Reports the maximum and minimum values found in the data.
- **Scale**: Describes the scale to use along each axis.
- **Units**: Identifies the unit associated with each dimension in the data.

In HDF, any multi-dimensional array accompanied by its dimension record and number type qualifies as a ***scientific data set***. Scientific data sets may also include attribute records, however, attribute records are optional and may be omitted. To learn more about scientific data sets, consult the HDF User's Guide.

2.6 The Annotation Model

When working with several different data types, it is often convenient to identify the contents of a file by adding a short text description or ***annotation***. Sometimes an annotation can serve as a short label for a file or data element, as in `COLLECTED 12/14/90` or `BLACK HOLE SIMULATION` (See FIGURE 2.6a). On other occasions, an annotation may require a more complex or lengthy description. For example, if the data collected on 12/14/90 originated as satellite data, the description might include the source of the data, pertinent environmental conditions, or other relevant information. Or in the case of the black hole simulation, the description might contain source code for the program that produced the simulation data.

FIGURE 2.6a **HDF can annotate both files and elements using labels or descriptions.**



Short annotations used to label files or data elements are called ***labels***, whereas long annotations are called ***descriptions***. Here the difference between short and long is not determined by byte length, but rather by the number of strings contained in the annotation. Labels are limited to one string containing any sequence of characters except zero, which will terminate the string. Descriptions, on the other hand, can contain any sequence of character data including zero, hence a description may contain any number of strings.

The HDF ***annotation model*** provides a framework for appending labels or descriptions to HDF files or the elements they contain. Although annotating a file is a rather simple operation, annotating a data element is bit more difficult. The difficulty lies in attaching the annotation to a specific piece of data in a file. To learn more about annotating HDF files and the elements they contain, consult the HDF User's Guide.

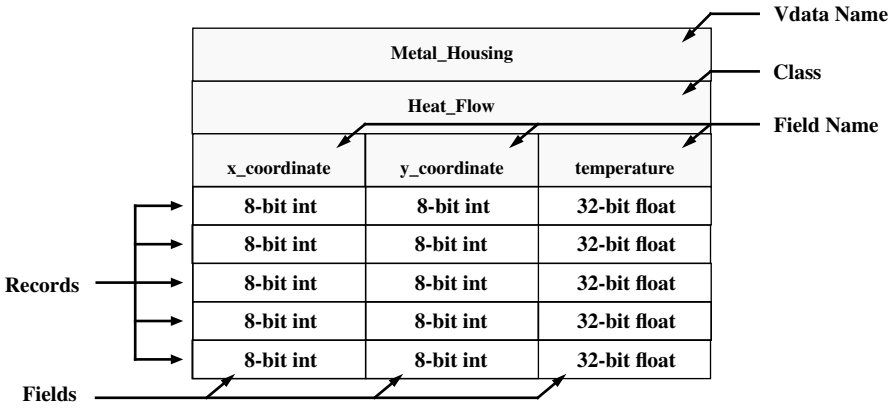
2.7 The Vdata Model

The HDF ***Vdata model*** provides a framework for storing customized tables in HDF files. Each table is comprised of a collection of similar records called ***Vdata records***, whose values are stored in fixed length fields. Each field can support its own number type, however, every Vdata record in a table must contain the same type of fields. Valid number types include 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating point numbers.

Vdata tables use three kinds of identifying information: a name, a class, and a series of individual field names. A ***Vdata name*** is a label typically assigned to describe the origin or contents of a table. It often serves as a search key for files containing multiple tables. A ***Vdata class*** further distinguishes an individual table by identifying the meaning of its data. A class assignment often aids in determining how to interpret the fields in a record. Finally, ***Vdata field names*** identify the individual fields that make up a record.

Consider a collection of Vdata records containing two 8-bit fields followed by a 32-bit field (See FIGURE 2.7a) Assume that each record represents the location and temperature of a specific point on a heated metal housing. If each location is represented by an x, y, and temperature value, “x_coordinate,” “y_coordinate,” and “temperature” are certainly logical choices for their field names. “Heat_Flow” and “Metal_Housing” also represent appropriate possibilities for the Vdata class and name. Heat_Flow is a fairly accurate description of what type of records are stored in the table. Any software capable of recognizing Heat_Flow Vdata, can expect to encounter the location of each point followed by its temperature. Metal_Housing is a useful name for the Vdata table because it can serve as a search key in a file containing the heat flow tables for several different items.

FIGURE 2.7a **The constituent parts of a Vdata table.**



The HDF Vdata model provides a framework for storing Vdata records in customized tables. Since the Vdata interface is more complex than those of the other HDF data models, it is not covered in this document. To learn more about reading and writing Vdata, consult the HDF User's Guide.

Chapter 3

The Bare Necessities for Reading and Writing Data

3.0 Chapter Overview

Each of the six data models supported by HDF provides an interface for reading, writing and organizing data of a particular type. In Chapter 4 you will learn to perform a simple read and write operation for each type of data using the "add" and "get" functions available in each model. Each of the six sections in this chapter is dedicated to a different data model. As a result, each section introduces one data model, describes its interface and provides both a C and FORTRAN example for reading and writing data using the appropriate function calls.

In an effort to promote simplicity, most of the code examples presented in Chapter 4 use the minimum number of function calls necessary to store and retrieve data from an HDF file. In most cases, these examples take advantage of HDF default parameters to simplify the process of reading and writing data. However, there are a few examples that include optional function calls in order to demonstrate how to override default settings. All optional calls are clearly marked as such.

In Chapter 4 you will learn to perform the following operations:

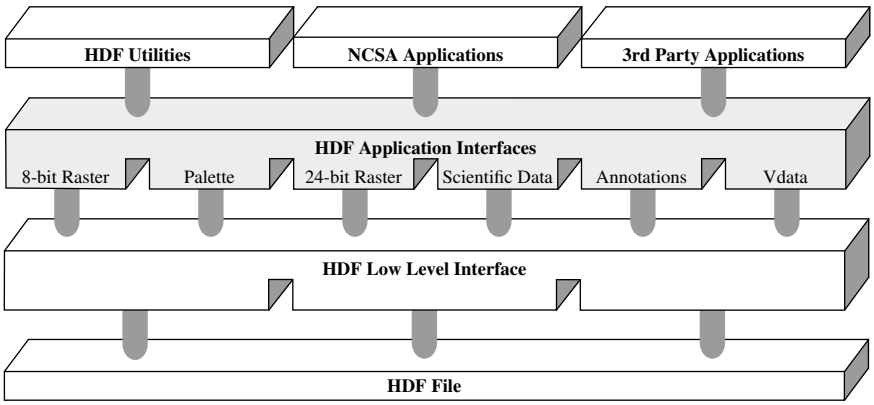
- The ***8-bit raster model***: Write and read an 8-bit raster image.
- The ***palette model***: Write and read an 8-bit palette outside of a raster image set.
- The ***24-bit raster model*** Write and read a 24-bit image using scan-plane interlacing.
- The ***scientific data model*** Write and read a 3-dimensional array 64-bit floats.
- The ***annotation model***: Write and read a file description.
- The ***Vdata model:*** Write and read 32-bit floating-point records.

After reading this chapter, you will know enough about HDF to begin creating simple files. In fact, after you complete this material you can use the data handling techniques introduced here without learning another thing. Of course, in order to master more sophisticated data hierarchies you'll need to continue working your way through this manual and the HDF User's Guide.

3.1 Introducing the Application Interfaces Supported by HDF

There are currently six different application interfaces available in the HDF library (See FIGURE 3.1a). Each interfaces provides a collection of functions for reading, writing, and organizing one type of data and its associated data elements.

FIGURE 3.1a **HDF supports six different application interfaces.**



In general, the more flexible the interface, the more "set-up" or "query" work you must do before reading or writing data (See TABLE 3A). The 8-bit raster data model, for example, is extremely well-defined, therefore no set-up work is required before reading or writing 8-bit raster data. Both operations are handled through one function call assuming the image dimensions are known. On the other hand, the most flexible, and least-structured interface is the Vdata interface. As a result, reading or writing Vdata takes a bit more set-up and query work than any other type of data.

TABLE 3A **Some function calls available for reading and writing data via the different HDF data models.**

Operation		8-bit Raster	Palette	24-bit Raster	Scientific Data	Annotations	Vdata
Writing	**Set-up**	-	-	-	DFSDsetNT*	-	VSattach VSsetname* VSfdefine VSsetfields VSdetach
	Write	DFR8addimage	DFPaddpal	DF24addimage	DFSDadddata	DFANaddfds	VSwrite
Reading	**Query**	DFR8getdims*		DF24getdims*	DFSDgetdims* DFSDgetNT*	DFANgetfdslen	VSgetid* VSattach VSinquire VSsetfields VSdetach
	Read	DFR8getimage	DFPgetpal	DF24getimage	DFSDgetdata	DFANgetfds	VSread

*indicates optional function call

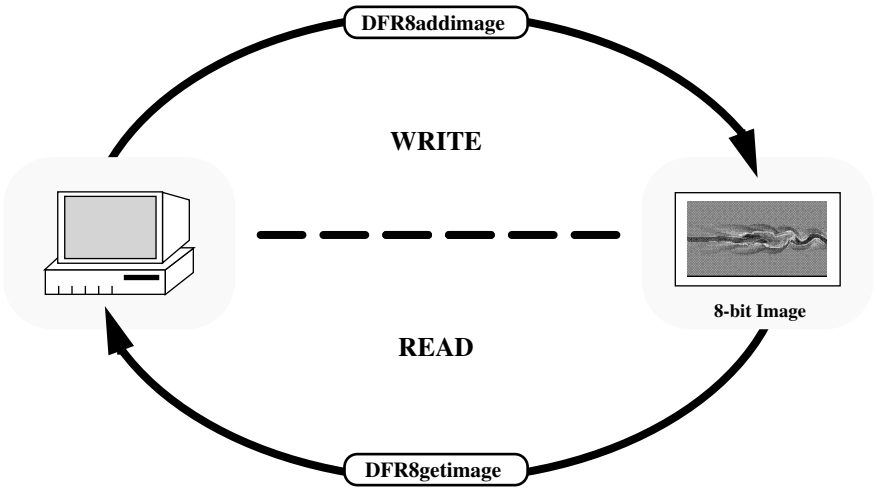
Every application interface in the HDF library is supported by an independent set of function calls. Each set of function calls is assigned an ***interface prefix*** which identifies the interface to which it belongs. In most cases the prefix is an abbreviation for the data model supported by the interface. For example, all 8-bit raster functions use the prefix DFR8; an abbreviation for "data format, raster 8." The palette model uses DFP, for "data format, palette" and the scientific data model uses DFSD for "data format, scientific data." Once you are familiar with the six different interfaces, identifying a function call by it prefix becomes quite simple.

3.2 HDF and the 8-bit Raster Interface

Every function in the 8-bit raster interface begins with the prefix ***DFR8*** which stands for "data format, raster 8." By using DFR8 function calls, HDF can make a series of assumptions about the raster operations you are performing. These assumptions, or ***default settings***, reduce the amount of work you must do to perform a successful read or write operation.

When you make a DFR8 call, HDF assumes that you are interested in reading or writing data as a two-dimensional array of 8-bit integers. Consequently, reading or writing an 8-bit raster image to file requires only one function call (See FIGURE 3.2a). To write a raster to a file, you can use **DFR8addimage**. To read a raster from a file, you must use **DFR8getimage**.

FIGURE 3.2a **8-bit image data can be read or written to a file via a single read or write function call.**



Although **DFR8addimage** and **DFR8getimage** are the only two calls required for reading and writing 8-bit raster data, the DFR8 interface provides several additional functions for more advanced operations. For more information on additional DFR8 function calls, consult the HDF User's Guide.

Writing an 8-bit Raster Image to an HDF File

If you are programming in C, **DFR8addimage** is the only function call you need to write a raster image to file**.** In FORTRAN, the equivalent function is called **d8aimg**. To write an 8-bit raster, the call made from your program should look something like this:

```
C:        DFR8addimage (filename, image, width, height, compress)
FORTRAN:  d8aimg (filename, image, width, height, compress)
```

In either case, the "add image" call appends an image to an existing HDF file or creates a new file if the named file does not exist. It does not overwrite data in existing files. The parameters for both the C and FORTRAN call are more explicitly defined below (See TABLE 3B on page 20).

TABLE 3B **The parameters required for DFR8addimage.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFR8addimage (d8aimg)	filename	character *(*)	character pointer	Name of file to store the image data.
	image	character array	character array	Array containing the image data.
	width	integer	32-bit integer	Number of columns in the image.
	height	integer	32-bit integer	Number of rows in the image.
	compress	integer	integer	Type of compression: (none=0, run length=DFTAG_RLE).

Notice in the following example, that the order in which the dimensions for the image array are declared differs between C and FORTRAN. To efficiently read and write raster data, the FORTRAN declaration requires the width (rows) before the height (columns), whereas the C declaration requires the height before the width.

EXAMPLE 1. **Writing an 8-bit raster image to an HDF file.**

In the following code example, **DFR8addimage** and **d8aimg** are used to write an 8-bit image to a file called “example.hdf.” In both the C and FORTRAN example, the pixel values for the raster image are initialized using the following values:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6

main()
{

    /* Initialize the image array */
    static uint8 raster_data[HEIGHT][WIDTH] =

                                          { 1,  2,  3,  4,  5,
                                            6,  7,  8,  9, 10,
                                           11, 12, 13, 14, 15,
                                           16, 17, 18, 19, 20,
                                           21, 22, 23, 24, 25,
                                           26, 27, 28, 29, 30 };


    /* Write the 8-bit raster image to file */
    DFR8addimage("example.hdf", raster_data, WIDTH, HEIGHT, 0);
}
```

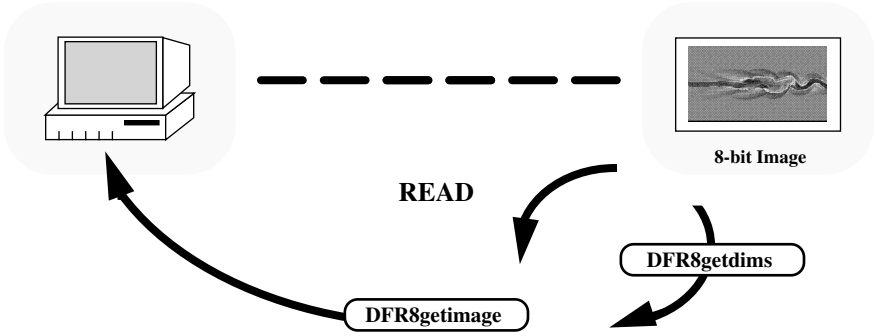
FORTRAN:

```
      PROGRAM RASTER8

      character*1 raster_data(5,6)
      integer istat, d8aimg

C Initialize the image array
      data raster_data /  1,  2,  3,  4,  5,
     $                    6,  7,  8,  9, 10,
     $                   11, 12, 13, 14, 15,
     $                   16, 17, 18, 19, 20,
     $                   21, 22, 23, 24, 25,
     $                   26, 27, 28, 29, 30 /

C Write the 8-bit raster image to the file
      istat = d8aimg('example.hdf', raster_data, 5, 6, 0)

      stop
      end
```

Reading an 8-bit Raster Image from an HDF File

If you are programming in C, **DFR8getimage** is the only function call you need to read a raster image from a file**.** In FORTRAN, the equivalent function is called **d8gimg**. If you are not sure of the dimensions of an image you intend to read, the **DFR8getdims** function (or **d8gdims** in FORTRAN) can obtain them for you (See FIGURE 3.2b). **DFR8getdims** is optional. However, by using **DFR8getdims** in your program, you can verify the size of the space required to hold the image before reading any data.

FIGURE 3.2b **DFR8getdims can be used before DFR8getimage to obtain the dimensions of an image.**



If you already know the dimensions of the image you intend to read, the function call made from your program should look something like this:

C:
```
DFR8getimage (filename, image, width, height, palette)
```
FORTRAN:
```
d8gimg (filename, image, width, height, palette)
```

However, if you wish to verify the dimensions before reading the image, your program should include the following calls:

C:
```
DFR8getdims (filename, width, height, haspalette)
DFR8getimage (filename, image, width, height, palette)
```

FORTRAN:
```
d8gdims (filename, width, height, haspalette)
d8gimg (filename, image, width, height, palette)
```

The first time the get dimensions call is made, it returns information about the first 8-bit raster image in the file. Any subsequent calls will return dimensions for successive images in the file. The “get image” call operates in a similar manner. The parameters for both the “get dimensions” and the “get image” calls are more explicitly defined below(See TABLE 3C).

TABLE 3C **The parameters required for DFR8getdims and DFR8getimage.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFR8getdims (d8gdims)	filename	character *(*)	character pointer	Name of file containing the image data.
	width	integer	32-bit integer pointer	Number of columns.
	height	integer	32-bit integer pointer	Number of rows.
	haspalette	integer	integer pointer	Reports on palette (yes=1, no=0).
DFR8getimage (d8gimg)	filename	character *(*)	character pointer	Name of file containing the image data.
	image	character array	character array	Array to hold the image data.
	width	integer	32-bit integer	Width of the buffer for the image data.
	height	integer	32-bit integer	Height of the buffer for the image data.
	palette	character array	character array	Array to hold the palette data.

Notice in the following example, as in the case of **DFR8addimage**, that the order in which the dimensions for the image array are declared differs between C and FORTRAN. The FORTRAN declaration requires the width before the height, whereas the C declaration requires the height before the width.

EXAMPLE 2. **Reading an 8-bit Raster Image from an HDF file**

The following example will query a file called “example.hdf” for the dimensions of an 8-bit image. Although the **DFR8getdims** call is optional, it is included as a demonstration of how to check the dimensions of an image before reading its image data. If the dimensions are already known, only the **DFR8getimage** call is required. This example also assumes that no palette is grouped with the image data, therefore NULL is passed as the palette parameter. Any time the palette argument is NULL (or 0 in the case of FORTRAN), all palette data is ignored.

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6

main()
{
    uint8 raster_data[HEIGHT][WIDTH];
    int32 width, height;
    intn haspal;

    /* Get the dimensions of the image */
    DFR8getdims("example.hdf", &width, &height, &haspal);

    /* Read the raster data if the dimensions are correct */
    if(width<=WIDTH && height<=HEIGHT)
        DFR8getimage("example.hdf",raster_data,width,height,NULL);
}
```

FORTRAN:

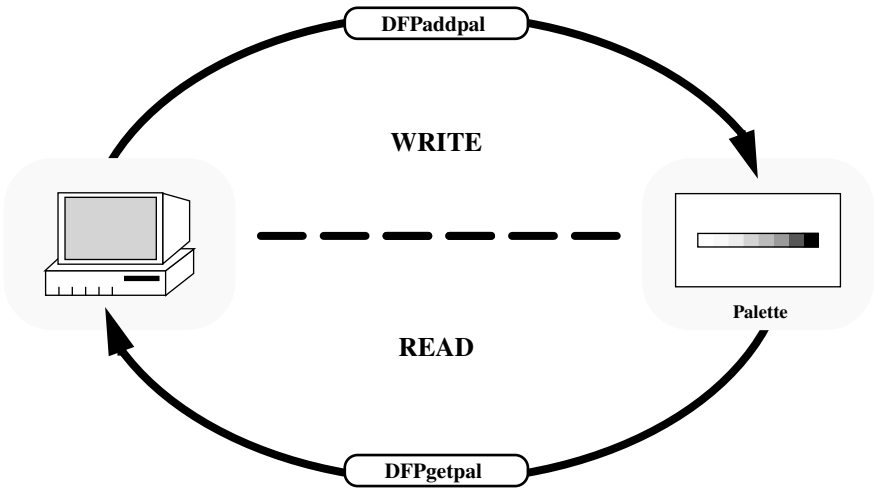
```
      PROGRAM RASTER8

      character*1 image(5,6)
      integer istat, height, width, d8gimg, haspal

C     Get the dimensions of the image
      istat = d8gdims('example.hdf', width, height, haspal)

C     Read the raster data if the dimensions are correct
      if ( (width .le. 6) .and. (height .le. 5) ) then
         istat = d8gimg('example.hdf', image, width, height, 0)
      endif

      stop
      end
```

3.3 HDF and the Palette Interface

Every function in the palette interface begins with the prefix ***DFP***, which stands for "data format, palette." By using DFP function calls, HDF can make a series of assumptions about the operations you are performing. These assumptions reduce the amount of work you must do to perform a successful read or write operation.

When you make a DFP call, HDF assumes you are interested in reading or writing data as a 768-byte array of 8-bit integers. HDF also assumes that the first 256-bytes in the array contain the red components, the second 256-bytes contain the green components, and that the last 256-bytes contain the blue components. Based on these assumptions, reading or writing a palette to an HDF file requires only one function call (See FIGURE 3.3a). To write a palette to a file, you can use the **DFPaddpal** function call. To read a palette from a file you must use **DFPgetpal**.

FIGURE 3.3a **Palettes can be read or written to a file via a single read or write function call.**



Although **DFPaddpal** and **DFPgetpal** are the only calls required for reading and writing palettes, the DFP interface provides several additional functions for more advanced palette operations. For more information on additional DFP function calls, consult the HDF User's Guide.

Writing a Palette to an HDF File

If you are programming in C, **DFPaddpal** is the only function call you need to write a palette to file**.** In FORTRAN, the equivalent function is called **dpapal**. To write a palette via the DFP interface, your program should contain the following function call:

```
C:          DFPaddpal (filename, palette)
FORTRAN:    dpapal (filename, palette)
```

In either case, the "add palette" call appends a palette to an existing HDF file or create a new one if the named file does not exist. It does not overwrite data in existing files. The parameters for both calls are more explicitly defined below (See TABLE 3D on page 25).

TABLE 3D **The parameters required for DFPaddpal.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFPaddpal (dpapal)	filename	character *(*)	character pointer	Name of file that will store the palette.
	palette	character array	character array	Array holding the palette.

Calling **DFPaddpal** immediately after writing an 8-bit raster image will not explicitly group the palette with the preceding image. Palettes written to file sequentially can be retrieved sequentially. However, to maintain a higher level of organization between multiple palettes and images stored in the same file, it's often better to explicitly group each palette with the image to which it belongs. To find out more about assigning a palette to an image, see the HDF User's Guide.

EXAMPLE 3. **Writing a palette to an HDF file**

In the following code example, **DFPaddpal** and **dpapal** are used to write a palette to a file called "example.hdf." In both the C and FORTRAN example, the palette data is initialized to the following grayscale values:

Palette Entry
0
1
. . .
255

Red Component	Green Component	Blue Component
0	0	0
1	1	1
. . .	. . .	. . .
255	255	255

C:

```
#include "hdf.h"

main()
{
    uint8 palette_data[768];
    intn i;

/* Initialize the palette to grayscale */
for(i=0; i<256; i++) {
    palette_data[i*3]=i;
    palette_data[i*3+1]=i;
    palette_data[i*3+2]=i;
  }

/* Write the palette to file */
DFPaddpal("example.hdf",palette_data);
}
```

FORTRAN:

```
      PROGRAM PALETTE

      INTEGER         dpapal, ret, i
      CHARACTER*1     palette_data(768)

C Initialize the palette to grayscale
      do 10, i=1,256
         palette_data((i-1)*3+1) = char(i-1)
         palette_data((i-1)*3+2) = char(i-1)
         palette_data((i-1)*3+3) = char(i-1)
   10 continue

C Write the palette to file
      ret = dpapal('example.hdf', palette_data)

      stop
      end
```

Reading a Palette from an HDF File

If you are programming in C, **DFPgetpal** is the only function call you need to read a palette from a file**.** In FORTRAN, the equivalent function is called **dpgpal**. To read a palette, the function call made from your program should look something like this:

```
C:        DFPgetpal (filename, palette)
FORTRAN:  dpgpal (filename, palette)
```

The first time a “get palette” call is made, it returns the first palette in the file. Repeatedly calling get palette returns successive palettes in the file. Successive get palette calls made via the DFP interface will return all palettes in a file, even those stored via the DFR8 interface. The parameters for the “get palette” call are more explicitly defined below (See TABLE 3E).

TABLE 3E **The parameters required for DFPgetpal.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFPgetpal (dpgpal)	filename	character*(*)	character pointer	Name of file storing the palette.
	palette	character array	character array	Buffer to hold the palette.

EXAMPLE 4. **Reading a Palette from an HDF file**

The following example will read a palette from an HDF called “example.hdf.”

C:

```
#include "hdf.h"

main()
{
    uint8 palette_data[768];

/* Read the palette data from a file */
DFPgetpal("example.hdf",palette_data);
}
```

FORTRAN:

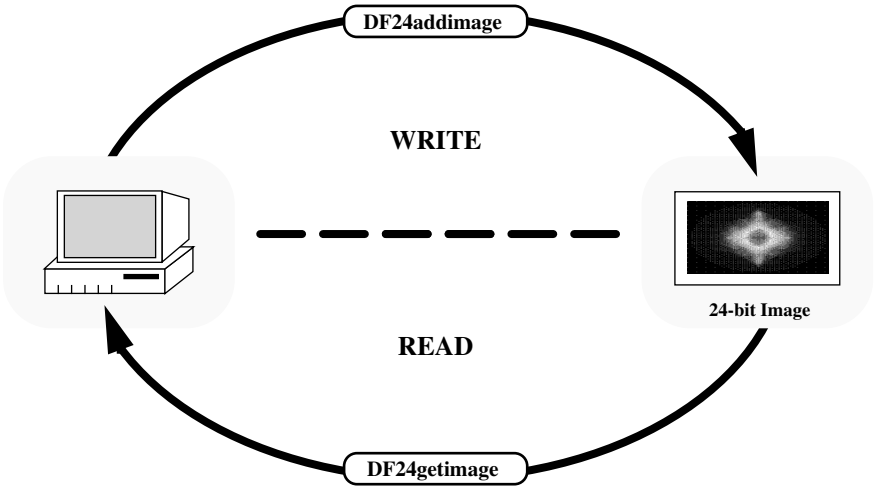
```
      PROGRAM PALETTE

      INTEGER         dpgpal
      CHARACTER*1     palette_data(768)

C Read the palette from a file
      ret = dpgpal('example.hdf', palette_data)

      stop
      end
```

3.4 HDF and the 24-bit Raster Image Interface

Every function in the 24-bit raster interface begins with the prefix ***DF24*** which stands for "data format, 24-bit image." By using DF24 function calls, HDF can make a series of assumptions about the raster operations you are performing. These assumptions reduce the amount of work you must do to perform a successful read or write operation.

When you make a DF24 call, HDF assumes that you are interested in reading or writing data as a three-dimensional array of 8-bit integers. HDF also assumes that you are reading or writing the image data using pixel interlacing. These assumptions make it possible for you to read or write 24-bit images using only one function call (See FIGURE 3.4a). To write an image to a file, you can use **DF24addimage**. To read a raster from a file, you must use **DF24getimage**.

FIGURE 3.4a **24-bit images can be read or written to a file via one read or write function call.**

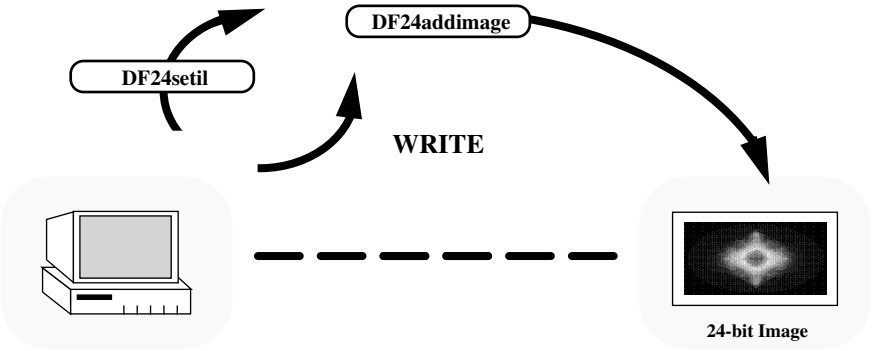


Although **DF24addimage** and **DF24getimage** are the only calls required for reading or writing 24-bit raster data to file, the DF24 interface provides a host of optional function for accessing data through more advanced operations. For more information on additional DF24 function calls, consult the HDF User's Guide.

Writing a 24-bit Image to an HDF File

If you are programming in C, **DF24addimage** is the only function call you need to write a 24-bit image to file. In FORTRAN, the equivalent function is called **d2aimg**. In either case, the default setting is pixel interlacing. If you prefer to store the image using scan-line interlacing or scan-plane interlacing, you must call **DF24setil** (or **d2setil** in FORTRAN) to change the interlacing scheme (See Figure 3.4b on page 28).

FIGURE 3.4b **DF24setil can be used to override the default pixel interlace setting.**



If you wish to write a 24-bit raster image to file using pixel interlacing, the function call made from your program should look something like this:

C: `DF24addimage (filename, image, width, height)`
FORTRAN: `d2aimg (filename, image, width, height)`

If you wish to change the interlacing scheme, your program should include the following calls:

C:
```
DF24setil (interlacing)
DF24addimage (filename, image, width, height)
```

FORTRAN:
```
d2setil (interlacing)
d2aimg (filename, image, width, height)
```

The parameters for both the “set interlace” and “get image” calls are more explicitly defined below (See TABLE 3F). Notice that passing DFIL_LINE via the interlacing argument in **DF24setil**, changes the pixel interlacing scheme to scan-line interlacing. In the same sense, passing DFIL_PLANE changes the interlacing to scan-plane interlacing.

TABLE 3F **The parameters required for DF24setil and DF24addimage.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DF24setil (d2setil)	interlacing	integer	integer	Specifies the type of interlacing to use. DFIL_PIXEL = pixel, DFIL_LINE = scan-line DFIL_PLANE =scan-plan
DF24addimage (d2aimg)	filename	character *(*)	character pointer	Name of file that will store the image.
	image	character array	character array	Array containing the image data.
	width	integer	32-bit integer	Number of columns in the image.
	height	integer	32-bit integer	Number of rows in the image.

EXAMPLE 5.

Writing a 24-bit Raster Image to an HDF File

In the following example, **DF24addimage** and **d2aimg** are used to write a 24-bit image to a file called "example.hdf." Although the **DF24setil** function call is optional, it is included here to demonstrate how to change the default interlace setting from pixel interlacing to scan-plane interlacing. Consequently in both the C and FORTRAN example, the pixels for the image are initialized using the following values:

	1	2	3	4	5		
	6	7	8	9	10		
	11	1	2	3	4	5	
	16	6	7	8	9	10	
	21	11	1	2	3	4	5
Red Plane	26	16	6	7	8	9	10
		21	11	12	13	14	15
Green Plane		26	16	17	18	19	20
			21	22	23	24	25
Blue Plane			26	27	28	29	30

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6

main()
{
    /* Initialize the image array */
    static uint8 raster_data[3][HEIGHT][WIDTH] =

                                          { 1,  2,  3,  4,  5,
                                            6,  7,  8,  9, 10,
                                           11, 12, 13, 14, 15,
                                           16, 17, 18, 19, 20,
                                           21, 22, 23, 24, 25,
                                           26, 27, 28, 29, 30,

                                            1,  2,  3,  4,  5,
                                            6,  7,  8,  9, 10,
                                           11, 12, 13, 14, 15,
                                           16, 17, 18, 19, 20,
                                           21, 22, 23, 24, 25,
                                           26, 27, 28, 29, 30,

                                            1,  2,  3,  4,  5,
                                            6,  7,  8,  9, 10,
                                           11, 12, 13, 14, 15,
                                           16, 17, 18, 19, 20,
                                           21, 22, 23, 24, 25,
                                           26, 27, 28, 29, 30  };

    /* Change interlace from pixel to scan-plane */
    DF24setil(DFIL_PLANE)

    /* Write the 24-bit image data to file */
    DF24addimage("example.hdf", raster_data, WIDTH, HEIGHT);
}
```

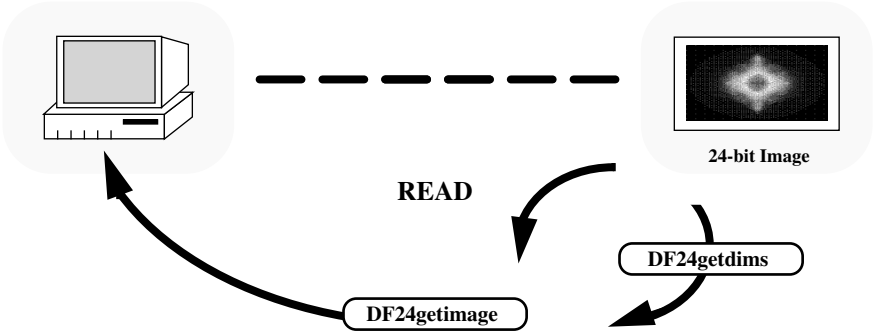
FORTRAN:

```
      PROGRAM RASTER24

      INTEGER         d2aimg, d2setil, ret, width, height
      CHARACTER*1     raster_data(5,6,3)
      DATA            width/5/, height/6/

C Initialize the image array
      DATA            raster_data /  1,  2,  3,  4,  5,
     $                               6,  7,  8,  9, 10,
     $                              11, 12, 13, 14, 15,
     $                              16, 17, 18, 19, 20,
     $                              21, 22, 23, 24, 25,
     $                              26, 27, 28, 29, 30,
     $
     $                               1,  2,  3,  4,  5,
     $                               6,  7,  8,  9, 10,
     $                              11, 12, 13, 14, 15,
     $                              16, 17, 18, 19, 20,
     $                              21, 22, 23, 24, 25,
     $                              26, 27, 28, 29, 30,
     $
     $                               1,  2,  3,  4,  5,
     $                               6,  7,  8,  9, 10,
     $                              11, 12, 13, 14, 15,
     $                              16, 17, 18, 19, 20,
     $                              21, 22, 23, 24, 25,
     $                              26, 27, 28, 29, 30 /

C Change interlace from pixel to scan plane
      ret = d2setil(2)

C Write the 24-bit image data to file
      ret = d2aimg('example.hdf', raster_data, width, height)

      stop
      end
```

Reading a 24-bit Raster Image from an HDF File

If you are programming in C, **DF24getimage** is the only function call you need to read a 24-bit raster image from a file**.** In FORTRAN, the equivalent function is called **d2gimg**. If you are not sure of the dimensions for an image you are attempting to read, the **DF24getdims** function (or **d2gdims** in FORTRAN) can obtain them for you (See FIGURE 3.4c). **DF24getdims** is optional. However, by using **DF24getdims** in your code, you can verify the size of the space required to hold the image data before reading any data. **DF24getdims** will also verify the interlacing scheme.

FIGURE 3.4c **DF24getdims can be used before DF24getimage to obtain the dimensions of an image.**



If you know the dimensions for the image you intend to read, the function call made from your program should look something like this:

```
C:          DF24getimage (filename, image, width, height)
FORTRAN:    d2gimg (filename, image, width, height)
```

However, if you wish to verify the dimensions before reading the image, your program should include the following calls:

```
C:          DF24getdims (filename, width, height, interlace)
            DF24getimage (filename, image, width, height)

FORTRAN:    d2gdims (filename, width, height, interlace)
            d2gimg (filename, image, width, height)
```

The first time a “get dims” call is made, it returns information about the first 24-bit raster image in the file. Repeatedly calling get dims returns information regarding successive 24-bit images in the file. “Get image” calls operate in a similar manner. Parameters for both the get dims and the get image call are more explicitly defined below (See TABLE 3G).

TABLE 3G **The parameters required for DF24getdims and DF24addimage.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DF24getdims (d2gdims)	filename	character *(*)	character pointer	Name of file containing the image data.
	width	integer	32-bit integer pointer	Number of columns in the image.
	height	integer	32-bit integer pointer	Number of rows in the image.
	interlace	integer	integer pointer	Type of color component interlacing
DF24getimage (d2gimg)	filename	character *(*)	character pointer	Name of file containing the image data.
	image	character array	character array	Buffer to hold the image data.
	width	integer	32-bit integer	Width of the image data buffer.
	height	integer	32-bit integer	Height of the image data buffer.

In the following example, as in the case of **DFR8addimage**, the order in which the dimensions for the image array are declared differs between C and FORTRAN. To efficiently read or write raster data, the FORTRAN declaration requires the width before the height, whereas the C declaration requires the height before the width.

EXAMPLE 6. **Reading a 24-bit Raster Image from an HDF file**

The following example queries an HDF file called “example.hdf” for the dimensions and interlacing scheme of a 24-bit image. Although the **DF24getdims** function call is optional, it is included in the following example as a demonstration of how to check the image dimensions and interlacing before reading any image data. If the image dimensions and interlacing scheme are already known, only the **DF24getimage** call is required.

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6

main()
{
    uint8 raster_data[3][HEIGHT][WIDTH];
    int32 width, height;
    intn interlace;

    /* Get the image dimensions from a file */
    DF24getdims("example.hdf",&width, &height, &interlace);

    /* Read raster data if the dimensions and interlace are correct */
    if(width<=WIDTH && height<=HEIGHT && interlace==DFIL_PLANE)
        DF24getimage("example.hdf", raster_data, width, height);
}
```

FORTRAN:

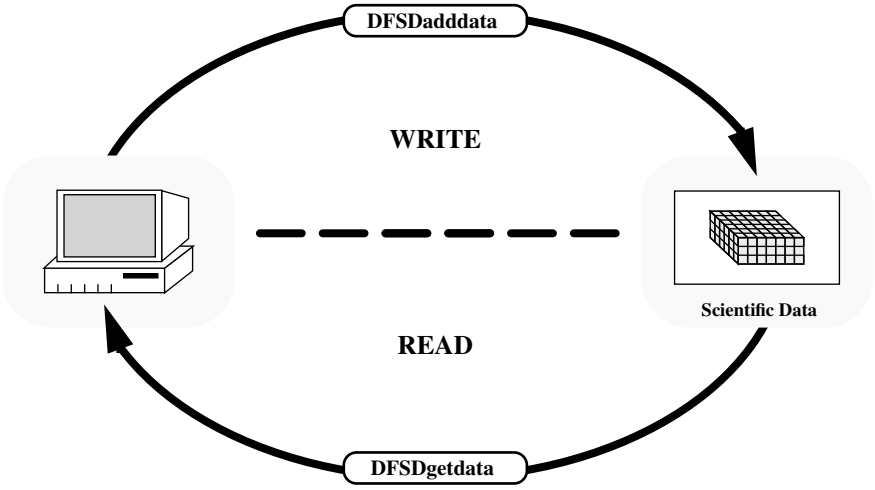
```
      PROGRAM RASTER24

      integer         d2gimg, ret, width, height, interlace
      character*1     raster_data(5,6,3)


C Read raster-24 image dimensions
      ret = d2gdims('example.hdf', width, height, interlace)

C Read raster-24 image data from file, if dimensions are correct
      if ( (width .eq. 5) .and. (height .eq. 6) ) then
          ret = d2gimg('example.hdf', raster_data, width, height)
      endif

      stop
      end
```

3.5 HDF and the Scientific Data Interface

Every function in the scientific data interface begins with the prefix ***DFSD*** which stands for "data format, scientific data." By using DFSD function calls, HDF can often make a series of assumptions about the operations you are performing. These assumptions reduce the amount of work you must do to perform a successful read or write operation.

When you make a single DFSD call to read or write scientific data, HDF assumes you are interested in reading or writing an array of 32-bit floating point data (See FIGURE 3.5a). To write the data to a file, you can use **DFSDadddata**. To read the data from a file, you must use **DFSDgetdata**.

FIGURE 3.5a **Float32 scientific data can be read or written to a file via a single read or write function call.**

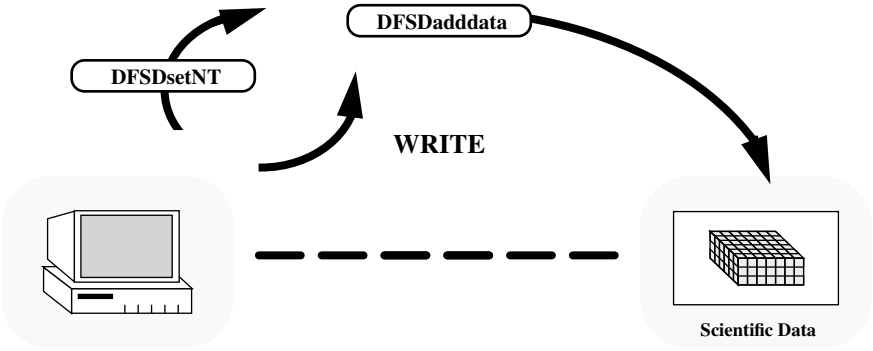


Although **DFSDadddata** and **DFSDgetdata** are the only two calls required for reading or writing 32-bit floating point data, the DFSD interface provides several additional functions for more advanced operations. For more information on additional DFSD function calls, consult the HDF User's Guide.

Writing Scientific Data to an HDF File

If you are programming in C, **DFSDadddata** is the only function call you need to write an array of 32-bit floating point data to file. In FORTRAN, the equivalent function call is **dsadata**. If you wish to store data of a different number type, you must call **DFSDsetNT** (or **dssnt** in FORTRAN) to change the default setting (See Figure 3.5b on page 34).

FIGURE 3.5b **DFSDsetNT can be used to override the DFSDadddata float32 default setting.**



If you are writing an array of 32-bit floating point data to file, the function call made from your program should look something like this:

C: `DFSDadddata (filename, rank, dimsizes, data)`
FORTRAN: `dsadata (filename, rank, dimsizes, data)`

However, if you wish to change the default number type, your program must include the following calls:

C:
```
DFSDsetNT (number_type)
DFSDadddata (filename, rank, dimsizes, data)
```

FORTRAN:
```
dssnt (number_type)
dsadata (filename, rank, dimsizes, data)
```

The parameters for both the “set number type” and the “add data” call are more explicitly defined below (See TABLE 3H). Notice that the set number type offers several alternative number types for overriding the default setting.

TABLE 3H **The parameters required for DFSDsetNT and DFSDadddata.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFSDsetNT (dssNT)	number_type	integer	32-bit integer	Specifies the number type: DFNT_FLOAT32 or 5 = 32-bit floats DFNT_FLOAT64 of 6 = 64-bit floats DFNT_INT8 or 20 = 8-bit ints DFNT_UINT8 or 21 = 8-bit unsign ints DFNT_INT16 or 22 = 16-bit ints DFNT_UINT16 or 23 = 16-bit unsign ints DFNT_INT32 or 24 = 32-bit ints DFNT_UINT32 or 25 = 32-bit unsign ints
DFSDadddata (dsadata)	filename	character*(*)	character pointer	Name of file storing the data.
	rank	integer	32-bit integer	Number of dimensions in data array.
	dimsizes	integer array	32-bit integer array	Size of each dimension in data array.
	data	<number type> array	<number type> pointer	Array holding the data.

EXAMPLE 7.

Writing Scientific Data to an HDF file

In the following code example, **DFSDadddata** and **dsadata** are used to write an array of 64-bit floating point numbers to a file called "example.hdf." Although the **DFSDsetNT** function call is optional, it is included here to demonstrate how to override the float32 default setting. In both the C and FORTRAN example, the scientific data is initialized using the following values:

1	2	3	4	5	
6	11	12	13	14	15
	16	21	22	23	24	25
		26	27	28	29	30

C:

```
#include "hdf.h"

#define LENGTH 3
#define HEIGHT 2
#define WIDTH 5

main()
{
    /* Create data array; store dimensions in array 'dims' */
    static float64 scien_data[LENGTH][HEIGHT][WIDTH] =
                                                  { 1., 2., 3., 4., 5.,
                                                    6., 7., 8., 9.,10.,

                                                   11.,12.,13.,14.,15.,
                                                   16.,17.,18.,19.,20.,

                                                   21.,22.,23.,24.,25.,
                                                   26.,27.,28.,29.,30. };
    int32 dims[3] = {LENGTH, WIDTH, HEIGHT};

    /* Set number type to 64-bit float */
    DFSDsetNT(DFNT_FLOAT64);

    /* Write the data to file */
    DFSDadddata("example.hdf", 3, dims, scien_data);
}
```

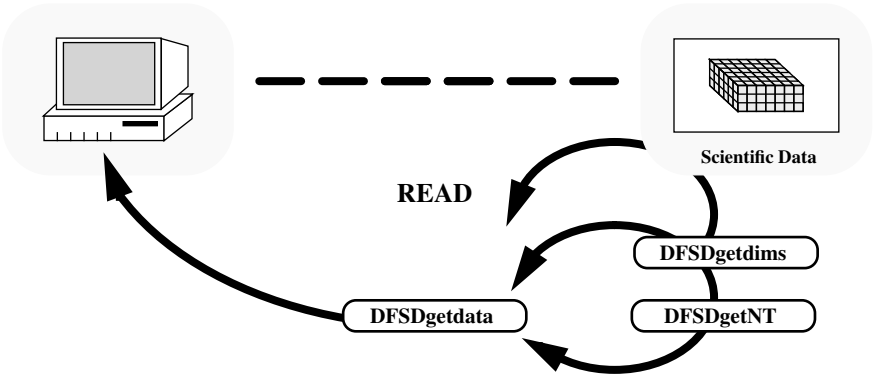
FORTRAN:

```
      PROGRAM SDS

      integer  dsadata, dssnt, dims(3), ret
      real*8   sci_data(5,2,3)

C Create data array called 'sci_data'; store dimensions in array 'dims'
      data     sci_data/             1., 2., 3., 4., 5.,
     $                               6., 7., 8., 9.,10.,
     $
     $                              11.,12.,13.,14.,15.,
     $                              16.,17.,18.,19.,20.,
     $
     $                              21.,22.,23.,24.,25.,
     $                              26.,27.,28.,29.,30./

      data dims /5,2,3/

C Set number type to 64-bit float
      ret = dssnt(6)

C Write the data to file
      ret = dsadata('example.hdf', 3, dims, sci_data)

      stop
      end
```

Reading Scientific Data from an HDF File

If you are programming in C, **DFSDgetdata** is the only function call you need to make to read a 32-bit floating point array from a file**.** In FORTRAN, the equivalent function is called **dsgdata**. If you are not sure of the dimensions or the number type for the array you intend to read, **DFSDgetdims** and **DFSDgetNT** (or **dsgnt** and **dsgdims** in FORTRAN) can obtain them for you. Both calls are optional, however **DFSDgetNT** will only work if **DFSDgetdims** is called first. By using both **DFSDgetdims** and **DFSDgetNT** in your program, you can verify both the size of the space required to hold the data as well as its number type (See FIGURE 3.5c).

FIGURE 3.5c **DF24getdims and DF24getNT will return the dimensions and number type of an array.**



If you know the dimensions and number type for the array you intend to read, the function call made from your program should look something like this:

C:
```
DFSDgetdata (filename, rank, dimsizes, maxrank)
```
FORTRAN:
```
dsgdata (filename, rank, dimsizes, maxrank)
```

However, if you wish to verify the dimensions and number type of an array, your program should include the following calls:

C:
```
DFSDgetdims (filename, rank, dimsizes, maxrank)
DFSDgetNT (number_type)
DFSDgetdata (filename, rank, dimsizes, data)
```

FORTRAN:
```
dsgNT (filename, rank, dimsizes, maxrank)
dsgdims (number_type)
dsgdata (filename, rank, dimsizes, data)
```

The first time the “get dims” and “get number type” call are made, they return information about the first array of scientific data in the named file. Any subsequent calls return the same information for successive arrays of scientific data in the file. The “get data” call operates in a similar manner. The parameters for the “get dims,” “get number type,” and the “get data” call are more explicitly defined below (See TABLE 3I on page 37).

TABLE 3I **The parameters required for DFSDgetNT, DFSDgetdims, and DFSDgetdata**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
DFSDgetdims (dsgdims)	filename	character* (*)	character pointer	Name of file containing the data.
	rank	integer	integer	Number of dimensions.
	dimsizes	integer	32-bit integer	Array holding the dimension sizes.
	maxrank	integer	integer	Size of array containing the dim sizes.
DFSDgetNT (dsgnt)	number_type	integer	32-bit integer	The number type for the data.
DFSDgetdata (dsgdata)	filename	character* (*)	character pointer	Name of file containing the data.
	rank	integer	32-bit integer	Number of dimensions.
	maxsizes	integer*4 array	32-bit integer array	Array holding the dimension sizes.
	data	<number type> array	<number type> pointer	Array to hold the scientific data.

EXAMPLE 8. **Reading an array of scientific data from an HDF file**

The following example queries a file called “example.hdf” for the dimensions and number type of an array containing scientific data. Although **DFSDgetdims** and **DFSDgetNT** are optional, they are included here as a demonstration of how to check the array dimensions and number type before reading any data. If the dimension and type are known, only the **DFSDgetdata** call is required.

C:

```
#include “hdf.h”

#define LENGTH 3
#define HEIGHT 2
#define WIDTH 5

main()
{
    float64 scien_data[LENGTH][HEIGHT][WIDTH];
    int32 number_type;
    intn rank;
    int32 dims[10];

    /* Get the dimensions and number type of the array */
    DFSDgetdims("example.hdf", &rank, dims, 10);
    DFSDgetNT(&number_type);

    /* Read the array if the dimensions are correct */
    if(dims[0] <= LENGTH && dims[1] <= HEIGHT && dims[2] <= WIDTH)
        DFSDgetdata("example.hdf", rank,dims, scien_data);
 }
```

FORTRAN:

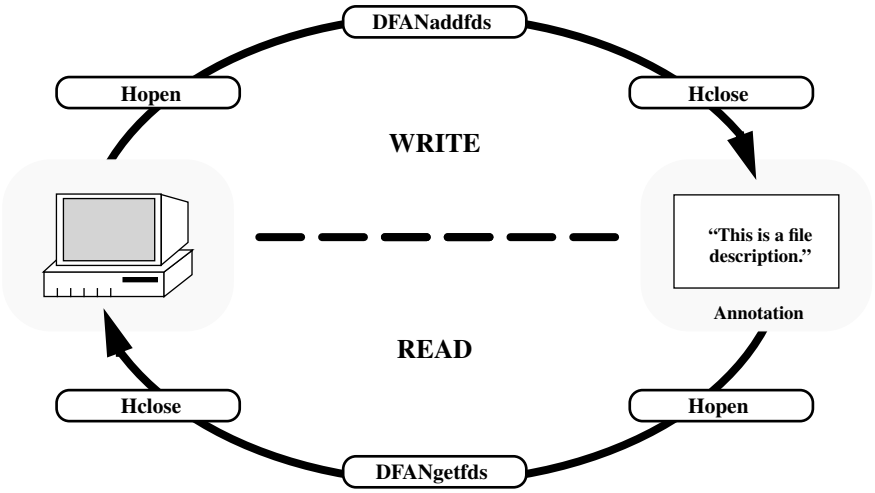
```
      PROGRAM SDS

      integer  dsgdata, dim(3), ret
      real*8   sci_data(5, 2, 3)

C Get the dimensions and number type of the array
      ret = dsgdims('example.hdf', rank, dim, 3)
      ret = dsgnt(num_type)

C Read the array if the dimensions are correct
      if((dim(1) .eq. 3) .and. (dim(2) .eq. 2) .and. (dim(3) .eq. 5)) then
          ret = dsgdata('example.hdf', 3, dims, sci_data)
      endif

      stop
      end
```

3.6 HDF and the Annotation Interface

Every function in the annotation interface begins with the prefix ***DFAN*** which stands for "data format, annotation." By using DFAN function calls, you can annotate a file with a file label (abbreviated as **fid**) or a file description (abbreviated as **fds**). Here the difference between labels and descriptions is that labels are terminated by zero or NULL, whereas descriptions are not.

Even though a reading or writing a file annotation requires only one call to the DFAN interface, both operations require two calls to the general HDF interface (See FIGURE 3.6a). Annotation operations rely on **Hopen** and **Hclose** function calls to open and close the appropriate HDF files. To begin an annotation operation, **Hopen** must open the named file and pass its corresponding file_id to the DFAN interface. Once the file is open and an annotation is written via a **DFANaddfds** call or read via a **DFANgetfds** call, **Hclose** completes the operation by closing the newly annotated file.

FIGURE 3.6a **Any HDF file can be annotated via one read or write DFAN function call.**



Although **DFANaddfds** and **DFANgetfds** are the only two DFAN calls required to read or write a file description, the DFAN interface provides several other functions for more advanced operations. For more information on annotating data elements and other useful annotation operations, consult the HDF User's Guide.

Writing a File Description

If you are programming in C, **DFANaddfds** is the only DFAN function call you will need to create a file description**.** In FORTRAN, the equivalent function is called **daafds**. Hopen must precede your "add file description" call, and Hclose must follow it. To add an annotation to a file, your program should contain a series of function calls something like this:

C:
```
Hopen (filename, access, block_size)
DFANaddfds (file_id, text, text_length)
Hclose (file_id)
```

FORTRAN:
```
hopen (filename, access, block_size)
daafds (filename, text, text_length)
hclose (file_id)
```

The parameters for the these functions are defined below (See TABLE 3J). To include additional annotations in a file, you must assign each additional annotation to an individual element. To learn more about annotating data elements, see the HDF User's Guide.

TABLE 3J **The parameters for the DFAN function calls required to write a file description.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
Hopen	filename	character*(*)	character pointer	Name of the file to be accessed.
	access	integer	integer	Operation to be performed: DFACC_READ or 1 = read only DFACC_RDWR or 3 = read/write DFACC_CREATE or 4 = create/overwrite
	block_size	integer	16-bit integer	0: if adding data to an existing file.
DFANaddfds (daafds)	file_id	integer	32-bit integer	The file identification for the opened file.
	text	character*(*)	character pointer	Text of the description.
	text_length	integer	32-bit integer	Length of the description in bytes.
Hclose	file_id	integer	32-bit integer	The file identification for the opened file.

EXAMPLE 9. **Writing a File Description to an HDF file**

The following example adds a text description to an HDF file called "example.hdf."

C:

```
#include "hdf.h"

main()
{
    int32 file_id;
    static char file_descript[] = "This is a file description";

    /* Open the HDF file to write an annotation */
    file_id = Hopen("example.hdf", DFACC_RDWR, 0);

    /* Use the file_id to write the description to a file */
    DFANaddfds(file_id, file_descript, 40);

    /* Close the file */
    Hclose(file_id);

}
```

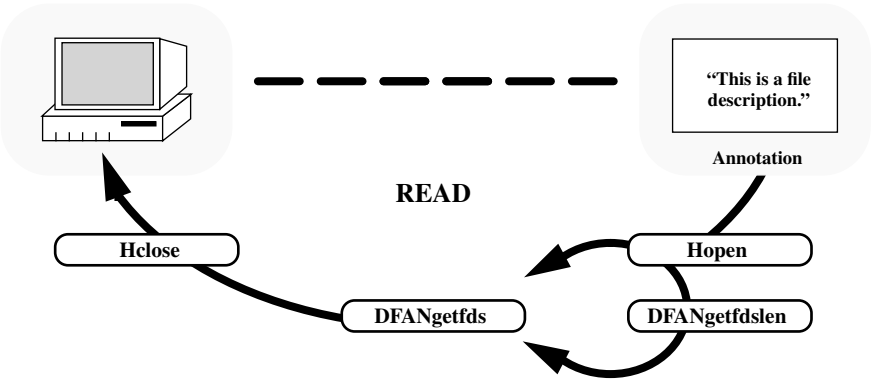
FORTRAN:

```
      PROGRAM ANNOTATION

      integer        daafds, ret, file_id, hopen, hclose, DFACC_RDWR
      character*256  f_descr

      parameter (DFACC_RDWR = 3)
      f_descr = 'This is a file description'

C Open the HDF file to write an annotation
      file_id = hopen('example.hdf', DFACC_RDWR, 0)

C Use the file_id to write the description to a file
      ret = daafds(file_id, f_descr, 40)

C Close the file
      ret = hclose(file_id)

      stop
      end
```

Reading a File Description

If you are programming in C, **DFANgetfds** is the only DFAN function call you need to read a file description**.** In FORTRAN, the equivalent function is called **dagfds**. If you are not sure of the length of the description you are attempting to read, the **DFANgetfdslen** function (or **dagfdsl** in FORTRAN) can obtain it for you. **DFANgetfdslen** is optional. However, by using **DFANgetfdslen** in your program, you can verify the size of the space required to hold the description before reading any data. In either case, Hopen must precede the "add file description" call, and Hclose must follow it (See FIGURE 3.6b).

FIGURE 3.6b **DFANgetfdslen can be used before DFANgetfds to obtain the length of an annotation.**



If you already know the length of the annotation you intend to read, the series of function calls made from your program should look something like this:

C:
```
Hopen (filename, access, block_size)
DFANgetfds (filename, description, desc_length, isfirst)
Hclose (file_id)
```

FORTRAN:
```
hopen (filename, access, block_size)
dagfds (filename, description, desc_length, isfirst)
hclose (file_id)
```

However, if you wish to verify the string length before reading the annotation, your program should include the following:

C:
```
Hopen (filename, access, block_size)
DFANgetfdslen (filename, isfirst)
DFANgetfds (filename, description, desc_length, isfirst)
Hclose (file_id)
```

FORTRAN:
```
hopen (filename, access, block_size)
dagfdsl (filename, isfirst)
dagfds (filename, description, desc_length, isfirst)
hclose (file_id)
```

The parameters for these function calls are more explicitly described in the table below (See TABLE 3K on page 41).

TABLE 3K **The parameters required for DFANgetfdslen and DFANgetfds.**

Function Call	Parameter	Variable Type		Description
		FORTRAN	C	
Hopen	filename	character*(*)	character pointer	Name of the file to be accessed.
	access	integer	integer	Operation to be performed: DFACC_READ or 1 = read only DFACC_RDWR or 3 = read/write DFACC_CREATE or 4 = create/overwrite
	block_size	integer	16-bit integer	0: if adding data to an existing file.
DFANgetfdslen (dagfdsl)	file_id	integer	32-bit integer	The file identification for the opened file.
	isfirst	integer	integer	1: start with the first annotation. 0: get the length of the next annotation.
DFANgetfds (dagfds)	file_id	integer	32-bit integer	The file identification for the opened file.
	description	character array	character array	Text of the description.
	desc_length	integer	32-bit integer	Length of the description.
	height	integer	integer	1: start with the first annotation. 0: get the length of the next annotation.
Hclose	file_id	integer	32-bit integer	The file identification for the opened file.

EXAMPLE 10.

Reading a File Description

The following example reads a file description from an HDF file called “example.hdf.” Although the **DFANgetfdslen** call is optional, it is included in the following example as a demonstration of how to check the length of a description before reading its data. If the length is already known, only the **DFANgetfds** call is required.

C:

```
#include “hdf.h”

main()
{
    int32 file_id, file_descript_len;
    char file_descript[100];

    /* Open the HDF file containing the annotation */
    file_id = Hopen("example.hdf", DFACC_READ, 0);

    /* Determine the length of the file description */
    file_descript_len=DFANfdslen(file_id,1);

    /* Read the file annotation if the length is correct */
    if(file_descrip_len<=100)
        DFANgetfds(file_id,file_descript,file_descript_len,1);

    /* Close the file */
    Hclose(file_id);
}
```

FORTRAN:

```
      PROGRAM ANNOTATE

      integer        dagfdsl, dagfds, ret, file_id, f_ds_len
      integer        hopen, hclose, DFACC_READ
      character*100  f_descr

      parameter (DFACC_READ = 1)

C Open the HDF file containing the annotation
      file_id = hopen('example.hdf', DFACC_READ, 0)

C Determine the length of the file description
      f_ds_len = dagfdsl(file_id, 1)

C Read the file annotation if the length is correct
      if ((f_ds_len .le. 100)) then
          ret = dagfds(file_id, f_descr, 100, 1)
      endif

C Close the file
      ret = hclose(fileid)

      stop
      end
```

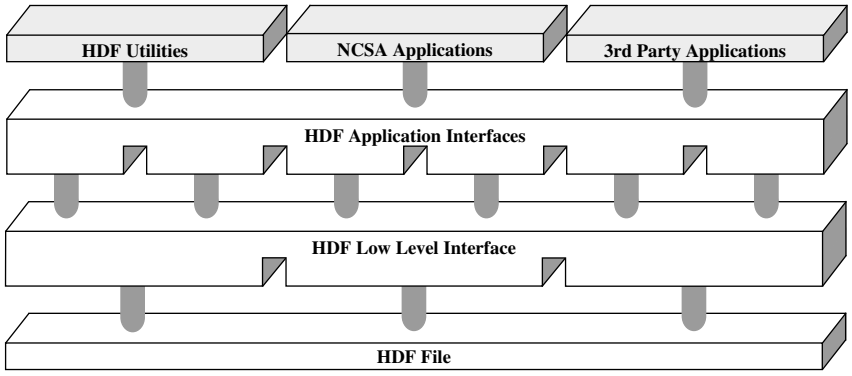
Chapter 4

HDF Utilities and Software Applications

4.0 Chapter Overview

At its highest level of interaction, HDF supports a collection of utilities, applications, and third-party commercial products (See FIGURE 4.0a). Utilities are typically short programs used to perform a single operation on an HDF file or its contents. All HDF utilities are executed through a simple "UNIX-like" command line interface. On a larger scale, NCSA applications and third-party commercial products are multi-purpose programs that provide sophisticated graphical user interfaces (GUIs) for extracting, analyzing, and storing data in HDF files.

FIGURE 4.0a **HDF supports a variety of utilities, applications and third party commercial products.**



Chapter 4 introduces the HDF utilities, demonstrates a few of their many command line options, and concludes with a table of freely available NCSA software and the data models they support.

4.1 Introducing the HDF Utilities

All ***HDF utilities*** are small application programs executed through a command line interface. In general, HDF utilities make it possible for you to perform common operations without requiring you to write your own programs.

There are currently 14 different utilities which, generally speaking, fall into three categories: inquiry utilities, conversion utilities, and compression utilities (See TABLE 4A). ***Inquiry utilities*** are primarily used to examine the contents of an HDF file. For example, the **hdfls** utility lists all the data elements contained in an HDF file. ***Conversion utilities*** include a variety of programs for converting between different HDF formats and raw data formats. Finally, the ***compression utilities*** are useful for compressing HDF files using standard compression algorithms and compacting techniques.

TABLE 4A **Utilities included in the HDF distribution.**

Type		Utility	Purpose
Inquiry		hdfls	Displays the tags and ref numbers for the data elements contained in a file.
		hdfed	Invokes and HDF editor which allows you to browse through HDF files and/ or manipulate the data elements.
		hdfseq	Displays sequences of 8-bit images directly to the screen.
Conversion	**Raw to HDF**	r8tohdf	Converts raw 8-bit images to HDF raster image sets and writes them to a file.
		paltohdf	Converts raw palettes to HDF format and writes them to file.
		r24hdf8	Converts raw 24-bit images to 8-bit raster image sets and writes them to file.
		fp2hdf	Converts 32-bit floating-point data to scientific data or 8-bit raster image sets.
	HDF to Raw	hdftor8	Converts HDF raster image sets to raw format and separates them into image files and palette files.
		hdftopal	Converts palettes in HDF files to raw format and writes them to separate files.
	HDF to HDF	ristosds	Converts several 8-bit raster images to one 3D scientific data set.
		hdf24hdf8	Converts 24-bit HDF images to 8-bit raster image sets.
		fp2hdf	Converts 32-bit floating-point data to scientific data or 8-bit raster image sets.
Compression		hdfpack	Condenses HDF files by removing the empty spaces left by deleted objects.
		hdfcomp	Compresses 8-bit raster image in an HDF file using run-length encoding or IMCOMP compression.

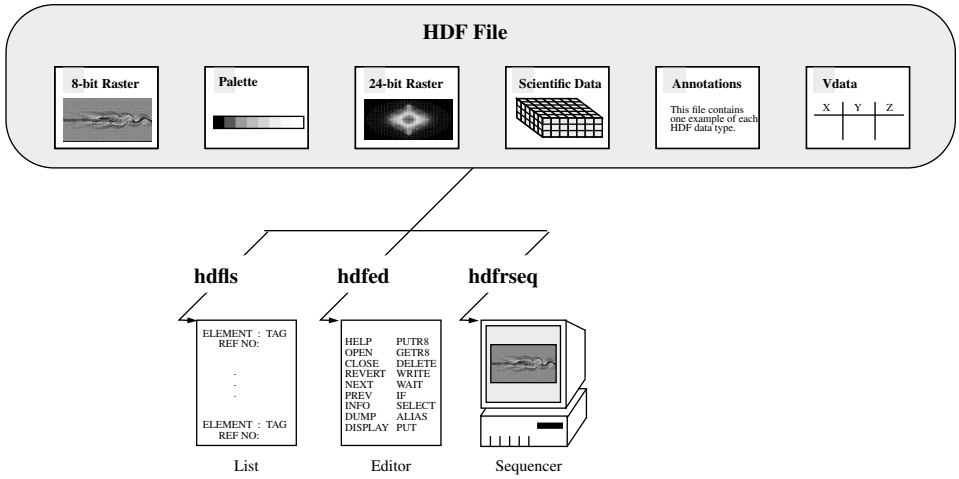
Each HDF utility supports a number of command line options. These options are exercised in a similar manner to UNIX commands, i.e. they are specified by a letter proceeded by a dash. For example, the **hdfls** utility will list the contents of an HDF file in ascending order if a −o is included on the command line. By default, hdfls lists information in descending numerical order.

In the following sections, you will explore each utility and its command line options. After reading this chapter, you will know enough about the HDF utilities to perform common operations on simple HDF files, convert data to and from the HDF format, and execute rudimentary compression operations. If, after becoming familiar with the information presented here, you need to access more information for a particular utility, consult the HDF User's Guide.

4.2 The HDF Inquiry Utilities

HDF supports three inquiry utilities (See FIGURE 4.2a). The “HDF list” utility or **hdfls** displays the contents of an HDF file by listing its tag and reference pairs. The “HDF editor” or **hdfed** provides an edit mode for examining or in some cases, modifying HDF data elements. Finally, the “HDF sequencer” or **hdfseq** extracts 8-bit images from HDF files and prints them to the local screen.

FIGURE 4.2a **HDF supports three command line inquiry utilities.**



hdfls

The **hdfls** utility lists the tag and reference number for every data element stored in the named file. A ***tag*** identifies the type of data element, and the ***reference number*** distinguishes a specific data element from the other elements with a similar tag. For example, HDF tags will differentiate raster data from Vdata, but only reference numbers can differentiate one raster element from another.

Consider a 64-bit floating point array stored as a scientific data set in a file called `example.hdf`. To display the contents of this file, enter the following command line:

```
hdfls example.hdf
```

The information printed to the screen would appear something like this:

```
example.hdf:
Version Descriptor    : (tag 30)
        Ref nos: 1
Number type           : (tag 106)
        Ref nos: 2
SciData description   : (tag 701)
        Ref nos: 2
Scientific Data       : (tag 702)
        Ref nos: 2
Numeric Data Group    : (tag 720)
        Ref nos: 2
```

At this point it is not necessary to identify the purpose of each data element in this file. Notice however, that each element is given a different tag and that reference numbers are often repeated for data elements with different tags. It is also important when retrieving data to note that reference numbers are not necessarily assigned in any order. For a more information on the form and function of data elements, their tags, and using reference numbers to retrieve data elements, see the HDF User's Guide.

hdfed

The HDF edit utility **hdfed** is an editor loosely modeled after "ed," the Unix line editor. After invoking the **hdfed** utility, the terminal screen functions as a window into an HDF file. Through this window, you can access the same information provided by **hdfls** or perform rudimentary operations on the data itself. These operations include browsing among elements in a file, displaying images, and dumping, deleting, copying or annotating data.

The **hdfed** utility is designed for users who need to manipulate HDF files at the data element level. It is not designed to provide a high-level view of HDF files. To use **hdfed** effectively requires a high level of familiarity with the format of HDF files, therefore it is not recommended for users who are unfamiliar with low level HDF components. See the HDF User's Guide for more complete instructions.

hdfseq/hdfrseq

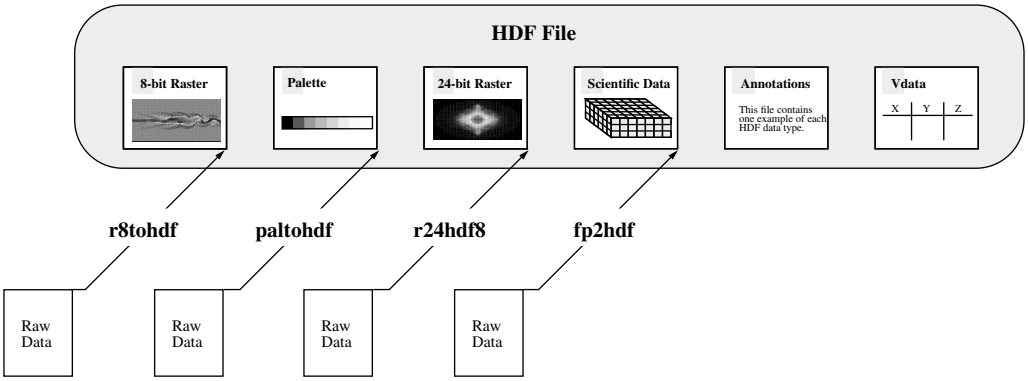
The HDF 8-bit image sequencer, otherwise known as **hdfseq**, displays one or more 8-bit images in a specified sequence. This sequencer is limited to Sun consoles and SGI's with access to the gllib library. Furthermore, all files must reside on the local machine and if more than one file is listed, **hdfseq** prints the images in the order in which their files are specified. Each new image will overwrite the preceding image on the screen.

Although **hdfseq** and **hdfrseq** perform essentially the same function, **hdfrseq** displays remote files on local Macintosh displays. The **hdfrseq** utility uses ICR, a protocol supported by NCSA MacTelnet, to transport data between machines. For more information on **hdfseq** or **hdfrseq**, consult the HDF User's Guide.

4.3 The Raw Data to HDF Conversion Utilities

HDF provides four utilities for converting data between raw data format and HDF files (See FIGURE 4.3a). The "raster 8 to HDF" utility or **r8tohdf** will convert raw two-dimensional 8-bit data into an 8-bit raster image set. The "palette to HDF" or **paltohdf** provides a similar feature for converting a raw 8-bit palette into HDF format. The "raster 24 to HDF 8" utility or **r24tohdf**8 converts raw 24-bit image data into 8-bit raster image sets. Finally, the "floating point to HDF" or **fptohdf** converts raw 32-bit floating point data into raster images and/or scientific data sets.

FIGURE 4.3a **HDF supports four Raw Data to HDF conversion utilities.**



Although each of the raw data to HDF conversion utilities are introduced here, for more detailed information on a particular utility, consult the HDF User's Guide.

r8tohdf

The r8tohdf utility converts raw 8-bit raster images into HDF raster image sets. To convert a raw raster file to an HDF file requires a command line operation similar to the following:

```
r8tohdf rows cols raster_set.hdf [-p palette_file.raw] raster_file.raw
```

Note that r8tohdf can add a palette to a raster image set if one is specified on the command line. Consider a 256 row x 512 column raw raster image stored in a file called raster.raw. Assume its palette is stored in raw form in a file called palette.raw. To create a single raster image set by including data from both files, use the following command line operation:

```
r8tohdf 256 512 raster_set.hdf -p palette.raw raster.raw
```

paltohdf

The **paltohdf** utility converts raw palette files to HDF format. The incoming palette is assumed to have 768 bytes organized in the following order: 256 red values, 256 green values, and 256 blue values. Once the palette is converted to HDF format, the palette data is interlaced as follows: RGB, RGB, ...RGB. To convert a raw palette into an HDF palette using **paltohdf**, execute a command line similar to:

```
paltohdf palette_file.raw palette_file.hdf
```

r24hdf8

The **r24hdf8** utility is useful for converting 24-bit images into 8-bit raster image sets containing palettes. The raw raster file is assumed to contain 8-bit color component values ordered in the following manner: RGB, RGB, ...RGB. To quantize a 24-bit image, execute a command line similar to the following:

```
r24hdf8 cols rows raster24_file.raw raster8_file.hdf
```

fp2hdf

The floating-point to HDF conversion utility, or **fp2hdf**, converts raw 32-bit or 64-bit floating-point data into an HDF file containing an 8-bit raster image set, a 32-bit floating point scientific data set, or both. The input data may originate from a text file, binary file or HDF file. To convert text files containing 2-dimensional data, the data must appear in the following format:

Format	Planes	Rows	Columns	Row Location	Column Location	Data
TEXT	1	10	20	row_1 row_2 . . . row_10	col_1 col_2 . . . col_20	data_1 data_2 . . . data_200

As an example of how to use fp2hdf, consider a text file containing raw 32-bit floating point data. To convert the raw file into an HDF file containing both an 8-bit raster image set and a scientific data set, the command line operation should appears as follows:

```
fp2hdf float32.raw –o float32.hdf [–r] [–f]
```

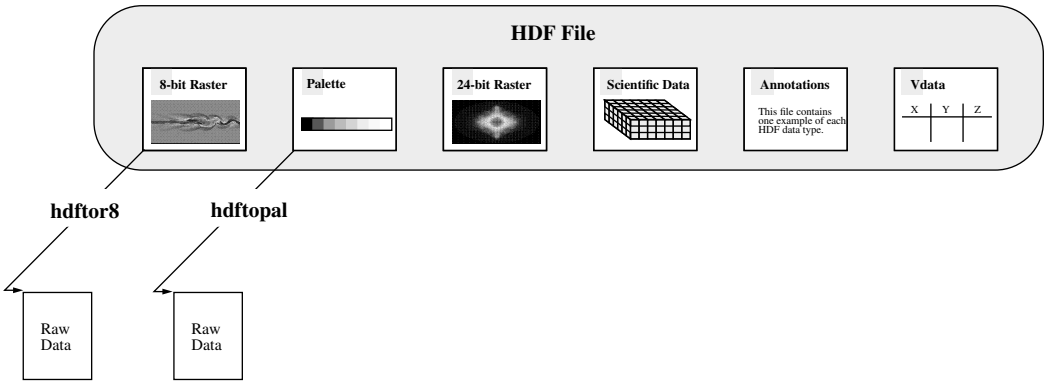
where the -r specifies the creation of a raster data set, and the -f specifies the creation of a scientific data set.

The **fp2hdf** utility also provides additional options for converting raw float-point data to HDF format. To learn more about these options, consult the HDF User's Guide or execute the command line: `fp2hdf –help`. To learn more about converting floating point data from scientific data sets to raster image sets, see Section 5.5, HDF to HDF Conversion Utilities.

4.4 The HDF to Raw Data Conversion Utilities

HDF supports two HDF to raw data conversion utilities, **hdftor8** and **hdftopal** (See FIGURE 4.4a). The **hdftor8** utility converts 8-bit raster image sets into raw data by writing the image and palette data to separate files. The **hdftopal** utility converts palettes stored in HDF files to palettes stored in raw data files. Both utilities are more completely described in the HDF User's Guide.

FIGURE 4.4a **HDF supports two HDF to Raw Data conversion utilities**



hdftor8

The **hdftor8** utility extracts raster image sets from HDF files, separates them into image data and palette data, and saves them to separate raw data files. To convert a single raster image set stored in an HDF file, execute a command line similar to the follows:

```
hdftor8 raster8_set.hdf [-r image_file.raw][-p palette_file.raw]
```

Keep in mind that when converting from HDF to raw data files the dimension record is not stored with the image data. As a result, it's best to make a note (mental or otherwise) of the image dimensions before deleting any HDF files.

hdftopal

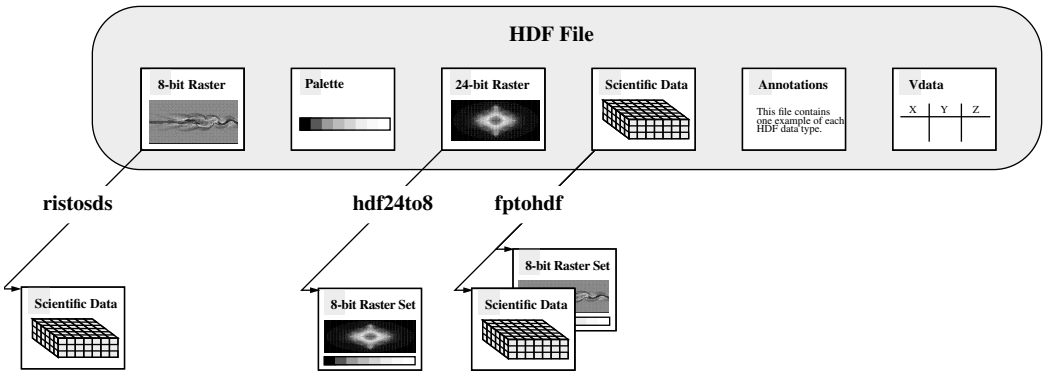
The **hdftopal** utility is used to convert palettes stored in HDF files to palettes stored in raw data files. Although HDF stores palettes by interlacing their color components, the raw file will contain a palette stored as 256 red components, followed by 256 green components, and finally 256 blue components. A typical **hdftopal** command line operation appears as follows:

```
hdftopal palette_file.hdf palette_file.raw
```

4.5 The HDF to HDF Conversion Utilities

There are three HDF to HDF conversion utilities, **ristosds**, **hdf24hdf8** and **fp2hdf** (See FIGURE 4.5a). The **ristosds** utility converts a group of raster images into a 3-dimensional scientific data set. The **hdf24hdf8** utility converts 24-bit raster image sets into 8-bit raster images using 8-bit palettes. Finally, the **fptohdf** utility converts 32-bit scientific data sets into files containing 8-bit raster image sets, scientific data sets, or both. This utility is the same utility outlined in the raw data to HDF conversion section, except that the data originates from an HDF file instead of a raw data file.

FIGURE 4.5a **HDF supports three HDF to HDF data conversion utilities.**



ristosds

The **ristosds** utility converts a series of HDF raster image files into a single HDF scientific data set. The resulting scientific data set is a 3-dimensional array whose width and height are equal to the dimensions of the raster images. Consequently, all raster images in a series must have the same dimensions. The third dimension of the array, or its length, is determined by the total number of images included in the conversion.

If the conversion is also to include a palette in the operation, the palette must appear in the first HDF file. All subsequent palettes are ignored. The command line for ristosds takes the following form:

```
ristosds raster_file.hdf {...raster_file.hdf} -o sds_file.hdf
```

Consider a directory containing twenty HDF files named storm001, storm002,... storm020. Assume each file contains a single raster image set with a 100x200 image. To combine this series of images into one scientific data set, execute the following command line:

```
ristosds storm*.hdf -o storm.hdf
```

hdf24to8

The **hdf24to8** utility quantizes a 24-bit raster image set into an 8-bit raster image set containing an 8-bit palette. All **hdf24to8** operations take the following form:

```
hdf24to8 raster24_file.hdf raster8_file.hdf
```

fp2hdf

The **fp2hdf** utility previously described in Section 4.3, can also convert 32-bit floating-point data sets into HDF files containing 8-bit raster image sets, 32-bit floating point scientific data set, or both.

As an example, consider an HDF file containing 32-bit floating point data. To convert the data into a raster image set with a palette, the command line operation appears as follows:

```
fp2hdf float32.hdf -o raster8.hdf [-r] [-p palette.hdf]
```

Notice that in order to add a palette to the raster image set, the palette data must also originate from an HDF file.

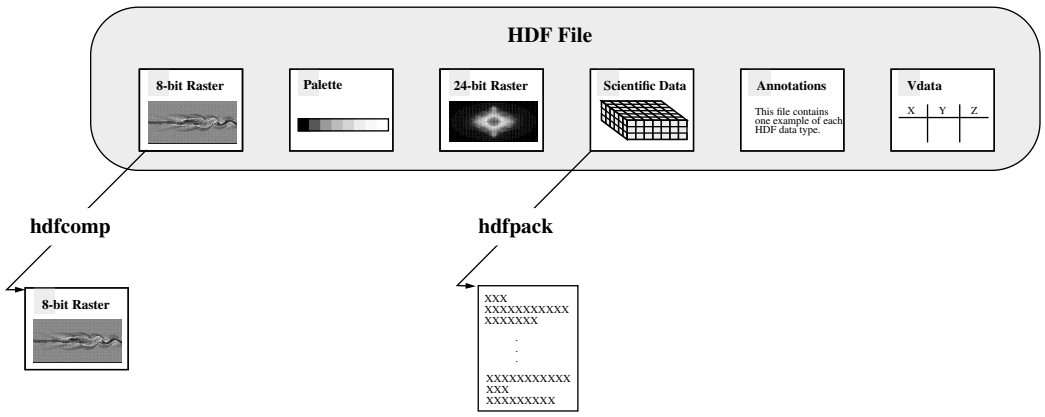
The **fp2hdf** utility provides a number of additional options for converting from float-point data to raster images. To learn more about converting HDF floating point data, see the HDF User's Guide.

4.6 The HDF Compression Utilities

There are two HDF compression utilities, **hdfpack** and **hdfcomp** (See FIGURE 4.5a). The **hdfpack** utility condenses the size of an HDF file by removing the spaces left behind by deleted data elements. The **hdfcomp** utility also condenses HDF file, but only those containing 8-bit raster image

sets. The **hdfcomp** utility compresses 8-bit raster image data using IMCOMP compression or run-length encoding.

FIGURE 4.6a **HDF supports two compression utilities.**



hdfpack

The **hdfpack** utility condenses HDF files by removing the spaces left behind by deleted data elements. It works by reading the existing data elements from the unpacked file and writing them out to a new or "packed" file. All hdfpack operations take the following form:

```
hdfpack unpacked_file.hdf packed_file.hdf
```

The **hdfpack** utility also provides a number of additional options for customizing the way HDF files are compacted. To learn more about these options, consult the HDF User's Guide.

hdfcomp

The **hdfcomp** utility reads encompassed 8-bit raster images from HDF files and writes compressed 8-bit raster images to HDF files. Compressed images are appended to the output file, if the output file exists. All **hdfcomp** command line operations take the following form:

```
hdfcomp uncompress_raster.hdf [-c | -i] compress_raster.hdf
```

where -c compresses the image using run-length encoding and -i compresses the image using IMCOMP compression. To use IMCOMP compression the file containing the uncompressed data must also contain a palette. For more information on **hdfcomp**, see the HDF User's Guide.

4.7 Introducing Applications that Support HDF

Generally speaking, there are two categories of applications that support HDF; those that are freely available through NCSA, and those that are available through commercial vendors. Of those applications available from NCSA, there are supported and unsupported applications. Supported applications are full-scale programs typically released with source code and full documentation (See TABLE). Unsupported NCSA applications are applications developed outside

NCSA. These applications are not official NCSA releases and are therefore not rigorously tested or documented. They are, however, made available through the HDF/contrib directory via the NCSA ftp server. Finally, as HDF improves, so does the outlook for more sophisticated commercial applications. Check with your software distributor for more details.

TABLE 4B **NCSA supported applications.**

Platform	Utility	Purpose	Supported Data Models	Version
Unix	**X-Image**	Displays and analyzes raster images using common color manipulation operations.	8-bit Raster Palette Scientific Data (2D only)	HDF 3.2 (or greater)
	X-DataSlice	Displays and analyzes rasters images created by slicing and scaling 3-dimensional floating point arrays..	8-bit Raster Palette Scientific Data	HDF 3.2 (or greater)
	Reformat	Converts non-HDF files (FITS, TIFF, GIF, raw raster, ASCII, SUN, and X-Windows dumps) into HDF files.	8-bit Raster Scientific Data	HDF 3.1 (or greater)
	Collage	Permits multiple platform synchronized viewing and analysis of raster images, animations, and floating-point data via a TCP/IP network connection.	8-bit Raster Palette Scientific Data (2D only) Annotation	HDF 3.2 (or greater)
SGI	**Polyview**	Creates and renders 3-dimensional objects using polygonal data.	Palette Scientific Data Vdata	HDF 3.1 (or greater)
	AlVis	Uses the alpha-shapes algorithm to create hulls from potentially unconnected points.	Palette Vdata	HDF 3.1 (or greater)
Macintosh	**Image**	Displays and analyzes of raster images using common color manipulation operations.	8-bit Raster Palette Scientific Data (2D only) Annotation	HDF 3.1 (or greater)
	DataScope	Analyzes and operates upon 2-dimensional arrays of floating-point numbers.	8-bit Raster Palette Scientific Data (2D only) Annotation	HDF 3.1 (or greater)
	PalEdit	Customizes 8-bit palettes.	Palette	HDF 3.2 (or greater)
	Layout	Arranges the screen display for photographing related images, spreadsheets, and palettes on a single canvas.	8-bit Raster Palette	HDF 3.1 (or greater)
	Import2HDF	Converts non-HDF files (FITS, TIFF, GIF, raw raster, and ASCII) into HDF files.	8-bit Raster Palette Scientific Data	HDF 3.1 (or greater)
	Collage	Synchronizes the viewing and analysis of raster images, animations, and floating-point data via a TCP/IP network connection.	8-bit Raster Palette Scientific Data (2D only) Annotation	HDF 3.2 (or greater)
IBM PC	**PC Show**	DOS application that displays 8-bit raster images using EGA, VGA or Rev512.	8-bit Raster Palette	HDF 3.2 (or greater)
	Audible Image	Displays and animates images using palettes and sound.	8-bit Raster Palette Scientific Data (2D only)	HDF 3.2 (or greater)
	Collage	MSWindows app that synchronizes the viewing and analysis of raster images, animations, and floating-point data via a TCP/IP network.	8-bit Raster Palette Scientific Data (2D only) Annotation	HDF 3.2 (or greater)

Glossary

A

annotation: a label or description specifically added to an HDF file to identify or describe a file or any data element(s) contained therein.

annotation model: a predefined data structure used to attach text strings to any HDF file or data element.

application interface: a collection of functions for reading, writing, and organizing the data elements associated with one data model.

attribute record: any one of six optional data elements describing a particular characteristic of a scientific data set.

B

bit-map: any image whose colors or grayscale intensities are stored as pixel values.

C

color component: one of three 8-bit values that determine the amount of red, green, or blue used to create a particular color.

color look-up table: the means by which each pixel value in an 8-bit image is assigned one of the 256 colors appearing in the palette.

compression utility: either of the two command line utilities capable of reducing the size of an HDF file.

conversion utility: any one of several command line utilities capable of converting data between HDF and raw data format.

D

data element: a data storage unit used by HDF to organize different types of data stored in the same file.

data model: a predefined data structure comprised of a group of related HDF data elements.

default settings: assumptions made by HDF in order to reduce the number of attributes that must be specified before performing a particular operation.

description: an annotation containing one or more text strings.

DF24: the interface prefix for the functions in the 24-bit raster interface.

DFAN: the interface prefix for the functions in the annotation interface.

DFP: the interface prefix for the functions in the palette interface.

DFR8: the interface prefix for the functions in the 8-bit raster interface.

DFSD: the interface prefix for the functions in the scientific data interface.

E

8-bit raster image: a bit-mapped image whose rows are painted from left to right using pixel values represented by a single 8-bit number.

8-bit raster image set: a group of data elements containing an 8-bit raster image, its dimension record and an optional palette.

8-bit raster model: a predefined data structure used by HDF to store an 8-bit raster image, its dimension record, and a palette in the same file.

F

ftp server: a NCSA Internet site (ftp.ncsa.uiuc.edu) that provides access to the HDF source code and documentation.

G

group element: a data element containing a list of related data elements that taken together form an HDF data set, such as a raster image set.

H

HDF data set: any predefined group of HDF data elements supported by an application interface.

HDF source distribution: any official release by NCSA of HDF source code, utilities, test routines and documentation.

HDF function call: a method of accessing a software routine contained in the HDF library by including the name of the function in a separate program.

HDF library: a collection of frequently used HDF functions that can be linked into other software application via HDF function calls.

I

inquiry utility: any one of three utilities capable of identifying, and in some cases altering, the contents of an HDF file.

interface prefix: the first few characters in each function name that identify the interface to which the function belongs.

L

label: an annotation containing one text string.

low level interface: a collection of HDF function calls that perform basic file I/O, error handling, memory management, and physical storage.

P

palette: the 256 colors used to display an 8-bit image.

palette model: a predefined data structure used by HDF to store palettes.

pixel interlacing: an interlacing scheme whereby the three color components for each pixel in a 24-bit image are grouped together and stored adjacent to one another in a file.

R

raster image: any bit-mapped image whose rows are painted from left to right, top to bottom one pixel at a time.

reference number: a number assigned to every element in order to distinguish it from other data elements of the same type.

S

scan-line interlacing: an interlacing scheme whereby the color components for each line in a 24-bit image are grouped together and stored adjacent to one another in a file.

scan-plane interlacing: an interlacing scheme whereby the color components for each color plane in a 24-bit image are grouped together and stored adjacent to one another in a file.

scientific data model: a predefined data structure used by HDF to store a multi-dimensional array, its dimension record, and its attributes in the same file.

scientific data set: a group of data elements containing an array of scientific data, its dimension record, and other optional attribute records.

T

tag: any one of a set of predefined numbers used to define the type of data stored in a data element.

24-bit raster image: any bit-mapped image whose rows are painted from left to right top to bottom using pixel values represented by three 8-bit numbers.

24-bit raster image set: a group of data elements containing a 24-bit raster image and its dimension record.

24-bit raster model: a predefined data structure used by HDf to store a 24-bit raster image and its dimension record in the same file.

V

Vdata class: a label used to describe the type of records stored in a table**.**

Vdata field name: a name assigned to a field in a Vdata record.

Vdata name: a label used to describe a Vdata table.

Vdata model: a framework for storing customized tables in HDF files.

Vdata record: a collection of fields whose number types may differ from one another.

version number: a method of labelling different versions of the HDF library which takes the form of X.YvZ where X is the major release number, Y is the minor release number, and Z is the revision number.