HDF User's Guide

The HDF Group

Table of Contents

CHAPTER 1 -- Introduction to HDF

1.1. Chapter Overview

This chapter provides a general description of HDF including its native object structures, application programming interface, and accompanying command-line utilities. It also provides a short discussion of HDF's original purpose and philosophy, the information about supported platforms, and a brief discussion on HDF4 versus HDF5.

1.2. What is HDF?

The ***Hierarchical Data Format***, or ***HDF***, is a multiobject file format for sharing scientific data in a distributed environment. HDF was created at the National Center for Supercomputing Applications, and is now developed and maintained by The HDF Group, to serve the needs of diverse groups of scientists working on projects in various fields. HDF was designed to address many requirements for storing scientific data, including:

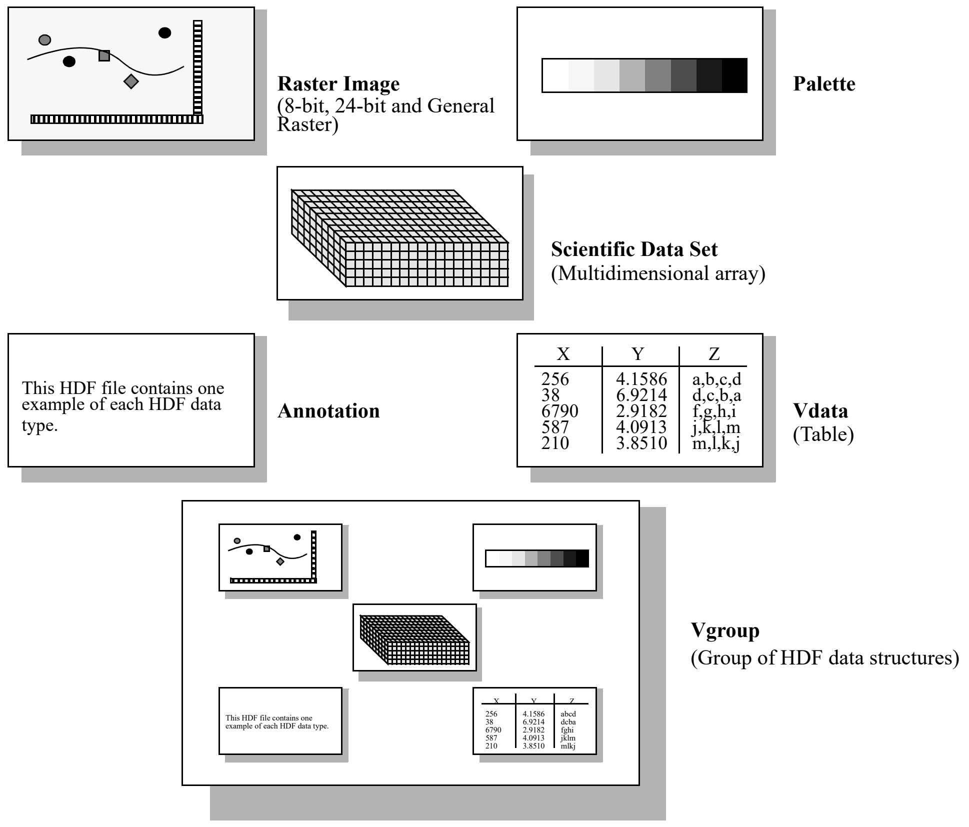
- Support for the types of data and metadata commonly used by scientists.
- Efficient storage of and access to large data sets.
- Platform independence.
- Extensibility for future enhancements and compatibility with other standard formats.

In this document, the term ***HDF data structures*** will be used to describe the primary constructs HDF provides to store data. These constructs include raster image, palette, scientific data set, annotation, vdata, and vgroup. They are illustrated in Figure 1a. Note that the construct vgroup is designed for the purpose of grouping HDF data structures.

HDF files are ***self-describing***. The term "self-description" means that, for each HDF data structure in a file, there is comprehensive information about the data and its location in the file. This information is often referred to as ***metadata***. Also, many types of data can be included within an HDF file. For example, it is possible to store symbolic, numerical and graphical data within an HDF file by using appropriate HDF data structures.

FIGURE 1a

HDF Data Structures

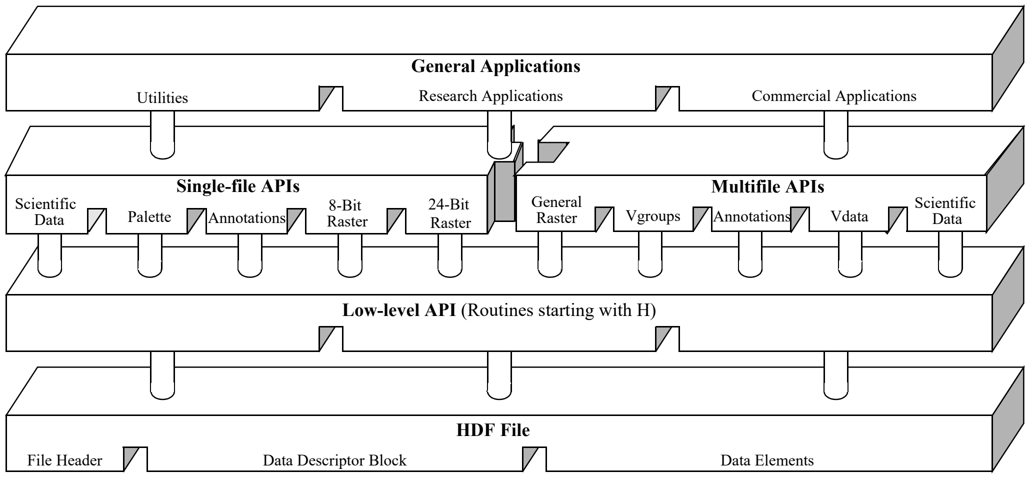


HDF can be viewed as several interactive levels. At its lowest level, HDF is a physical file format for storing scientific data. At its highest level, HDF is a collection of utilities and applications for manipulating, viewing, and analyzing data stored in HDF files. Between these levels, HDF is a software library that provides high-level and low-level programming interfaces. It also includes supporting software that make it easy to store, retrieve, visualize, analyze, and manage data in HDF files. See Figure 1b for an illustration of the interface levels.

The basic interface layer, or the ***low-level API***, is reserved for software developers. It was designed for direct file I/O of data streams, error handling, memory management, and physical storage. It is a software toolkit for experienced HDF programmers who wish to make HDF do something more than what is currently available through the higher-level interfaces. Low-level routines are available only in C.

The HDF ***application programming interfaces***, or ***APIs***, include several independent sets of routines, with each set specifically designed to simplify the process of storing and accessing one type of data. These interfaces are represented in Figure 1b as the second layer from the top. Although each interface requires programming, all the low-level details can be ignored. In most cases, all one must do is make the correct function call at the correct time, and the interface will take care of the rest. Most HDF interface routines are available in both FORTRAN-77 and C. A complete list of the high-level interfaces is provided in Section *High-Level HDF APIs*.

FIGURE 1b **Three Levels of Interaction with the HDF File**



On the highest level, ***general applications***, HDF includes various ***command-line utilities*** for managing and viewing HDF files, several ***research applications*** that support data visualization and analysis, and a variety of ***third-party applications***. The HDF utilities are included in the HDF distribution.

Source code and documentation for the HDF libraries, as well as binaries for supported platforms, is freely available but subject to the restrictions listed with the copyright notice at the beginning of this guide. This material and information regarding a variety of HDF applications is available from The HDF Group at `http://www.hdfgroup.org/products/hdf4`.

1.3. Why Was HDF Created?

Scientists commonly generate and process data files on several different machines, use various software packages to process files and share data files with others who use different machines and software. Also, they may include different kinds of information within one particular file, or within a group of files, and the mixture of these different kinds of information may vary from one file to another. Files may be conceptually related but physically separated. For example, some data may be dispersed among different files and some in program code. It is also possible that data may be related only in the scientist's conception of the data; no physical relationship may exist.

HDF addresses these problems by providing a general-purpose file structure that:

- Provides the mechanism for programs to obtain information about the data in a file from within the file, rather than from another source.
- Lets the user store mixtures of data from different sources into a single file as well as store the data and its related information in separate files, even when the files are processed by the same application program.
- Standardizes the formats and descriptions of many types of commonly-used data sets, such as raster images and multidimensional arrays.
- Encourages the use of a common data format by all machines and programs that produce files containing specific data.
- Can be adapted to accommodate virtually any kind of data.

1.4. High-Level HDF APIs

HDF APIs are divided into two categories: multifile interfaces (new) and single-file interfaces (old). The multifile interfaces are those that provide simultaneous access to several HDF files from within an application, which is an important feature that the single-file interfaces do not support. It is recommended that the user explore the new interfaces and their features since they are an improvement over the old interfaces. The old interfaces remain simply because of the need for backward compatibility.

The HDF I/O library consists of C and FORTRAN-77 routines for accessing objects and associated information. Although there is some overlap among object types, in most cases an API operates on data of only one type. Therefore, you need only familiarize yourself with the APIs specific to your needs to access data in an HDF file.

The following lists include all of the currently available HDF interfaces and the data that each interface supports.

The new multifile interfaces are:

SD API	Stores, manages and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes, in more than one file. It is described in Chapter 3, *Scientific Data Sets (SD API)*.
VS API	Stores, manages and retrieves multivariate data stored as records in a table. It is described in Chapter 4, *Vdatas (VS API)*.
V API	Creates groups of any primary HDF data structures. It is described in Chapter 5, *Vgroups (V API)*.
GR API	Stores, manages and retrieves raster images, their dimensions and palettes in more than one file. It can also manipulate unattached palettes in more than one file. It is described in Chapter 8, *General Raster Images (GR API)*.
AN API	Stores, manages and retrieves text used to describe a file or any of the data structures contained in the file. This interface can operate on several files at once. It is described in Chapter 10, *Annotations (AN API)*.

The old single-file interfaces are:

DFR8 API	Stores, manages and retrieves 8-bit raster images, with their dimensions and palettes in one file. It is described in Chapter 6, *8-Bit Raster Images (DFR8 API)*.
DF24 API	Stores, manages and retrieves 24-bit images and their dimensions in one file. It is described in Chapter 7, *24-bit Raster Images (DF24 API)*.
DFP API	Stores and retrieves 8-bit palettes in one file. It is described in Chapter 9, *Palettes (DFP API)*.
DFAN API	Stores, manages and retrieves text strings used to describe a file or any of the data structures contained in the file. This interface only operates on one file at a time. It is described in Chapter 11, *Single-file Annotations (DFAN API)*.
DFSD API	Stores, manages and retrieves multidimensional arrays of integer or floating-point data, along with their dimensions and attributes, in one file. It is described in Chapter 12, *Single-File Scientific Data Sets (DFSD API)*.

As these interfaces are the tools used to read and write HDF files, they are the primary focus of this manual.

In every interface, various programming examples are provided to illustrate the use of the interface routines. Both C and FORTRAN-77 versions are available. Their source code, in ASCII format, is located on the FTP servers in the subdirectory `samples/`, as mentioned in Section *What is HDF?*

Note that the goal of these examples is to illustrate the use of the interface routines; thus, for simplicity, many assumptions have been made, such as the availability or the authentication of the data. Based on these assumptions, these examples skip the verification of the returned status of each function. In practice, it is strongly recommended that the user verify the returned value of every function to ensure the reliability of the user application.

1.5. HDF Command-Line Utilities and Visualization Tools

HDF application software fall within the following three categories:

1. The FORTRAN-77 and C APIs described in Section *High-Level HDF APIs*.
2. Scientific visualization and analysis tools that read and write HDF files.
3. Command-line utilities that operate directly on HDF files.

Scientific visualization and analysis software that can read and write HDF files is available. This software includes tools such as HDFview, user-developed software, and commercial packages. The use of HDF files guarantees the interoperability of such tools. Some tools operate on raster images, others on color palettes. Some use images, others color palettes, still others data and annotations, and so forth. HDF provides the range of data types that these tools need, in a format that allows different tools with different data requirements to operate on the same files without confusion.

The HDF ***command-line utilities*** are application programs that can be executed by entering them at the command prompt, like UNIX commands. They perform common operations on HDF files for which one would otherwise have to write a program. The HDF utilities are described in detail in Chapter 15, *HDF Command-line Utilities*.

1.6. Primary HDF Platforms

The HDF library and utilities are maintained on a number of different machines and operating systems. For a complete list of the machines, operating systems (with versions), C and FORTRAN-77 compilers (also with versions), refer to section Platforms Tested in the RELEASE.txt.

1.7. HDF4 versus HDF5

Backward compatibility has always been an integral part of the design of HDF Versions 1, 2, 3, and 4 and the HDF4 library can access files from all earlier versions. This manual describes HDF4 and, to the extent appropriate, the earlier versions.

To take advantage of the capabilities of many of the more recent computing platforms and to meet the requirements of science applications that require ever-larger data sets, HDF5 had to be a completely new product, with a new format and a new library. HDF5 is conceptually related to HDF4 but incompatible; it cannot directly read or work with HDF4 files or the HDF4 library. HDF5 software and documentation are available at `https://www.hdfgroup.org/solutions/hdf5/`.

Both HDF4 and HDF5 are supported by The HDF Group, who will continue to maintain HDF4 as long as funds are available to do so. There are no plans to add any new features to HDF4, but bugs are fixed and the library is regularly built and tested on new operating system versions.

The HDF Group strongly recommends using HDF5, especially if you are a new user and are not constrained by existing applications to using HDF4. We also recommend that you consider migrating existing applications from HDF4 to HDF5 to take advantage of the improved features and performance of HDF5. Information about converting from HDF4 to HDF5 and tools to facilitate that conversion are available at `https://support.hdfgroup.org/ftp/HDF5/releases/tools/h4toh5/`.

See Section *Working with Both HDF4 and HDF5 File Formats*, for further discussions of and links to some of these tools.

Chapter 2 -- HDF Fundamentals

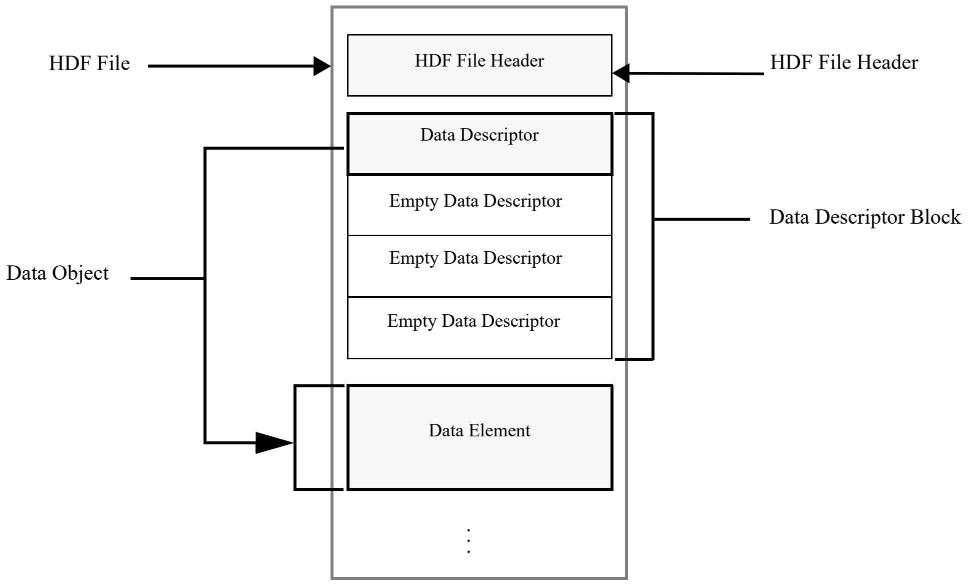
2.1. Chapter Overview

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

2.2. HDF File Format

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

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



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

2.2.1. File Header

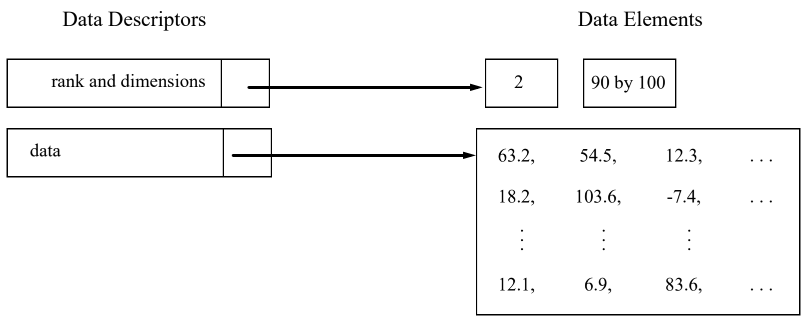
The first component of an HDF file is the file header, which takes up the first four bytes of the HDF file. Specifically, it consists of four one-byte values that are ASCII representations of control characters: the first is a control-N, the second is a control-C, the third is a control-S, and the fourth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

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

2.2.2. Data Object

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

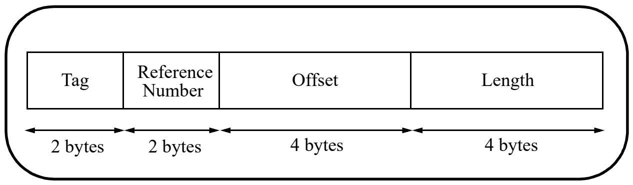
FIGURE 2b Two Data Objects



2.2.2.1. Data Descriptor

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

FIGURE 2c The Contents of a Data Descriptor



Tag

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

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

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

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

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

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

A list of commonly-used tags and their descriptions is included in Appendix A, *Reserved HDF Tags* of this document.

Reference Number

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

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

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

Data Offset and Length

The data offset field points to the location of the data element in the file by storing the number of bytes from the beginning of the file to the beginning of the data element. The length field contains the size of the data element in bytes. The data offset and the length are both 32-bit signed integers. This results in a file-size limit of 2 gigabytes.

2.2.2.2. Data Elements

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

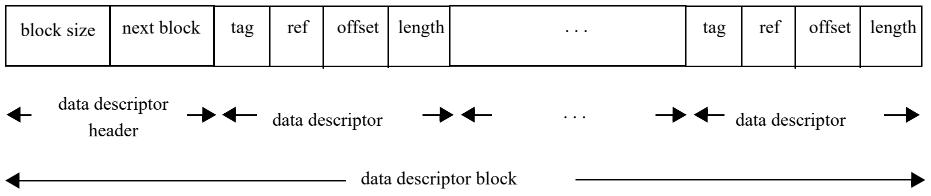
2.2.3. Data Descriptor Block

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

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

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

FIGURE 2d Data Descriptor Block



2.2.4. Grouping Data Objects in an HDF File

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

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

Additional information about data objects, including the options available for storing them, can be found in the *HDF Specifications and Developer's Guide* from the HDF web site at http://www.hdf-group.org/doc.html.

2.3. Basic Operations on HDF Files Using the Multifile Interfaces

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

2.3.1. File Identifiers

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

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

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

2.3.2. Opening HDF Files: Hopen

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

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

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

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

TABLE 2A Hopen Parameter List

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

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

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

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

Prior to HDF 4.2r2, the maximum number of open files was limited to 32, but it now can be up to what the system allowed, minus a few for stdout, etc.

It has been reported that opening/closing file in loops is very slow; thus, it is not recommended to perform such operations too many times, particularly, when data is being added to the file between opening/closing.

Note that, in the SD interface, SDstart is used to open files instead of Hopen. To access a file that contains both SD API objects and non-SD API objects, the application must call SDstart/SDend and Hopen/Hclose on the file. The non-SD API functions access the file via the identifier returned by Hopen and the SD API functions use the identifier returned by SDstart. These identifiers must be released by Hclose and SDend, respectively. Refer to Chapter 3, *Scientific Data Sets (SD API)*, of this document for more information on SDstart/SDend.

TABLE 2B File Access Code Flags

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

2.3.3. Closing HDF Files: Hclose

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

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

FORTRAN:    status = hclose(file_id)
```

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

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

TABLE 2C Hclose Parameter List

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

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

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

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

The syntax of these routines is as follows:

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

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

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

TABLE 2D Hgetlibversion and Hgetfileversion Parameter Lists

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

2.4. Determining whether a File Is an HDF File: Hishdf/hishdff

The Hishdf routine is used to determine whether the file filename is an HDF file. The Hishdf syntax is as follows:

```
C:          status = Hishdf(filename)

FORTRAN:    status = hishdff(filename)
```

This routine returns a value of TRUE (or 1) if if the file is an HDF file or FALSE (or 0) otherwise.

TABLE 2E Hishdf/hishdff Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Hishdf [intn] (hishdff)	filename	char*	character*(*)	Filename

2.5. Programming Issues

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

2.5.1. Header File Information

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

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

2.5.2. HDF Definitions

The HDF library provides several sets of definitions which can be used easily in the user applications. These sets include the definitions of the data types, the data type flags, and the limits that set various maximum values. The definitions of the data types supported by HDF are located in the hdf.h header file, and the data type flags are located in the hntdefs.h header file. Both are also included in (See Table 2F on page 14), (See Table 2G on page 15), and (See Table 2H on page 16). HDF data types are used for portability in the declaration of variables, and data type flags are used as parameters in various HDF interface routines.

2.5.2.1. Standard HDF Data Types

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

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

TABLE 2F Standard HDF Data Types and Flags

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

Fortran programmers should refer to Section *FORTRAN-77 and C Language Issues* for a discussion of the Fortran data types.

2.5.2.2. Native Format Data Types

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

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

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

TABLE 2G Native Format Data Type Definitions

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

2.5.2.3. Little-Endian Data Types

HDF normally writes data in big-endian format, but provides a little-endian option forcing all data written to disk to be written in little-endian format. This is primarily for users of Intel-based machines who do not want to incur the cost of reordering data when writing to an HDF file. Note that direct conversions are supported between little-endian and all other byte-order formats supported by HDF.

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

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

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

TABLE 2H Little-Endian Format Data Type Definitions

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

2.5.2.4. Tag Definitions

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

2.5.2.5. Limit Definitions

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

TABLE 2I

Limit Definitions

Definition Name	Definition Value	Description
FIELDNAMELENMAX	128	Maximum length of a vdata field in bytes - 128 characters
H4_MAX_NC_ATTRS	3000	Maximum number of file or variable attributes
H4_MAX_NC_DIMS	5000	Maximum number of dimensions per file
H4_MAX_NC_NAME	256	Maximum length of a name - NC interface
H4_MAX_NC_OPEN	MAX_FILE	Maximum number of files can be open at the same time
H4_MAX_NC_VARS	5000	Maximum number of variables per file
H4_MAX_VAR_DIMS	32	Maximum number of dimensions per variable
MAXNVELT	64	Maximum number of objects in a vgroup
MAX_FIELD_SIZE	65535	Maximum length of a field
MAX_FILE	32	Maximum number of open files
MAX_ORDER	65535	Maximum order of a vdata field
MAX_PATH_LEN	1024	Maximum length of an external file name
MAX_GROUPS	8	Maximum number of groups
MAX_GR_NAME	256	Maximum length of a name - GR interface
MAX_REF	65535	The largest number that will fit into a 16-bit word reference variable
MAX_BLOCK_SIZE	65536	Maximum size of blocks in linked blocks
VSNAMELENMAX	64	Maximum length of a vdata name in bytes - 64 characters
VGNAMELENMAX	64	Maximum length of a vgroup name in bytes - 64 characters
VSFIELDMAX	256	Maximum number of fields per vdata (64 for Macintosh)
VDEFAULTBLKSIZE	4096	Default block size in a vdata
VDEFAULTNBLKS	32	Default number of blocks in a vdata

2.5.3. FORTRAN-77 and C Language Issues

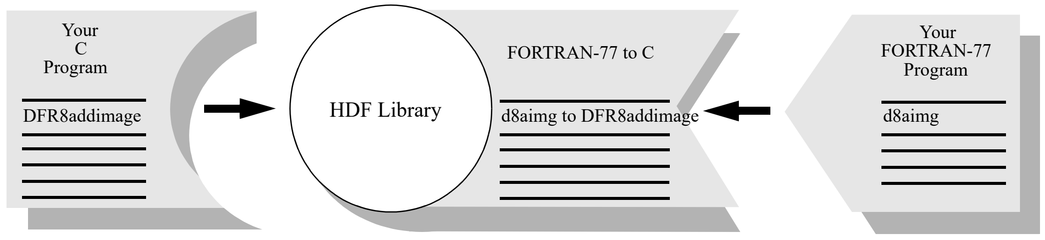
HDF provides both FORTRAN-77 and C versions of most of its interface routines. In order to make the FORTRAN-77 and C versions of each routine as similar as possible, some compromises have been made in the process of simplifying the interface for both programming languages.

FORTRAN-77-to-C Translation

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

FIGURE 2e

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



Case Sensitivity

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

Name Length

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

Header Files

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

Data Type Specifications

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

TABLE 2J Correspondence Between Fortran and HDF C Data Types

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

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

String and Array Specifications

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

- *character*(*)* defines a string of an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold the data to be stored in the string.

- *real x(*)* means that *x* refers to an array of reals of indefinite size and of indefinite rank. It is the responsibility of the calling program to allocate an actual array with the correct number of dimensions and dimension sizes.
- *<valid numeric data type> x* means that *x* may have one of the numeric data types listed in the Description column of Table 2J.
- *<valid data type> x* means that *x* may have any of the data types listed in the Description column of Table 2J.

FORTRAN-77 and ANSI C

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

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

CHAPTER 3 -- Scientific Data Sets (SD API)

3.1. Chapter Overview

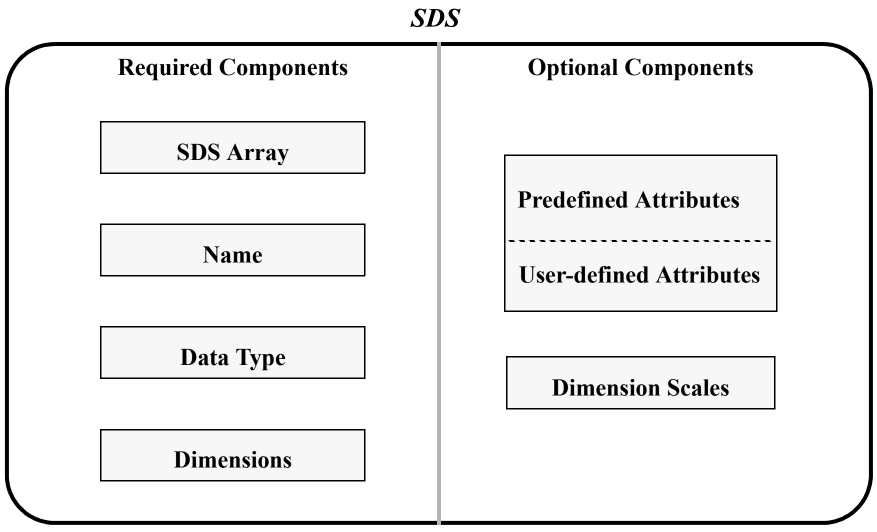
This chapter describes the scientific data model and the interface routines provided by HDF for creating and accessing the data structures included in the model. This interface is known as the SD interface or the SD API.

3.2. The Scientific Data Set Data Model

The scientific data set, or SDS, is a group of data structures used to store and describe multidimensional arrays of scientific data. Refer to Figure 3a for a graphical overview of the SD data set. Note that in this chapter the terms ***SDS***, ***SD data set***, and ***data set*** are used interchangeably; the terms ***SDS array*** and ***array*** are also used interchangeably.

A scientific data set consists of required and optional components, which will be discussed in the following subsections.

FIGURE 3a **The Contents of a Scientific Data Set**



3.2.1. Required SDS Components

Every SDS must contain the following components: an ***SDS array***, a ***name***, a ***number type***, and the ***dimensions*** of the SDS, which are actually the dimensions of the SDS array.

SDS Array

An ***SDS array*** is a multidimensional data structure that serves as the core structure of an SDS. This is the primary data component of the SDS model and can be compressed (refer to Section "*Compressing SDS Data: SDsetcompress"* for a description of SDS compression) and/or stored in external files (refer the Section "*Creating a Data Set with Data Stored in an External File: SDsetexternalfile"* for a description of external SDS storage). Users of netCDF should note that SDS arrays are conceptually equivalent to ***variables*** in the netCDF data model[1].

An SDS has an index and a reference number associated with it. The ***index*** is a non-negative integer that describes the relative position of the data set in the file. A valid index ranges from 0 to the total number of data sets in the file minus 1. The ***reference number*** is a unique positive integer assigned to the data set by the SD interface when the data set is created. Various SD interface routines can be used to obtain an SDS index or reference number depending on the available information about the SDS. The index can also be determined if the sequence in which the data sets are created in the file is known.

In the SD interface, an ***SDS identifier*** uniquely identifies a data set within the file. The identifier is created by the SD interface access routines when a new SDS is created or an existing one is selected. The identifier is then used by other SD interface routines to access the SDS until the access to this SDS is terminated. For an existing data set, the index of the data set can be used to obtain the identifier. Refer to Section "*Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect"* for a description of the SD interface routine that creates SDSs and assigns identifiers to them.

SDS Name

The ***name*** of an SDS can be provided by the calling program, or is set to "DataSet" by the HDF library at the creation of the SDS. The name consists of case-sensitive alphanumeric characters, is assigned only when the data set is created, and cannot be changed. SDS names do not have to be unique within a file, but their uniqueness makes it easy to semantically distinguish among data sets in the file.

Number Type

The data contained in an SDS array has a ***number type*** associated with it. The standard types supported by the SD interface include 32- and 64-bit floating-point numbers, 8-, 16- and 32-bit signed integers, 8-, 16- and 32-bit unsigned integers, and 8-bit characters. The SD interface also allows the creation of SD data sets consisting of data elements of non-standard lengths (1 to 32 bits). See Section "*Creating SDS Arrays Containing Non-standard Length Data: SDsetnbitdataset"* for more information.

Dimensions

SDS ***dimensions*** specify the shape and size of an SDS array. The number of dimensions of an array is referred to as the ***rank*** of the array. Each dimension has an index and an identifier assigned to it. A dimension also has a size and may have a name associated with it.

1. *netCDF-3 User's Guide for C* (June 5, 1997), Section 7, `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/`.

A dimension ***identifier*** is a positive number uniquely assigned to the dimension by the library. This dimension identifier can be retrieved via an SD interface routine. Refer to Section "*Selecting a Dimension: SDgetdimid"* for a description of how to obtain dimension identifiers.

A dimension ***index*** is a non-negative number that describes the ordinal location of a dimension among others in a data set. In other words, when an SDS dimension is created, an index number is associated with it and is one greater than the index associated with the last created dimension that belongs to the same data set. The dimension index is convenient in a sequential search or when the position of the dimension among other dimensions in the SDS is known.

The ***size*** of a dimension is a positive integer. Also, one dimension of an SDS array can be assigned the predefined size `SD_UNLIMITED` (or `0`). This dimension is referred to as an ***unlimited dimension***, which, as the name suggests, can grow to any length. Refer to Section "*Appending Data to an SDS Array along an Unlimited Dimension"* for more information on unlimited dimensions.

Names can optionally be assigned to dimensions, however, dimension names are not treated in the same way as SDS array names. For example, if a name assigned to a dimension was previously assigned to another dimension the SD interface treats both dimensions as the same data component and any changes made to one will be reflected in the other.

Important Note:

HDF4 allows a dimension and a one-dimensional SDS to be given the same name. The library also stores a dimension and a data set the same way internally. Prior to HDF 4.2.2, however, the library did not adequately distinguish these two types of objects. Thus, when a dimension and a one-dimensional SDS shared a name, writing to the SDS or the dimension could cause data corruption to the other. The corrupted data was unrecoverable.

This problem was fixed in Release 4.2.2 and such data corruption will not occur in files created with a 4.2.2 or later library. Note, however, that the fix is effective only in new files; a dimension and a one-dimensional SDS of the same name that were created with a pre-4.2.2 HDF4 Library remain vulnerable to data corruption if an application is unaware of the potential conflict. To safely handle pre-4.2.2 files, the library now provides two functions, **SDgetnumvars_byname** and **SDnametoindices**. **SDgetnumvars_byname** can be used to determine whether a name is unique. If the function reports one ('1') variable by that name, the name is unique and no further precaution needs to be taken. If the name is not unique, i.e., the number of variables by that name is greater than one, **SDnametoindices** must then be used to retrieve the index and the type of each variable with that name. The desired variable can then be safely selected via its index. These functions are described in detail in this User's Guide and the HDF4 Reference Manual.

A similar problem is possible when a multi-dimensional SDS and a dimension are created with the same name by a pre-4.2.2 library. The HDF Group has not seen such a failure, however, and it is thought to be very unlikely. Note that the fix introduced in Release 4.2.2 also prevents data corruption from happening for this situation even though the data was created with libraries prior to 4.2.2, assuming no corruption had yet occurred.

3.2.2. Optional SDS Components

There are three types of optional SDS components: ***user-defined attributes***, ***predefined attributes***, and ***dimension scales***. These optional components are only created when specifically requested by the calling program.

Attributes describe the nature and/or the intended usage of the file, data set, or dimension they are attached to. Attributes have a name and value which contains one or more data entries of the same type. Thus, in addition to name and value, the number type and number of values are specified when the attribute is created.

User-Defined Attributes

User-defined attributes are defined by the calling program and contain auxiliary information about a file, SDS array, or dimension. They are more fully described in Section "*User-defined Attributes"*.

Predefined Attributes

Predefined attributes have reserved names and, in some cases, predefined number types and/or number of data entries. Predefined attributes are useful because they establish conventions that applications can depend on. They are further described in Section "*Predefined Attributes"*.

Dimension Scales

A dimension scale is a sequence of numbers placed along a dimension to demarcate intervals along it. Dimension scales are described in Section "*Dimension Scales"*.

3.2.3. Annotations and the SD Data Model

In the past, annotations were supported in the SD interface to allow the HDF user to attach descriptive information (called ***metadata***) to a data set. With the expansion of the SD interface to include user-defined attributes, the use of annotations to describe metadata should be eliminated. Metadata once stored as an annotation is now more conveniently stored as an attribute. However, to ensure backward compatibility with scientific data sets and applications relying on annotations, the AN annotation interface, described in Chapter 10, *Annotations (AN API)* can be used to annotate SDSs.

There is no cross-compatibility between attributes and annotations; creating one does not automatically create the other.

3.3. The SD Interface

The SD interface provides routines that store, retrieve, and manipulate scientific data using the SD data model. The SD interface supports simultaneous access to more than one SDS in more than one HDF file. In addition, the SD interface is designed to support a general scientific data model which is very similar to the netCDF data model developed by the Unidata Program Center[1].

For those users who have been using the DFSD interface, the SD interface provide a model compatible with that supported by the DFSD interface. It is recommended that DFSD users apply the SD model and interface to their applications since the DFSD interface is less flexible and less powerful than the SD interface and will eventually be removed from the HDF library.

This section specifies the header file to be used with the SD interface and lists all available SD interface routines, each of which is accompanied by its purpose and the section where the routine is discussed.

3.3.1. Header Files Required by the SD Interface

The `mfhdf.h` header file must be included in programs that invoke SD interface routines. FORTRAN-77 users should refer to Section "*FORTRAN-77 and C Language Issues"*.

1. *netCDF-3 User's Guide for C* (June 5, 1997), Section 2, `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/`.

3.3.2. SD Interface Routines

All C routines in the SD interface begin with the prefix "SD". The equivalent FORTRAN-77 routines use the prefix "sf". These routines are categorized as follows:

- ***Access routines*** initialize and terminate access to HDF files and data sets.
- ***Read and write routines*** read and write data sets.
- ***General inquiry routines*** return information about the location, contents, and description of the scientific data sets in an HDF file.
- ***Dimension routines*** access and define characteristics of dimensions within a data set.
- ***Dimension scale routines*** define and access dimension scales within a data set.
- ***User-defined attribute routines*** create and access user-defined attributes of an HDF file, data set, or dimension.
- ***Predefined attribute routines*** access previously-defined attributes of an HDF file, data set, or dimension.
- ***Compression routines*** compress SDS data and retrieves compresion information.
- ***Chunking/tiling routines*** manage chunked data sets.
- ***Miscellaneous routines*** provide other operations such as external file, n-bit data set, and compatibility operations.
- ***Raw Data Information routines*** provide information that allows applications to read raw data from HDF files without the use of HDF library. These functions are described in Chapter 16, *Raw Data Information* of this document, together with the same type of routines that belong to other interfaces.

The SD routines are listed in the following table and are discussed in the following sections of this chapter.

TABLE 3A

SD Interface Routines

Category	Routine Name		Description and Reference
	C	FORTRAN-77	
Access	`SDstart`	`sfstart`	Opens the HDF file and initializes the SD interface (Section "*Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect"*)
	`SDcreate`	`sfcreate`	Creates a new data set (Section "*Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect"*)
	`SDselect`	`sfselect`	Selects an existing SDS given its index (Section "*Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect"*)
	`SDendaccess`	`sfendacc`	Terminates access to an SDS (Section "*Terminating Access to Files and Data Sets: SDendaccess and SDend"*)
	`SDend`	`sfend`	Terminates access to the SD interface and closes the file (Section "*Terminating Access to Files and Data Sets: SDendaccess and SDend"*)
Read and Write	`SDreaddata`	`sfrdata/` `sfrcdata`	Reads data from a data set (Section "*Reading Data from an SDS Array: SDreaddata"*)
	`SDwritedata`	`sfwdata/` `sfwcdata`	Writes data to a data set (Section "*Writing Data to an SDS Array: SDwritedata"*)

General Inquiry	SDcheckempty	sfchempty	Determines whether a scientific dataset (an SDS) is empty (Section "*Determining whether an SDS is empty: SDcheckempty"*)
	SDfileinfo	sffinfo	Retrieves information about the contents of a file (Section "*Obtaining Information about the Contents of a File: SDfileinfo"*)
	SDgetfilename	sfgetfname	Given a file identifier, retrieves the name of the file (Section "*Obtaining the Name of a File: SDgetfilename"*)
	SDgetinfo	sfginfo	Retrieves information about a data set (Section "*Obtaining Information about a Specific SDS: SDgetinfo"*)
	SDget_maxopen-files	sfgmaxopenf	Retrieves current and maximum number of open files (Section "*Obtaining Current Limits on Opened Files: SDget_maxopen-files"*)
	SDgetnamelen	sfgetnamelen	Retrieves the length of the name of a file, a dataset, or a dimension (Section "*Obtaining the Length of an HDF4 Object's Name: SDgetnamelen"*)
	SDget_numopen-files	sfgnumopenf	Returns the number of files currently open (Section "*Obtaining Number of Opened Files: SDget_numopenfiles"*)
	SDgetnumvars_by-name	sfgnvars_by-name	Retrieves the number of data sets having the same name (Section "*Getting Number of Data Sets Given a Name: SDgetnumvars_by-name"*)
	SDidtoref	sfid2ref	Returns the reference number of a data set (Section "*Obtaining the Reference Number Assigned to the Specified SDS: SDidtoref"*)
	SDidtype	sfidtype	Given an identifier, returns the type of object the identifier represents (Section "*Obtaining the Type of an HDF4 Object: SDidtype"*)
	SDiscoordvar	sfiscvar	Distinguishes data sets from dimension scales (Section "*Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar"*)
	SDisrecord	sfisrcrd	Determines whether a data set is appendable, i.e., having unlimited dimension (Section "*Determining whether an SDS Array is Appendable: SDisrecord"*)
	SDnametoindex	sfn2index	Returns the index of a data set specified by its name (Section "*Locating an SDS by Name: SDnametoindex"*)
	SDnametoindices	sfn2indices	Retrieves a list of indices of data sets having the same given name (Section "*Locating More Than One SDS by the Same Name: SDnametoindices"*)
	SDreftoindex	sfref2index	Returns the index of a data set specified by its reference number (Section "*Locating an SDS by Reference Number: SDreftoindex"*)
	SDreset_maxopen-files	sfrmaxopenf	Resets the maximum number of files that can be open at the same time (Section "*Resetting the Allowed Number of Opened Files: SDreset_maxopenfiles"*)
Dimensions	SDdiminfo	sfgdinfo	Gets information about a dimension (Section "*Obtaining Dimension Scale and Other Dimension Information: SDdiminfo"*)
	SDgetdimid	sfdimid	Returns the identifier of a dimension (Section "*Selecting a Dimension: SDgetdimid"*)
	SDsetdimname	sfsdimname	Associates a name with a dimension (Section "*Naming a Dimension: SDsetdimname"*)
Dimension Scales	SDgetdimscale	sfgdscale	Retrieves the scale values for a dimension (Section "*Reading Dimension Scales: SDgetdimscale"*)
	SDsetdimscale	sfsdscale	Stores the scale values of a dimension (Section "*Writing Dimension Scales: SDsetdimscale"*)
User-defined Attributes	SDattrinfo	sfgainfo	Gets information about an attribute (Section "*Querying User-defined Attributes: SDfindattr and SDattrinfo"*)
	SDfindattr	sffattr	Returns the index of an attribute specified by its name (Section "*Querying User-defined Attributes: SDfindattr and SDattrinfo"*)
	SDreadattr	sfrnatt/sfr-catt	Reads the values of an attribute specified by its index (Section "*Reading User-defined Attributes: SDreadattr"*)
	SDsetattr	sfsnatt/sfs-catt	Creates a new attribute and stores its values (Section "*Creating or Writing User-defined Attributes: SDsetattr"*)

Predefined Attributes	`SDgetcal`	`sfgcal`	Retrieves calibration information (Section "*Reading Calibrated Data: SDgetcal"*)
	`SDgetdatastrs`	`sfgdtstr`	Returns the predefined-attribute strings of a data set (Section "*Reading String Attributes of an SDS: SDgetdatastrs"*)
	`SDgetdimstrs`	`sfgdmstr`	Returns the predefined-attribute strings of a dimension (Section "*Reading a String Attribute of a Dimension: SDgetdimstrs"*)
	`SDgetfillvalue`	`sfgfill/sfgc-fill`	Reads the fill value if it exists (Section "*Reading a Fill Value Attribute: SDgetfillvalue"*)
	`SDgetrange`	`sfgrange`	Retrieves the range of values in the specified data set (Section "*Reading a Range Attribute: SDgetrange"*)
	`SDsetcal`	`sfscal`	Defines the calibration information (Section "*Setting Calibration Information: SDsetcal"*)
	`SDsetdatastrs`	`sfsdtstr`	Sets predefined attributes of the specified data set (Section "*Writ-ing String Attributes of an SDS: SDsetdatastrs"*)
	`SDsetdimstrs`	`sfsdmstr`	Sets predefined attributes of the specified dimension (Section "*Writing a String Attribute of a Dimension: SDsetdimstrs"*)
	`SDsetfillvalue`	`sfsfill/sfsc-fill`	Defines the fill value for the specified data set (Section "*Writing a Fill Value Attribute: SDsetfillvalue"*)
	`SDsetfillmode`	`sfsflmd`	Sets the fill mode to be applied to all data sets in the specified file (Section "*Setting the Fill Mode for all SDSs in the Specified File: SDsetfillmode"*)
	`SDsetrange`	`sfsrange`	Defines the maximum and minimum values of the specified data set (Section "*Writing a Range Attribute: SDsetrange"*)
Compression	`SDsetcompress`	`sfscompress`	Compresses a data set using a specified compression method (Sec-tion "*Compressing SDS Data: SDsetcompress"*)
	`SDsetnbitdataset`	`sfsnbit`	Defines the non-standard bit length of the data set data (Section "*Creating SDS Arrays Containing Non-standard Length Data: SDsetnbitdataset"*)
	`SDgetcompinfo`	`sfgcompress`	Retrieves data set compression type and compression information (Section "*Obtaining Data Set Compression Information: SDget-compinfo"*)
Chunking/ Tiling	`SDgetchunkinfo`	`sfgichnk`	Obtains information about a chunked data set (Section "*Obtaining Information about a Chunked SDS: SDgetchunkinfo"*)
	`SDreadchunk`	`sfrchnk/ sfrcchnk`	Reads data from a chunked data set (Section "*Reading Data from Chunked SDSs: SDreadchunk and SDreaddata"*)
	`SDsetchunk`	`sfschnk`	Makes a non-chunked data set a chunked data set (Section "*Mak-ing an SDS a Chunked SDS: SDsetchunk"*)
	`SDsetchunkcache`	`sfcchnk`	Sets the size of the chunk cache (Section "*Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache"*)
	`SDwritechunk`	`sfwchnk/sfw-cchnk`	Writes data to a chunked data set (Section "*Writing Data to Chun-ked SDSs: SDwritechunk and SDwritedata"*)
Raw Data Information	`SDgetanndatainfo`	`unvailable`	Retrieves location and size of annotations' data (Section "*Retriev-ing Data Information of an Annotation in SD API: SDgetann-datainfo"*)
	`SDgetattdatainfo`	`unvailable`	Retrieves location and size of an attribute's data (Section "*Retriev-ing Data Information of an Attribute: SDgetattdatainfo"*)
	`SDgetdatainfo`	`unvailable`	Retrieves location and size of data blocks in a specified data set (Section "*Retrieving Data Information of an SDS: SDgetdatainfo"*)
	`SDgetoldat-tdatainfo`	`unvailable`	Retrieves location and size of an old predefined attribute's data (Section "*Determining the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp"*)

Miscellaneous	`SDgetexternalinfo`	`unvailable`	Gets information about external file of a data set (Section "*Getting External File Information of a Data Set: SDgetexternalinfo"*)
	`SDsetblocksize`	`sfsblsz`	Sets the block size used for storing data sets with unlimited dimension (Section "*Setting the Block Size: SDsetblocksize"*)
	`SDsetexternalfile`	`sfsextf`	Specifies that a data set is to be stored in an external file (Section "*Creating a Data Set with Data Stored in an External File: SDsetexternalfile"*)
	`SDisdimval_bwcomp`	`sfisdmvc`	Determines the current compatibility mode of a dimension (Section "*Determining the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp"*)
	`SDsetdimval_comp`	`sfsdmvc`	Sets the future compatibility mode of a dimension (Section "*Setting the Future Compatibility Mode of a Dimension: SDsetdimval_comp"*)
	`SDsetaccesstype`	`sdfsacct`	Sets the I/O access type for an SDS (Section "*Setting the I/O Access Type of an SDS: SDsetaccesstype"*)

3.3.3. Tags in the SD Interface

A complete list of SDS tags and their descriptions appears in Table AD in Appendix A. Refer to Section "*Data Descriptor"* for a description of tags.

3.4. Programming Model for the SD Interface

This section describes the routines used to initialize the SD interface, create a new SDS or access an existing one, terminate access to that SDS, and shut down the SD interface. Writing to existing scientific data sets will be described in Section "*Writing Data to an SDS"*.

To support multifile access, the SD interface relies on the calling program to initiate and terminate access to files and data sets. The SD programming model for creating and accessing an SDS in an HDF file is as follows:

1. Open a file and initialize the SD interface.
2. Create a new data set or open an existing one using its index.
3. Perform desired operations on this data set.
4. Terminate access to the data set.
5. Terminate access to the SD interface and close the file.

To access a single SDS in an HDF file, the calling program must contain the following calls:

```
C:         sd_id = SDstart(filename, access_mode);

           sds_id = SDcreate(sd_id, sds_name, ntype, rank, dim_sizes);
    OR     sds_id = SDselect(sd_id, sds_index);

           <Optional operations>
           status = SDendaccess(sds_id);
           status = SDend(sd_id);

FORTRAN:   sd_id = sfstart(filename, access_mode)

           sds_id = sfcreate(sd_id, sds_name, ntype, rank, dim_sizes)
    OR     sds_id = sfselect(sd_id, sds_index)

           <Optional operations>
           status = sfendacc(sds_id)
           status = sfend(sd_id)
```

If the file contains non-SD-API objects, such as vdatas or raster images, the application must use **Hopen**/**Hclose** to access these objects while **SDstart**/**SDend** the SD-API objects. The non-SD API functions access the file via the identifier returned by **Hopen** and the SD API functions use the identifier returned by **SDstart**.

To access several files at the same time, a program must obtain a separate SD file identifier (*sd_id*) for each file to be opened. Likewise, to access more than one SDS, a calling program must obtain a separate SDS identifier (*sds_id*) for each SDS. For example, to open two SDSs stored in two files a program would execute the following series of function calls.

```
C:        sd_id_1 = SDstart(filename_1, access_mode);
          sds_id_1 = SDselect(sd_id_1, sds_index_1);
          sd_id_2 = SDstart(filename_2, access_mode);
          sds_id_2 = SDselect(sd_id_2, sds_index_2);
          <Optional operations>
          status = SDendaccess(sds_id_1);
          status = SDend(sd_id_1);
          status = SDendaccess(sds_id_2);
          status = SDend(sd_id_2);

FORTRAN:  sd_id_1 = sfstart(filename_1, access_mode)
          sds_id_1 = sfselect(sd_id_1, sds_index_1)
          sd_id_2 = sfstart(filename_2, access_mode)
          sds_id_2 = sfselect(sd_id_2, sds_index_2)
          <Optional operations>
          status = sfendacc(sds_id_1)
          status = sfend(sd_id_1)
          status = sfendacc(sds_id_2)
          status = sfend(sd_id_2)
```

3.4.1. Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect

In the SD interface, **SDstart** is used to open files rather than **Hopen**. **SDstart** takes two arguments, *filename* and *access_mode*, and returns the SD interface identifier, *sd_id*. Note that the SD interface identifier, *sd_id*, is *not* interchangeable with the file identifier, *file_id*, created by **Hopen** and used in other HDF APIs.

The argument *filename* is the name of an HDF or netCDF file.

The argument *access_mode* specifies the type of access required for operations on the file. All the valid values for *access_mode* are listed in Table 3B. If the file does not exist, specifying `DFACC_READ` or `DFACC_WRITE` will cause **SDstart** to return a `FAIL` (or `-1`). Specifying `DFACC_CREATE` creates a new file with read and write access. If `DFACC_CREATE` is specified and the file already exists, the contents of this file will be replaced.

TABLE 3B **File Access Code Flags**

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

The SD interface identifiers can be obtained and discarded in any order and all SD interface identifiers must be individually discarded, by **SDend**, before the termination of the calling program.

Although it is possible to open a file more than once, it is recommended that the appropriate access mode be specified and **SDstart** called only once per file. Repeatedly calling **SDstart** on the same file and with different access modes may cause unexpected results. Note that it has been reported that opening/closing file in loops is very slow; thus, it is not recommended to perform such operations too many times, particularly, when data is being added to the file between opening/closing.

Prior to HDF 4.2.2, the maximum number of open files was limited to 32; but, it now can be up to what the system allowed.

SDstart returns an SD identifier or a value of `FAIL` (or `-1`). The parameters of **SDstart** are defined in Table 3C.

SDcreate defines a new SDS using the arguments *sd_id*, *sds_name*, *ntype*, *rank*, and *dim_sizes* and returns the data set identifier, *sds_id*.

The parameter *sds_name* is a character string containing the name to be assigned to the SDS. The SD interface will generate a default name, "`DataSet`", for the SDS, if one is not provided, i.e., when the parameter *sds_name* is set to `NULL` in C, or an empty string in FORTRAN-77. The maximum length of an SDS name is no longer limited to 64 characters, starting in HDF 4.2.2. Applications should use the API **SDgetnamelen** in order to allocate sufficient space when reading the name. Note that when an older version of the library reads a data set, which was created by a library of version 4.2.2 or later and has the name that is longer than 64 characters, the retrieved name will contain some garbage after 64 characters.

The parameter *ntype* is a defined name, prefaced by `DFNT`, and specifies the type of the data to be stored in the data set. The header file "hntdefs.h" contains the definitions of all valid number types, which are described in Chapter 2, *HDF Fundamentals*, and listed in Table 2F.

The parameter *rank* is a positive integer specifying the number of dimensions of the SDS array. The maximum rank of an SDS array is defined by `H4_MAX_VAR_DIMS` (or `32`), which is defined in the header file "hlimits.h". Note that, in order for HDF4 and NetCDF models to work together, HDF allows SDS to have rank 0. However, there is no intention for data to be written to this type of SDS, but only to store attribute as part of the data description. Consequently, setting compression and setting chunk are disallowed.

Each element of the one-dimensional array *dim_sizes* specifies the length of the corresponding dimension of the SDS array. The size of *dim_sizes* must be the value of the parameter *rank*. To create a data set with an unlimited dimension, assign the value of `SD_UNLIMITED` (or `0`) to *dim_sizes[0]* in C, and to *dim_sizes(rank)* in FORTRAN-77. See the notes regarding the potential performance impact of unlimited dimension data sets in Section "*Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance*".

Once an SDS is created, you cannot change its name, number type, size, or shape. However, it is possible to modify the data set’s data or to create an empty data set and later add values. To add data or modify an existing data set, use **SDselect** to get the data set identifier instead of **SDcreate**.

Note that the SD interface retains no definitions about the size, contents, or rank of an SDS from one SDS to the next, or from one file to the next.

SDselect initiates access to an existing data set. The routine takes two arguments: *sd_id* and *sds_index* and returns the SDS identifier *sds_id*. The argument *sd_id* is the SD interface identifier returned by **SDstart**, and *sds_index* is the position of the data set in the file. The argument *sds_index* is zero-based, meaning that the index of first SDS in the file is 0.

Similar to SD interface identifiers, SDS identifiers can be obtained and discarded in any order as long as they are discarded properly. Each SDS identifier must be individually disposed of, by **SDendaccess**, before the disposal of the identifier of the interface in which the SDS is opened.

SDcreate and **SDselect** each returns an SDS identifier or a value of `FAIL` (or `-1`). The parameters of **SDstart, SDcreate,** and **SDselect** are further described in Table 3C.

3.4.2. Terminating Access to Files and Data Sets: SDendaccess and SDend

SDendaccess terminates access to the data set and disposes of the data set identifier *sds_id*. The calling program must make one **SDendaccess** call for every **SDselect** or **SDcreate** call made during its execution. Failing to call **SDendaccess** for each call to **SDselect** or **SDcreate** may result in a loss of data.

SDend terminates access to the file and the SD interface and disposes of the file identifier *sd_id*. The calling program must make one **SDend** call for every **SDstart** call made during its execution. Failing to call **SDend** for each **SDstart** may result in a loss of data.

SDendaccess and **SDend** each returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDendaccess** and **SDend** are further described in Table 3C.

TABLE 3C

SDstart, SDcreate, SDselect, SDendaccess, and SDend Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDstart [int32] **(sfstart)**	filename	char *	character*(*)	Name of the HDF or netCDF file
	access_mode	int32	integer	Type of access
SDcreate [int32] **(sfcreate)**	sd_id	int32	integer	SD interface identifier
	sds_name	char *	character*(*)	ASCII string containing the name of the data set
	ntype	int32	integer	Number type of the data set
	rank	int32	integer	Number of dimensions in the array
	dim_sizes	int32[]	integer(*)	Array defining the size of each dimension
SDselect [int32] **(sfselect)**	sd_id	int32	integer	SD interface identifier
	sds_index	int32	integer	Position of the data set within the file
SDendaccess [intn] **(sfendacc)**	sds_id	int32	integer	Data set identifier
SDend [intn] **(sfend)**	sd_id	int32	integer	SD interface identifier

EXAMPLE 1.

Creating an HDF file and an Empty SDS.

This example illustrates the use of **SDstart/sfstart**, **SDcreate/sfcreate**, **SDendaccess/sfendacc**, and **SDend/sfend** to create the HDF file named SDS.hdf, and an empty data set with the name SDStemplate in the file.

Note that the Fortran program uses a transformed array to reflect the difference between C and Fortran internal data storages. When the actual data is written to the data set, SDS.hdf will contain the same data regardless of the language being used.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"
#define SDS_NAME      "SDStemplate"
#define X_LENGTH      5
#define Y_LENGTH      16
#define RANK          2  /* Number of dimensions of the SDS */

main( )
{
   /************************* Variable declaration **************************/

   int32 sd_id, sds_id;     /* SD interface and data set identifiers */
   int32 dim_sizes[2];      /* sizes of the SDS dimensions */
   intn  status;            /* status returned by some routines; has value
                               SUCCEED or FAIL */

   /******************** End of variable declaration **********************/

   /*
   * Create the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_CREATE);
   /*
   * Define the dimensions of the array to be created.
   */
   dim_sizes[0] = Y_LENGTH;
   dim_sizes[1] = X_LENGTH;

   /*
   * Create the data set with the name defined in SDS_NAME. Note that
   * DFNT_INT32 indicates that the SDS data is of type int32. Refer to
   * Table 2E for definitions of other types.
   */
   sds_id = SDcreate (sd_id, SDS_NAME, DFNT_INT32, RANK, dim_sizes);

   /*
   * Terminate access to the data set.
   */
   status = SDendaccess (sds_id);

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  create_SDS
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*11 SDS_NAME
      integer      X_LENGTH, Y_LENGTH, RANK
      parameter   (FILE_NAME = 'SDS.hdf',
     +             SDS_NAME = 'SDStemplate',
     +             X_LENGTH = 5,
     +             Y_LENGTH = 16,
     +             RANK     = 2)
      integer      DFACC_CREATE, DFNT_INT32
      parameter   (DFACC_CREATE = 4,
     +             DFNT_INT32 = 24)
C
C     Function declaration.
C
      integer sfstart, sfcreate, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, dim_sizes(2)
      integer status
C
C**** End of variable declaration ************************************
C
C
C     Create the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_CREATE)
C
C     Define dimensions of the array to be created.
C
      dim_sizes(1) = X_LENGTH
      dim_sizes(2) = Y_LENGTH
C
C     Create the array with the name defined in SDS_NAME.
C     Note that DFNT_INT32 indicates that the SDS data is of type
C     integer. Refer to Tables 2E and 2I for the definition of other types.
C
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT32, RANK,
     .                  dim_sizes)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.5. Writing Data to an SDS

An SDS can be written partially or entirely. Partial writing includes writing to a contiguous region of the SDS and writing to selected locations in the SDS according to patterns defined by the user. This section describes the routine **SDwritedata** and how it can write data to part of an SDS or to an entire SDS. The section also illustrates the concepts of compressing SDSs and using external files to store scientific data.

3.5.1. Writing Data to an SDS Array: SDwritedata

SDwritedata can completely or partially fill an SDS array or append data along the dimension that is defined to be of unlimited length (see Section "*Appending Data to an SDS Array along an Unlimited Dimension"* for a discussion of unlimited-length dimensions). It can also skip a specified number of SDS array elements between write operations along each dimension.

To write to an existing SDS, the calling program must contain the following sequence of routine calls:

```
C:          sds_id = SDselect(sd_id, sds_index);
            status = SDwritedata(sds_id, start, stride, edges, data);

FORTRAN:    sds_id = sfselect(sd_id, sds_index)

            status = sfwdata(sds_id, start, stride, edges, data)

      OR    status = sfwcdata(sds_id, start, stride, edges, data)
```

To write to a new SDS, simply replace the call **SDselect** with the call **SDcreate**, which is described in Section "*Establishing Access to Files and Data Sets: SDstart, SDcreate, and SDselect"*.

SDwritedata takes five arguments: *sds_id*, *start*, *stride*, *edges*, and *data*. The argument *sds_id* is the data set identifier returned by **SDcreate** or **SDselect**.

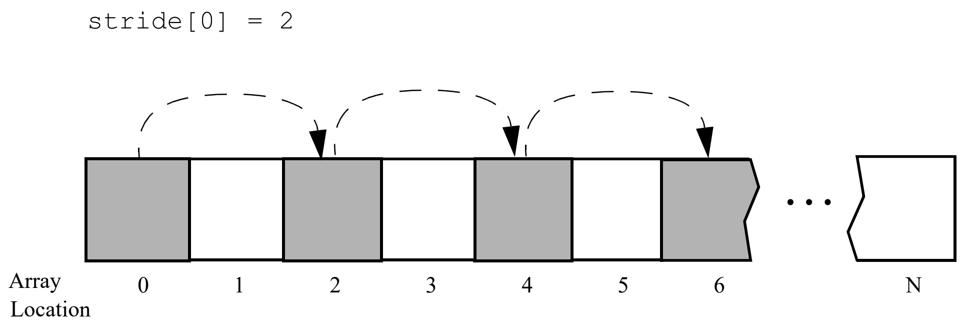
Before proceeding with the description of the remaining arguments, an explanation of the term ***hyperslab*** (or ***slab***, as it will be used in this chapter) is in order. A ***slab*** is a group of SDS array elements *that are stored in consecutive locations.* It can be of any size and dimensionality as long as it is a subset of the array, which means that a single array element and the entire array can both be considered slabs. A slab is defined by the multidimensional coordinate of its initial vertex and the lengths of each dimension.

Given this description of the slab concept, the usage of the remaining arguments should become apparent. The argument *start* is a one-dimensional array specifying the location in the SDS array at which the write operation will begin. The values of each element of the array *start* are relative to 0 in both the C and FORTRAN-77 interfaces. The size of *start* must be the same as the number of dimensions in the SDS array. In addition, each value in *start* must be smaller than its corresponding SDS array dimension unless the dimension is unlimited. Violating any of these conditions causes **SDwritedata** to return `FAIL`.

The argument *stride* is a one-dimensional array specifying, for each dimension, the interval between values to be written. For example, setting the first element of the array *stride* equal to 1 writes data to every location along the first dimension. Setting the first element of the array *stride* to 2 writes data to every other location along the first dimension. Figure 3b illustrates this example, where the shading elements are written and the white elements are skipped. If the argument *stride* is set to `NULL` in C (or either `0` or `1` in FORTRAN-77), **SDwritedata** operates as if every element of *stride* contains a value of 1, and a contiguous write is performed. For better performance, it is recommended that the value of *stride* be defined as `NULL` (i.e., `0` or `1` in FORTRAN-77) rather than being set to 1.

The size of the array *stride* must be the same as the number of dimensions in the SDS array. Also, each value in *stride* must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. Violating any of these conditions causes **SDwritedata** to return `FAIL`.

FIGURE 3b **An Example of Access Pattern ("Strides")**



The argument *edges* is a one-dimensional array specifying the length of each dimension of the slab to be written. If the slab has fewer dimensions than the SDS data set has, the size of *edges* must still be equal to the number of dimensions in the SDS array and all the elements corresponding to the additional dimensions must be set to 1.

Each value in the array *edges* must not be larger than the length of the corresponding dimension in the SDS data set unless the dimension is unlimited. Attempting to write slabs larger than the size of the SDS data set will result in an error condition.

In addition, the sum of each value in the array *edges* and the corresponding value in the *start* array must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. Violating any of these conditions causes **SDwritedata** to return `FAIL`. When **SDreaddata** returns `FAIL` (or `-1`) due to any invalid argements, the error code `DFE_ARGS` will be pushed on the stack.

The parameter *data* contains the SDS data to be written. If the SDS array is smaller than the buffer *data*, the amount of data written will be limited to the maximum size of the SDS array.

Be aware that the mapping between the dimensions of a slab and the order in which the slab values are stored in memory is different between C and FORTRAN-77. In C, the values are stored with the assumption that the last dimension of the slab varies fastest (or "row-major order" storage), but in FORTRAN-77 the first dimension varies fastest (or "column-major order" storage). These storage order conventions can cause some confusion when data written by a C program is read by a FORTRAN-77 program or vice versa.

There are two FORTRAN-77 versions of this routine: **sfwdata** and **sfwcdata**. The routine **sfwdata** writes numeric scientific data and **sfwcdata** writes character scientific data.

SDwritedata returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are described in Table 3D.

TABLE 3D **SDwritedata Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDwritedata [intn] **(sfwdata/ sfwcdata)**	sds_id	int32	integer	Data set identifier
	start	int32 []	integer(*)	Array containing the position at which the write will start for each dimension
	stride	int32 []	integer(*)	Array specifying the interval between the values that will be read along each dimension
	edges	int32 []	integer(*)	Array containing the number of data elements that will be written along each dimension
	data	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer for the data to be written

3.5.1.1. Filling an Entire Array

Filling an array is a simple slab operation where the slab begins at the origin of the SDS array and fills every location in the array. **SDwritedata** fills an entire SDS array with data when all elements of the array *start* are set to `0`, the argument *stride* is set equal to `NULL` in C or each element of the array *stride* is set to `1` in both C and FORTRAN-77, and each element of the array *edges* is equal to the length of each dimension.

EXAMPLE 2. **Writing to an SDS.**

This example illustrates the use of the routines **SDselect/sfselect** and **SDwritedata/sfwrite** to select the first SDS in the file SDS.hdf created in Example 1 and to write actual data to it.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"
#define X_LENGTH      5
#define Y_LENGTH      16

main( )
{
   /************************* Variable declaration *************************/
```

```
    int32 sd_id, sds_id, sds_index;
    intn  status;
    int32 start[2], edges[2];
    int32 data[Y_LENGTH][X_LENGTH];
    int   i, j;

    /********************* End of variable declaration ***********************/

    /*
    * Data set data initialization.
    */
    for (j = 0; j < Y_LENGTH; j++) {
        for (i = 0; i < X_LENGTH; i++)
             data[j][i] = (i + j) + 1;
    }

    /*
    * Open the file and initialize the SD interface.
    */
    sd_id = SDstart (FILE_NAME, DFACC_WRITE);

    /*
    * Attach to the first data set.
    */
    sds_index = 0;
    sds_id = SDselect (sd_id, sds_index);

    /*
    * Define the location and size of the data to be written to the data set.
    */
    start[0] = 0;
    start[1] = 0;
    edges[0] = Y_LENGTH;
    edges[1] = X_LENGTH;

    /*
    * Write the stored data to the data set. The third argument is set to NULL
    * to specify contiguous data elements. The last argument must
    * be explicitly cast to a generic pointer since SDwritedata is designed
    * to write generic data.
    */
    status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)data);

    /*
    * Terminate access to the data set.
    */
    status = SDendaccess (sds_id);

    /*
    * Terminate access to the SD interface and close the file.
    */
    status = SDend (sd_id);
}
```

FORTRAN:

```
      program  write_data
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*11 SDS_NAME
      integer      X_LENGTH, Y_LENGTH, RANK
      parameter   (FILE_NAME = 'SDS.hdf',
     +             SDS_NAME = 'SDStemplate',
     +             X_LENGTH = 5,
     +             Y_LENGTH = 16,
     +             RANK     = 2)
      integer      DFACC_WRITE, DFNT_INT32
      parameter   (DFACC_WRITE = 2,
     +             DFNT_INT32 = 24)
C
C     Function declaration.
C

      integer sfstart, sfselect, sfwdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer start(2), edges(2), stride(2)
      integer i, j
      integer data(X_LENGTH, Y_LENGTH)
C
C**** End of variable declaration ************************************
C

C
C     Data set data initialization.
C
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            data(i, j) = i + j - 1
10         continue
20    continue

C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)

C
C     Attach to the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)

C
C     Define the location and size of the data to be written
C     to the data set. Note that setting values of the array stride to 1
C     specifies the contiguous writing of data.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Write the stored data to the data set named in SDS_NAME.
C     Note that the routine sfwdata is used instead of sfwcdata
C     to write the numeric data.
C
      status = sfwdata(sds_id, start, stride, edges, data)
```

```
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.5.1.2. Writing Slabs to an SDS Array

To allow preexisting data to be modified, the HDF library does not prevent **SDwritedata** from overwriting one slab with another. As a result, the calling program is responsible for managing any overlap when writing slabs. The HDF library will issue an error if a slab extends past the valid boundaries of the SDS array. However, appending data along an unlimited dimension is allowed.

EXAMPLE 3. **Writing a Slab of Data to an SDS.**

This example shows how to fill a 3-dimensional SDS array with data by writing series of 2-dimensional slabs to it.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SLABS.hdf"
#define SDS_NAME      "FilledBySlabs"
#define X_LENGTH      4
#define Y_LENGTH      5
#define Z_LENGTH      6
#define RANK          3

main( )
{
   /************************* Variable declaration **************************/

   int32 sd_id, sds_id;
   intn  status;
   int32 dim_sizes[3], start[3], edges[3];
   int32 data[Z_LENGTH][Y_LENGTH][X_LENGTH];
   int32 zx_data[Z_LENGTH][X_LENGTH];
   int   i, j, k;

   /******************** End of variable declaration **********************/

   /*
   * Data initialization.
   */
   for (k = 0; k < Z_LENGTH; k++)
       for (j = 0; j < Y_LENGTH; j++)
           for (i = 0; i < X_LENGTH; i++)
               data[k][j][i] = (i + 1) + (j + 1) + (k + 1);

   /*
   * Create the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_CREATE);

   /*
```

```
    * Define dimensions of the array to be created.
    */
    dim_sizes[0] = Z_LENGTH;
    dim_sizes[1] = Y_LENGTH;
    dim_sizes[2] = X_LENGTH;

    /*
    * Create the array with the name defined in SDS_NAME.
    */
    sds_id = SDcreate (sd_id, SDS_NAME, DFNT_INT32, RANK, dim_sizes);

    /*
    * Set the parameters start and edges to write
    * a 6x4 element slab of data to the data set; note
    * that edges[1] is set to 1 to define a 2-dimensional slab
    * parallel to the ZX plane.
    * start[1] (slab position in the array) is initialized inside
    * the for loop.
    */
    edges[0] = Z_LENGTH;
    edges[1] = 1;
    edges[2] = X_LENGTH;
    start[0] = start[2] = 0;
    for (j = 0; j < Y_LENGTH; j++)
    {
        start[1] = j;

        /*
        * Initialize zx_data buffer (data slab).
        */
        for ( k = 0; k < Z_LENGTH; k++)
        {
            for ( i = 0; i < X_LENGTH; i++)
            {
                  zx_data[k][i] = data[k][j][i];
            }
    }

    /*
    * Write the data slab into the SDS array defined in SDS_NAME.
    * Note that the 3rd parameter is NULL which indicates that consecutive
    * slabs in the Y direction are written.
    */
    status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)zx_data);
    }

    /*
    * Terminate access to the data set.
    */
    status = SDendaccess (sds_id);

    /*
    * Terminate access to the SD interface and close the file.
    */
    status = SDend (sd_id);
}
```

FORTRAN:

```
      program  write_slab
      implicit none
C
C     Parameter declaration.
```

```
C
      character*9  FILE_NAME
      character*13 SDS_NAME
      integer      X_LENGTH, Y_LENGTH, Z_LENGTH, RANK
      parameter   (FILE_NAME = 'SLABS.hdf',
     +             SDS_NAME = 'FilledBySlabs',
     +             X_LENGTH = 4,
     +             Y_LENGTH = 5,
     +             Z_LENGTH = 6,
     +             RANK     = 3)
      integer      DFACC_CREATE, DFNT_INT32
      parameter   (DFACC_CREATE = 4,
     +             DFNT_INT32 = 24)
C
C     Function declaration.
C
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id
      integer dim_sizes(3), start(3), edges(3), stride(3)
      integer i, j, k, status
      integer data(X_LENGTH, Y_LENGTH, Z_LENGTH)
      integer xz_data(X_LENGTH, Z_LENGTH)
C
C**** End of variable declaration ************************************
C
C
C     Data initialization.
C
      do 30 k = 1, Z_LENGTH
         do 20 j = 1, Y_LENGTH
            do 10 i = 1, X_LENGTH
               data(i, j, k) = i + j + k
10          continue
20       continue
30    continue
C
C     Create the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_CREATE)
C
C     Define dimensions of the array to be created.
C
      dim_sizes(1) = X_LENGTH
      dim_sizes(2) = Y_LENGTH
      dim_sizes(3) = Z_LENGTH
C
C     Create the data set with the name defined in SDS_NAME.
C
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT32, RANK,
     .                  dim_sizes)
C
C     Set the parameters start and edges to write
C     a 4x6 element slab of data to the data set;
C     note that edges(2) is set to 1 to define a 2 dimensional slab
C     parallel to the XZ plane;
C     start(2) (slab position in the array) is initialized inside the
C     for loop.
C
      edges(1) = X_LENGTH
      edges(2) = 1
      edges(3) = Z_LENGTH
      start(1) = 0
      start(3) = 0
      stride(1) = 1
      stride(2) = 1
      stride(3) = 1

      do 60 j = 1, Y_LENGTH
       start(2) = j - 1
C
C     Initialize the buffer xz_data (data slab).
C
       do 50 k = 1, Z_LENGTH
        do 40 i = 1, X_LENGTH
         xz_data(i, k) = data(i, j, k)
40      continue
50     continue
C
C     Write the data slab into SDS array defined in SDS_NAME.
C     Note that the elements of array stride are set to 1 to
C     specify that the consecutive slabs in the Y direction are written.
C
         status = sfwdata(sds_id, start, stride, edges, xz_data)
60    continue
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

EXAMPLE 4.

Altering Values within an SDS Array.

This example demonstrates how the routine **SDwritedata** can be used to alter the values of the elements in the 10th and 11th rows, at the 2nd column, in the SDS array created in the Example 1 and written in Example 2. FORTRAN-77 routine **sfwdata** is used to alter the elements in the 2nd row, 10th and 11th columns, to reflect the difference between C and Fortran internal storage.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"

main( )
{
   /************************* Variable declaration *************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 start[2], edges[2];
   int32 new_data[2];
   int   i, j;

   /******************** End of variable declaration **********************/
   /*
   * Open the file and initialize the SD interface with write access.
   */
   sd_id = SDstart (FILE_NAME, DFACC_WRITE);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Set up the start and edge parameters to write new element values
   * into 10th row, 2nd column place, and 11th row, 2nd column place.
   */
   start[0] = 9;     /* starting at 10th row   */
   start[1] = 1;     /* starting at 2nd column */
   edges[0] = 2;     /* rows 10th and 11th     */
   edges[1] = 1;     /* column 2nd only        */

   /*
   * Initialize buffer with the new values to be written.
   */
   new_data[0] = new_data[1] = 1000;

   /*
   * Write the new values.
   */
   status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)new_data);

   /*
   * Terminate access to the data set.
   */
   status = SDendaccess (sds_id);

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  alter_data
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      integer      DFACC_WRITE
      parameter   (FILE_NAME = 'SDS.hdf',
     +             DFACC_WRITE = 2)
C
C     Function declaration.
C
      integer sfstart, sfselect, sfwdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index
      integer start(2), edges(2), stride(2)
      integer status
      integer new_data(2)
C
C**** End of variable declaration ************************************
```

```
C

C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Select the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)

C
C     Initialize the start, edge, and stride parameters to write
C     two elements into 2nd row, 10th column and 11th column places.
C
C     Specify 2nd row.
C
      start(1) = 1
C
C     Specify 10th column.
C
      start(2) = 9
      edges(1) = 1
C
C     Two elements are written along 2nd row.
C
      edges(2) = 2
      stride(1) = 1
      stride(2) = 1
C
C     Initialize the new values to be written.
C
      new_data(1) = 1000
      new_data(2) = 1000
C
C     Write the new values.
C
      status = sfwdata(sds_id, start, stride, edges, new_data)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.5.1.3. Appending Data to an SDS Array along an Unlimited Dimension

An SDS array can be made appendable, however, only along one dimension. This dimension must be specified as an ***appendable dimension*** when it is created.

In C, only the first element of the **SDcreate** parameter *dim_sizes* (i.e., the dimension of the lowest rank or the slowest-changing dimension) can be assigned the value `SD_UNLIMITED` (or `0`) to make the first dimension unlimited. In FORTRAN-77, only the *last* dimension (i.e., the dimension of the highest rank or the slowest-changing dimension) can be unlimited. In other words, in FORTRAN-77 *dim_sizes(rank)* must be set to the value `SD_UNLIMITED` to make the last dimension appendable.

To append data to a data set without overwriting previously-written data, the user must specify the appropriate coordinates in the *start* parameter of the **SDwritedata** routine. For example, if 15 data elements have been written to an unlimited dimension, appending data to the array requires a *start* coordinate of 15. Specifying a starting coordinate less than the current number of elements written to the unlimited dimension will result in data being overwritten. In either case, all of the coordinates in the array except the one corresponding to the unlimited dimension must be equal to or less than the lengths of their corresponding dimensions.

Any time an unlimited dimension is appended to, the HDF library will automatically adjust the dimension record to the new length. If the newly-appended data begins beyond the previous length of the dimension, the locations between the old data and the beginning of the newly-appended data are initialized to the assigned fill value if there is one defined by the user, or the default fill value if none is defined. Refer to Section "*Fill Values and Fill Mode"* for a discussion of fill value.

3.5.1.4. Determining whether an SDS Array is Appendable: SDisrecord

SDisrecord determines whether the data set identified by the parameter *sds_id* is appendable, which means that the slowest-changing dimension of the SDS array is declared unlimited when the data set is created. The syntax of **SDisrecord** is as follows:

```
C:          status = SDisrecord(sds_id);

FORTRAN:    status = sfisrcrd(sds_id)
```

SDisrecord returns `TRUE` (or `1`) when the data set specified by *sds_id* is appendable and `FALSE` (or `0`) otherwise. The parameter of this routine is defined in Table 3E.

TABLE 3E **SDisrecord Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDisrecord [int32] **(sfisrcrd)**	sds_id	int32	integer	Data set identifier

3.5.1.5. Setting the Block Size: SDsetblocksize

SDsetblocksize sets the size of the blocks used for storing the data for unlimited dimension data sets. This is used only when creating new data sets; it does not have any affect on existing data sets. The syntax of this routine is as follows:

```
C:          status = SDsetblocksize(sds_id, block_size);

FORTRAN:    status = sfsblsz(sds_id, block_size)
```

SDsetblocksize must be called after **SDcreate** or **SDselect** and before **SDwritedata**. The parameter *block_size* should be set to a multiple of the desired buffer size.

SDsetblocksize returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3F.

3.5.1.6. Setting the I/O Access Type of an SDS: SDsetaccesstype

SDsetaccesstype sets the type of I/O (serial, parallel,...) for accessing the data of the data set identified by *sds_id*. Valid values of *access_types* are `DFACC_SERIAL` (or `1`), `DFACC_PARALLEL` (or `11`), and `DFACC_DEFAULT` (or `0`.) The syntax of this routine is as follows:

```
C:          status = SDsetaccesstype(sds_id, accesstype);

FORTRAN:    status = sdfsacct(sds_id, accesstype)
```

SDsetaccesstype returns a value of `SUCCEED` (or `0`) if the SDS data can be accessed via accesstype or `FAIL` (or `-1`) otherwise. Its parameters are further described in Table 3F.

TABLE 3F **SDsetblocksize and SDsetaccesstype Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetblocksize [intn] **(sfsblsz)**	sds_id	int32	integer	Data set identifier
	block_size	int32	integer	Block size
SDsetaccesstype [intn] **(sdfsacct)**	sds_id	int32	integer	Data set identifier
	accesstype	int32	integer	I/O access type

EXAMPLE 5. **Appending Data to an SDS Array with an Unlimited Dimension.**

This example creates a 10x10 SDS array with one unlimited dimension and writes data to it. The file is reopened and the routine **SDisrecord/sfisrcrd** is used to determine whether the selected SDS array is appendable. Then new data is appended, starting at the 11th row.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDSUNLIMITED.hdf"
#define SDS_NAME      "AppendableData"
#define X_LENGTH      10
#define Y_LENGTH      10
#define RANK          2

main( )
{
   /************************* Variable declaration **************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 dim_sizes[2];
   int32 data[Y_LENGTH][X_LENGTH], append_data[X_LENGTH];
   int32 start[2], edges[2];
   int   i, j;

   /******************** End of variable declaration ***********************/

   /*
   * Data initialization.
   */
   for (j = 0; j < Y_LENGTH; j++)
   {
       for (i = 0; i < X_LENGTH; i++)
           data[j][i] = (i + 1) + (j + 1);
   }

   /*
   * Create the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_CREATE);

   /*
   * Define dimensions of the array. Make the first dimension
   * appendable by defining its length to be unlimited.
   */
   dim_sizes[0] = SD_UNLIMITED;
   dim_sizes[1] = X_LENGTH;

   /*
   * Create the array data set.
   */
   sds_id = SDcreate (sd_id, SDS_NAME, DFNT_INT32, RANK, dim_sizes);

   /*
   * Define the location and the size of the data to be written
   * to the data set.
   */
   start[0] = start[1] = 0;
   edges[0] = Y_LENGTH;
   edges[1] = X_LENGTH;

   /*
   * Write the data.
   */
   status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)data);

   /*
   * Terminate access to the array data set, terminate access
   * to the SD interface, and close the file.
   */
   status = SDendaccess (sds_id);
   status = SDend (sd_id);

   /*
   * Store the array values to be appended to the data set.
   */
   for (i = 0; i < X_LENGTH; i++)
       append_data[i] = 1000 + i;

   /*
   * Reopen the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_WRITE);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Check if selected SDS is unlimited. If it is not, then terminate access
   * to the SD interface and close the file.
   */
   if ( SDisrecord (sds_id) )
```

```
         {

         /*
         * Define the location of the append to start at the first column
         * of the 11th row of the data set and to stop at the end of the
         * eleventh row.
         */
         start[0] = Y_LENGTH;
         start[1] = 0;
         edges[0] = 1;
         edges[1] = X_LENGTH;

         /*
         * Append data to the data set.
         */
         status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)append_data);
         }

         /*
         * Terminate access to the data set.
         */
         status = SDendaccess (sds_id);

         /*
         * Terminate access to the SD interface and close the file.
         */
         status = SDend (sd_id);
      }
```

FORTRAN:

```
      program append_sds
      implicit none
C
C     Parameter declaration.
C
      character*16  FILE_NAME
      character*14  SDS_NAME
      integer       X_LENGTH, Y_LENGTH, RANK
      parameter    (FILE_NAME = 'SDSUNLIMITED.hdf',
     +              SDS_NAME = 'AppendableData',
     +              X_LENGTH = 10,
     +              Y_LENGTH = 10,
     +              RANK     = 2)
      integer       DFACC_CREATE, DFACC_WRITE, SD_UNLIMITED,
     +              DFNT_INT32
      parameter    (DFACC_CREATE = 4,
     +              DFACC_WRITE  = 2,
     +              SD_UNLIMITED = 0,
     +              DFNT_INT32 =   24)
C
C     Function declaration.
C
      integer sfstart, sfcreate, sfwdata, sfselect
      integer sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer dim_sizes(2)
      integer start(2), edges(2), stride(2)
      integer i, j
      integer data (X_LENGTH, Y_LENGTH), append_data(X_LENGTH)
```

```
C
C**** End of variable declaration ************************************
C
C
C     Data initialization.
C
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            data(i, j) = i + j
10      continue
20    continue
C
C     Create the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_CREATE)
C
C     Define dimensions of the array. Make the
C     last dimension appendable by defining its length as unlimited.
C
      dim_sizes(1) = X_LENGTH
      dim_sizes(2) = SD_UNLIMITED

C     Create the array data set.
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT32, RANK,
     .                   dim_sizes)
C
C     Define the location and the size of the data to be written
C     to the data set. Note that the elements of array stride are
C     set to 1 for contiguous writing.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Write the data.
C
      status = sfwdata(sds_id, start, stride, edges, data)
C
C     Terminate access to the data set, terminate access
C     to the SD interface, and close the file.
C
      status = sfendacc(sds_id)
      status = sfend(sd_id)
C
C     Store the array values to be appended to the data set.
C
      do 30 i = 1, X_LENGTH
         append_data(i) = 1000 + i - 1
30    continue
C
C     Reopen the file and initialize the SD.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Select the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)
C
C     Define the location of the append to start at the 11th
```

```
C     column of the 1st row and to stop at the end of the 10th row.
C
      start(1) = 0
      start(2) = Y_LENGTH
      edges(1) = X_LENGTH
      edges(2) = 1
C
C     Append the data to the data set.
C
      status = sfwdata(sds_id, start, stride, edges, append_data)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.5.2. Compressing SDS Data: SDsetcompress

The **SDsetcompress** routine compresses an existing data set or creates a new compressed data set. It is a simplified interface to the **HCcreate** routine, and should be used instead of **HCcreate** unless the user is familiar with the lower-level routines.

The compression algorithms currently supported by **SDsetcompress** are:

- Adaptive Huffman
- GZIP "deflation" (Lempel/Ziv-77 dictionary coder)
- Run-length encoding
- NBIT
- Szip

The syntax of the routine **SDsetcompress** is as follows:

```
C:          status = SDsetcompress(sds_id, comp_type, &c_info);

FORTRAN:    status = sfscompress(sds_id, comp_type, comp_prm)
```

The parameter *comp_type* specifies the compression type definition and is set to

`COMP_CODE_RLE` (or 1) for run-length encoding (RLE)
`COMP_CODE_SKPHUFF` (or 3) for Skipping Huffman
`COMP_CODE_DEFLATE` (or 4) for GZIP compression
`COMP_CODE_SZIP` (or 5) for Szip compression

Compression information is specified by the parameter *c_info* in C, and by the parameter *comp_prm* in FORTRAN-77. The parameter *c_info* is a pointer to a union structure of type *comp_info*. Refer to the **SDsetcompress** entry in the *HDF Reference Manual* for the description of the *comp_info* structure.

If *comp_type* is set to `COMP_CODE_RLE`, the parameters *c_info* and *comp_prm* are not used; *c_info* can be set to `NULL` and *comp_prm* can be undefined.

If *comp_type* is set to `COMP_CODE_SKPHUFF`, then the structure *skphuff* in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) must be provided with the size, in bytes, of the data elements.

If *comp_type* is set to `COMP_CODE_DEFLATE`, the deflate structure in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) must be provided with the information about the compression effort.

If *comp_type* is set to `COMP_CODE_SZIP`, the Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset must be specified in `c_info.szip.options_mask` and `c_info.szip.pixels_per_block` in C, and *comp_prm(1)* and *comp_prm(2)* in Fortran, respectively.

For example, to compress signed 16-bit integer data using the adaptive Huffman algorithm, the following definition and **SDsetcompress** call are used.

```
C:         comp_info c_info;
           c_info.skphuff.skp_size = sizeof(int16);
           status = SDsetcompress(sds_id, COMP_CODE_SKPHUFF, &c_info);

FORTRAN:   comp_prm(1) = 2
           COMP_CODE_SKPHUFF = 3
           status = sfscompress(sds_id, COMP_CODE_SKPHUFF, comp_prm)
```

To compress a data set using the gzip deflation algorithm with the maximum effort specified, the following definition and **SDsetcompress** call are used.

```
C:         comp_info c_info;
           c_info.deflate.level = 9;
           status = SDsetcompress(sds_id, COMP_CODE_DEFLATE, &c_info);

FORTRAN:   comp_prm(1) = 9
           COMP_CODE_DEFLATE = 4
           status = sfscompress(sds_id, COMP_CODE_DEFLATE, comp_prm)
```

SDsetcompress functionality is currently limited to the following:

- Write the compressed data, in its entirety, to the data set. The data set is built in-core then written in a single write operation.
- Compression is not supported on an SDS with unlimited dimension. **SDsetcompress** will return FAIL for such SDS and any subsequent writing to this SDS will write uncompressed data.

The existing compression algorithms supported by HDF do *not* allow partial modification to a compressed datastream. In addition, compressed data sets cannot be stored in external files (see Section 3.5.3.)

SDsetcompress returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The C version parameters are further described in Table 3G and the FORTRAN-77 version parameters are further described in Table 3H.

TABLE 3G

SDsetcompress Parameter List

Routine Name [Return Type]	Parameter	Parameter Type	Description
		C	
SDsetcompress [intn]	sds_id	int32	Data set identifier
	comp_type	int32	Compression method
	c_info	comp_info*	Pointer to compression information structure

TABLE 3H

sfscompress Parameter List

Routine Name	Parameter	Parameter Type	Description
		FORTRAN-77	
sfscompress	sds_id	integer	Data set identifier
	comp_type	integer	Compression method
	comp_prm	integer(*)	Compression parameters array

EXAMPLE 6.

Compressing SDS Data.

This example uses the routine **SDsetcompress/sfscompress** to compress SDS data with the GZIP compression method. See comments in the program regarding the use of the Skipping Huffman or RLE compression methods.

C:

```
#include "mfhdf.h"

#define FILE_NAME      "SDScompressed.hdf"
#define SDS_NAME       "SDSgzip"
#define X_LENGTH       5
#define Y_LENGTH       16
#define RANK           2

main( )
{
   /************************* Variable declaration **************************/

   int32     sd_id, sds_id, sds_index;
   intn      status;
   int32     comp_type;    /* Compression flag */
   comp_info c_info;   /* Compression structure */
   int32     start[2], edges[2], dim_sizes[2];
   int32     data[Y_LENGTH][X_LENGTH];
   int       i, j;

   /********************* End of variable declaration **********************/

   /*
   * Buffer array data and define array dimensions.
   */
   for (j = 0; j < Y_LENGTH; j++)
   {
    for (i = 0; i < X_LENGTH; i++)
            data[j][i] = (i + j) + 1;
   }
   dim_sizes[0] = Y_LENGTH;
   dim_sizes[1] = X_LENGTH;
```

```
       /*
       * Create the file and initialize the SD interface.
       */
       sd_id = SDstart (FILE_NAME, DFACC_CREATE);

       /*
       * Create the data set with the name defined in SDS_NAME.
       */
       sds_id = SDcreate (sd_id, SDS_NAME, DFNT_INT32, RANK, dim_sizes);

       /*
       * Ininitialize compression structure element and compression
       * flag for GZIP compression and call SDsetcompress.
       *
       *   To use the Skipping Huffman compression method, initialize
       *          comp_type = COMP_CODE_SKPHUFF
       *          c_info.skphuff.skp_size = value
       *
       *   To use the RLE compression method, initialize
       *          comp_type = COMP_CODE_RLE
       *   No structure element needs to be initialized.
       */
       comp_type = COMP_CODE_DEFLATE;
       c_info.deflate.level = 6;
       status = SDsetcompress (sds_id, comp_type, &c_info);

       /*
       * Define the location and size of the data set
       * to be written to the file.
       */
       start[0] = 0;
       start[1] = 0;
       edges[0] = Y_LENGTH;
       edges[1] = X_LENGTH;

       /*
       * Write the stored data to the data set. The last argument
       * must be explicitly cast to a generic pointer since SDwritedata
       * is designed to write generic data.
       */
       status = SDwritedata (sds_id, start, NULL, edges, (VOIDP)data);

       /*
       * Terminate access to the data set.
       */
       status = SDendaccess (sds_id);

       /*
       * Terminate access to the SD interface and close the file.
       */
       status = SDend (sd_id);

    }
```

FORTRAN:

```
      program  write_compressed_data
      implicit none
C
C     Parameter declaration.
C
      character*17  FILE_NAME
      character*7   SDS_NAME
      integer       X_LENGTH, Y_LENGTH, RANK
      parameter    (FILE_NAME = 'SDScompressed.hdf',
     +              SDS_NAME = 'SDSgzip',
     +              X_LENGTH = 5,
     +              Y_LENGTH = 16,
     +              RANK     = 2)
      integer       DFACC_CREATE, DFNT_INT32
      parameter    (DFACC_CREATE = 4,
     +              DFNT_INT32 = 24)
      integer       COMP_CODE_DEFLATE
      parameter    (COMP_CODE_DEFLATE = 4)
      integer       DEFLATE_LEVEL
      parameter    (DEFLATE_LEVEL = 6)
C     To use Skipping Huffman compression method, declare
C            integer   COMP_CODE_SKPHUFF
C            parameter(COMP_CODE_SKPHUFF = 3)
C     To use RLE compression method, declare
C            integer   COMP_CODE_RLE
C            parameter(COMP_CODE_RLE = 1)
C
C
C     Function declaration.
C
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend,
     +        sfscompress
C
C**** Variable declaration *******************************************
C
      integer  sd_id, sds_id, status
      integer  start(2), edges(2), stride(2), dim_sizes(2)
      integer  comp_type
      integer  comp_prm(1)
      integer  data(X_LENGTH, Y_LENGTH)
      integer  i, j
C
C**** End of variable declaration ************************************
C
C
C     Buffer array data and define array dimensions.
C
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            data(i, j) = i + j - 1
10       continue
20    continue
      dim_sizes(1) = X_LENGTH
      dim_sizes(2) = Y_LENGTH
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_CREATE)
C
C     Create the data set with the name SDS_NAME.
C
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT32, RANK, dim_sizes)
C
C     Initialize compression parameter (deflate level)
C     and call sfscompress function
C     For Skipping Huffman compression, comp_prm(1) should be set
C     to skipping sizes value (skp_size).
C
      comp_type   = COMP_CODE_DEFLATE
      comp_prm(1) = deflate_level
      status      = sfscompress(sds_id, comp_type, comp_prm(1))
C
C     Define the location and size of the data that will be written to
C     the data set.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Write the stored data to the data set.
C
      status = sfwdata(sds_id, start, stride, edges, data)
C
C     Terminate access to the  data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.5.3. External File Operations

The HDF library provides routines to store SDS arrays in an ***external file*** that is separate from the ***primary file*** containing the metadata for the array. Such an SDS array is called an ***external SDS array***. With external arrays, it is possible to link data sets in the same HDF file to multiple external files or data sets in different HDF files to the same external file.

External arrays are functionally identical to arrays in the primary data file. The HDF library keeps track of the beginning of the data set and adds data at the appropriate position in the external file. When data is written or appended along a specified dimension, the HDF library writes along that dimension in the external file and updates the appropriate dimension record in the primary file.

There are two methods for creating external SDS arrays. The user can create a new data set in an external file or move data from an existing internal data set to an external file. In either case, only the array values are stored externally, all metadata remains in the primary HDF file.

When an external array is created, a sufficient amount of space is reserved in the external file for the entire data set. The data set will begin at the specified byte offset and extend the length of the data set. The write operation will overwrite the target locations in the external file. The external file may be of any format, provided the number types, byte ordering, and dimension ordering are supported by HDF. However, the primary file must be an HDF file.

Routines for manipulating external SDS arrays can only be used with HDF files. Unidata-formatted netCDF files are not supported by these routines.

Note: Compressed data sets (see Section 3.5.2.) cannot be stored in external files.

3.5.3.1. Specifying the Directory Search Path of an External File: HXsetdir

There are three filesystem locations the HDF external file routines check when determining the location of an external file. They are, in order of search precedence:

1. The directory path specified by the last call to the **HXsetdir** routine.
2. The directory path specified by the $HDFEXTDIR shell environment variable.
3. The file system locations searched by the standard **open(3)** routine.

The syntax of **HXsetdir** is as follows:

```
C:          status = HXsetdir(dir_list);

FORTRAN:    status = hxisdir(dir_list, dir_length)
```

HXsetdir has one argument, a string specifying the directory list to be searched. This list can consist of one directory name or a set of directory names separated by colons. The FORTRAN-77 version of this routine takes an additional argument, *dir_length*, which specifies the length of the directory list string.

If an error condition is encountered, **HXsetdir** leaves the directory search path unchanged. The directory search path specified by **HXsetdir** remains in effect throughout the scope of the calling program.

HXsetdir returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **HXsetdir** are described in Table 3I.

3.5.3.2. Specifying the Location of the Next External File to be Created: HXsetcreatedir

HXsetcreatedir specifies the directory location of the next external file to be created. It overrides the directory location specified by $HDFEXTCREATEDIR and the locations searched by the **open(3)** call in the same manner as **HXsetdir**. Specifically, the search precedence is:

1. The directory specified by the last call to the **HXsetcreatedir** routine.
2. The directory specified by the $HDFEXTCREATEDIR shell environment variable.
3. The locations searched by the standard **open(3)** routine.

The syntax of **HXsetcreatedir** is as follows:

```
C:          status = HXsetcreatedir(dir);

FORTRAN:    status = hxiscdir(dir, dir_length)
```

HXsetcreatedir has one argument, the directory location of the next external file to be created. The FORTRAN-77 version of this routine takes an additional argument, *dir_length*, which specifies the length of the directory list string. If an error is encountered, the directory location is left unchanged.

HXsetcreatedir returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **HXsetcreatedir** are described in Table 3I.

TABLE 3I **HXsetdir and HXsetcreatedir Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
HXsetdir [intn] **(hxisdir)**	dir_list	char *	character*(*)	Directory list to be searched
	dir_length	Not applicable	integer	Length of the `dir_list` string
HXsetcreatedir [intn] **(hxiscdir)**	dir	char *	character*(*)	Directory location of the next external file to be created
	dir_length	Not applicable	integer	Length of the `dir` string

3.5.3.3. Creating a Data Set with Data Stored in an External File: SDsetexternalfile

Creating a data set in an external file involves the following steps:

1. Create the data set.
2. Specify that an external data file is to be used.
3. Write data to the data set.
4. Terminate access to the data set.

To create a data set with data stored in an external file, the calling program must make the following calls.

```
C:          sds_id = SDcreate(sd_id, name, ntype, rank, dim_sizes);
            status = SDsetexternalfile(sds_id, filename, offset);
            status = SDwritedata(sds_id, start, stride, edges, data);
            status = SDendaccess(sds_id);

FORTRAN:    sds_id = sfcreate(sd_id, name, ntype, rank, dim_sizes)
            status = sfsextf(sds_id, filename, offset)

            status = sfwdata(sds_id, start, stride, edges, data)
    OR      status = sfwcdata(sds_id, start, stride, edges, data)

            status = sfendacc(sds_id)
```

For a newly-created data set, **SDsetexternalfile** marks the SDS identified by *sds_id* as one whose data is to be written to an external file. It does not actually write data to an external file; it marks the data set as an external data set for all subsequent **SDwritedata** operations.

Note that data can only be moved once for any given data set, i.e., **SDsetexternalfile** can only be called once after a data set has been created. It is the user's responsibility to make sure that the external data file is kept with the primary HDF file.

The parameter *filename* is the name of the external data file and *offset* is the number of bytes from the beginning of the external file to the location where the first byte of data should be written. If a file with the name specified by *filename* exists in the current directory search path, HDF will access it as the external file. If the file does not exist, HDF will create one in the directory named in the last call to **HXsetcreatefile**. If an absolute pathname is specified, the external file will be created at the location specified by the pathname, overriding the location specified by the last call to **HXsetcreatefile**. Use caution when writing to existing external or primary files since the HDF library starts the write operation at the specified offset without determining whether data is being overwritten.

Once the name of an external file is established, it cannot be changed without breaking the association between the data set's metadata and the data it describes.

SDsetexternalfile returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDsetexternalfile** are described in Table 3J.

3.5.3.4. Getting External File Information of a Data Set: SDgetexternalinfo

SDgetexternalinfo retrieves external file information of a data set, when the data set has external element. The information includes the external file's name, the position, where the data set's data had been written in the external file, and the length of the external data. **SDgetexternalinfo** will return `0` if the data set does not have external element.

The syntax of **SDgetexternalinfo** is as follows:

```
C:          status = SDgetexternalinfo(sds_id, buf_size, filename, &offset,
                              &length);

FORTRAN:    Currently unavailable
```

The application must provide sufficient buffer for the external file name. When the external file name is available and *buf_size* is 0, **SDgetexternalinfo** simply returns the length of the external file name. Thus, application can call **SDgetexternalinfo** passing in 0 for *buf_size* first, then allocate the buffer sufficiently before calling **SDgetexternalinfo** again passing in the proper length for *buf_size* and appropriately allocated buffer *filename*. **SDgetexternalinfo** stores the external file name in the buffer *filename* up to the name's length or the value in *buf_size*, whichever smaller.

SDgetexternalinfo stores in the parameter *offset* the number of bytes from the beginning of the external file to the location where the first byte of data had been written and in the parameter *length* the length of the data.

SDgetexternalinfo returns one of the following values:

- the actual length of the external file name or the length of the retrieved file name, if there is external element
- `0`, if there is no external element
- `FAIL` (or `-1`), if failure occurs

The parameters of **SDgetexternalinfo** are described in Table 3J.

TABLE 3J **SDsetexternalfile Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDsetexternalfile [intn] **(sfsextf)**	sds_id	int32	integer	Data set identifier
	filename	char *	character*(*)	Name of the file to contain the external data set
	offset	int32	integer	Offset in bytes from the beginning of the external file to where the SDS data will be written
SDgetexternalinfo [intn] **(unvailable)**	sds_id	int32	N/A	Data set identifier
	buf_size	uintn	N/A	Size of buffer for external file name
	filename	char *	N/A	Buffer for external file name
	offset	*int32	N/A	Offset in bytes from the beginning of the external file to where the SDS data had been written
	length	*int32	N/A	Length of the data written in the external file

3.5.3.5. Moving Existing Data to an External File

Data can be moved from a primary file to an external file. The following steps perform this task:

1. Select the data set.
2. Specify the external data file.
3. Terminate access to the data set.

To move data set data to an external file, the calling program must make the following calls:

```
C:          sds_id = SDselect(sd_id, sds_index);
            status = SDsetexternalfile(sds_id, filename, offset);
            status = SDendaccess(sds_id);

FORTRAN:    sds_id = sfselect(sd_id, sds_index)
            status = sfsextf(sds_id, filename, offset)
            status = sfendacc(sds_id)
```

For an existing data set, **SDsetexternalfile** moves the data to the external file. Any data in the external file that occupies the space reserved for the external array will be overwritten as a result of this operation. Data of an existing data set in the primary file can only be moved to the external file once. During the operation, the data is written to the external file as a contiguous stream regardless of how it is stored in the primary file. Because data is moved as is, any unwritten locations in the data set are preserved in the external file. Subsequent read and write operations performed on the data set will access the external file.

EXAMPLE 7. **Moving Data to the External File.**

This example illustrates the use of the routine **SDsetexternalfile/sfsextf** to move the SDS data written in Example 2 to the external file.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"
#define EXT_FILE_NAME "ExternalSDS"
#define OFFSET        24
```

```
main( )
{

   /************************* Variable declaration *************************/

   int32 sd_id, sds_id, sds_index, offset;
   intn  status;

   /******************** End of variable declaration **********************/

   /*
   * Open the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_WRITE);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Create a file with the name EXT_FILE_NAME and move the data set
   * values into it, starting at byte location OFFSET.
   */
   status = SDsetexternalfile (sds_id, EXT_FILE_NAME, OFFSET);

   /*
   * Terminate access to the data set, SD interface, and file.
   */
   status = SDendaccess (sds_id);
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  write_extfile
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*11 EXT_FILE_NAME
      integer      OFFSET
      integer      DFACC_WRITE
      parameter   (FILE_NAME      = 'SDS.hdf',
     +             EXT_FILE_NAME  = 'ExternalSDS',
     +             OFFSET         = 24,
     +             DFACC_WRITE    = 2)

C
C     Function declaration.
C
      integer sfstart, sfselect, sfsextf, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, offset
      integer status
C
C**** End of variable declaration ************************************
C
C
```

```
C     Open the HDF file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Select the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)
C
C     Create a file with the name EXT_FILE_NAME and move the data set
C     into it, starting at byte location OFFSET.
C
      status = sfsextf(sds_id, EXT_FILE_NAME, OFFSET)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.6. Reading Data from an SDS Array: SDreaddata

Data of an SDS array can be read as an entire array, a subset of the array, or a set of samples of the array. SDS data is read from an external file in the same way that it is read from a primary file; whether the SDS array is stored in an external file is transparent to the user. Reading data from an SDS array involves the following steps:

1. Select the data set.
2. Define the portion of the data to be read.
3. Read data portion as defined.

To read data from an SDS array, the calling program must contain the following function calls:

```
C:         sds_id = SDselect(sd_id, sds_index);
           status = SDreaddata(sds_id, start, stride, edges, data);

FORTRAN:   sds_id = sfselect(sd_id, sds_index)

           status = sfrdata(sds_id, start, stride, edges, data)

    OR     status = sfrcdata(sds_id, start, stride, edges, data)
```

Note that step 2 is not illustrated in the function call syntax; it is carried out by assigning values to the parameters *start*, *stride*, and *edges* before the routine **SDreaddata** is called in step 3.

SDreaddata reads the data according to the definition specified by the parameters *start*, *stride*, and *edges* and stores the data into the buffer provided, *data*. The argument *sds_id* is the SDS identifier returned by **SDcreate** or **SDselect**. As with **SDwritedata**, the arguments *start*, *stride*, and *edges* describe the starting location, the number of elements to skip after each read, and the number of elements to be read, respectively, for each dimension. For additional information on the parameters *start*, *stride*, and *edges*, refer to Section "*Writing Data to an SDS Array: SDwritedata*".

There are two FORTRAN-77 versions of this routine: **sfrdata** reads numeric data and **sfrcdata** reads character data.

SDreaddata returns a value of `SUCCEED` (or `0`), including the situation when the data set does not contain data, or `FAIL` (or `-1`). The parameters of **SDreaddata** are further described in Table 3K.

TABLE 3K **SDreaddata Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDreaddata [intn] **(sfrdata/ sfrcdata)**	sds_id	int32	integer	Data set identifier
	start	int32[]	integer(*)	Array containing the position at which the read will start for each dimension
	stride	int32[]	integer(*)	Array containing the number of data locations the current location is to be moved forward before the next read
	edges	int32[]	integer(*)	Array containing the number of data elements to be read along each dimension
	data	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer the data will be read into

EXAMPLE 8. **Reading from an SDS.**

This example uses the routine **SDreaddata/sfrdata** to read the data that has been written in Example 2, modified in Example 4, and moved to the external file in the Example 7. Note that the original file SDS.hdf that contains the SDS metadata and the external file ExternalSDS that contains the SDS raw data should reside in the same directory. The fact that raw data is in the external file is transparent to the user's program.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"
#define X_LENGTH      5
#define Y_LENGTH      16

main( )
{
   /************************* Variable declaration *************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 start[2], edges[2];
   int32 data[Y_LENGTH][X_LENGTH];
   int   i, j;

   /******************** End of variable declaration **********************/

   /*
   * Open the file for reading and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_READ);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Set elements of array start to 0, elements of array edges
   * to SDS dimensions,and use NULL for the argument stride in SDreaddata
   * to read the entire data.
   */
   start[0] = 0;
   start[1] = 0;
   edges[0] = Y_LENGTH;
   edges[1] = X_LENGTH;

   /*
   * Read entire data into data array.
   */
   status = SDreaddata (sds_id, start, NULL, edges, (VOIDP)data);

   /*
   * Print 10th row; the following numbers should be displayed.
   *
   *         10 1000 12 13 14
   */
   for (j = 0; j < X_LENGTH; j++) printf ("%d ", data[9][j]);
   printf ("\n");

   /*
   * Terminate access to the data set.
   */
   status = SDendaccess (sds_id);

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  read_data
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      integer      X_LENGTH, Y_LENGTH
      parameter   (FILE_NAME  = 'SDS.hdf',
     +             X_LENGTH = 5,
     +             Y_LENGTH = 16)
      integer      DFACC_READ, DFNT_INT32
      parameter   (DFACC_READ = 1,
     +             DFNT_INT32 = 24)

C
C     Function declaration.
C
      integer sfstart, sfselect, sfrdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer start(2), edges(2), stride(2)
      integer data(X_LENGTH, Y_LENGTH)
      integer j
```

```
C
C**** End of variable declaration ***********************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)

C
C     Select the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)

C
C     Set elements of the array start to 0, elements of the array edges to
C     SDS dimensions, and elements of the array stride to 1 to read the
C     entire data.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Read entire data into data array. Note that sfrdata is used
C     to read the numeric data.
C
      status = sfrdata(sds_id, start, stride, edges, data)

C
C     Print 10th column; the following numbers are displayed:
C
C           10 1000 12 13 14
C
      write(*,*) (data(j,10), j = 1, X_LENGTH)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

EXAMPLE 9.

Reading Subsets of an SDS.

This example shows how parameters *start*, *stride*, and *edges* of the routine **SDreadata/sfrdata** can be used to read three subsets of an SDS array.

C:

For the first subset, the program reads every 3rd element of the 2nd column starting at the 4th row of the data set created in Example 2 and modified in Examples 4 and 7.
For the second subset the program reads the first 4 elements of the 10th row.
For the third subset, the program reads from the same data set every 6th element of each column and 4th element of each row starting at 1st column, 3d row.

FORTRAN-77:

Fortran program reads transposed data to reflect the difference in C and Fortran internal storage.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"
#define SUB1_LENGTH   5
#define SUB2_LENGTH   4
#define SUB3_LENGTH1  2
#define SUB3_LENGTH2  3

main( )
{
   /************************* Variable declaration *************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 start[2], edges[2], stride[2];
   int32 sub1_data[SUB1_LENGTH];
   int32 sub2_data[SUB2_LENGTH];
   int32 sub3_data[SUB3_LENGTH2][SUB3_LENGTH1];
   int   i, j;

   /********************* End of variable declaration **********************/

   /*
   * Open the file for reading and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_READ);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);
   /*
   *         Reading the first subset.
   *
   * Set elements of start, edges, and stride arrays to read
   * every 3rd element in the 2nd column starting at 4th row.
   */
   start[0] = 3;   /* 4th row */
   start[1] = 1;   /* 2nd column */
   edges[0] = SUB1_LENGTH; /* SUB1_LENGTH elements are read along 2nd column*/
   edges[1] = 1;
   stride[0] = 3;  /* every 3rd element is read along 2nd column */
   stride[1] = 1;

   /*
   * Read the data from the file into sub1_data array.
   */
   status = SDreaddata (sds_id, start, stride, edges, (VOIDP)sub1_data);

   /*
   * Print what we have just read; the following numbers should be displayed:
   *
   *             5 8 1000 14 17
   */
   for (j = 0; j < SUB1_LENGTH; j++) printf ("%d ", sub1_data[j]);
   printf ("\n");
```

```
/*
*         Reading the second subset.
*
* Set elements of start and edges arrays to read
* first 4 elements of the 10th row.
*/
start[0] = 9;  /* 10th row  */
start[1] = 0;  /* 1st column */
edges[0] = 1;
edges[1] = SUB2_LENGTH; /* SUB2_LENGTH elements are read along 10th row */

/*
* Read data from the file into sub2_data array. Note that the third
* parameter is set to NULL for contiguous reading.
*/
status = SDreaddata (sds_id, start, NULL, edges, (VOIDP)sub2_data);

/*
* Print what we have just read; the following numbers should be displayed:
*
*            10 1000 12 13
*/
for (j = 0; j < SUB2_LENGTH; j++) printf ("%d ", sub2_data[j]);
printf ("\n");

/*
*         Reading the third subset.
*
* Set elements of the arrays start, edges, and stride to read
* every 6th element in the column and 4th element in the row
* starting at 1st column, 3d row.
*/
start[0] = 2;  /* 3d row */
start[1] = 0;  /* 1st column */
edges[0] = SUB3_LENGTH2; /* SUB3_LENGTH2 elements are read along
                            each column */
edges[1] = SUB3_LENGTH1; /* SUB3_LENGTH1 elements are read along
                            each row */
stride[0] = 6; /* read every 6th element along each column */
stride[1] = 4; /* read every 4th element along each row */

/*
* Read the data from the file into sub3_data array.
*/
status = SDreaddata (sds_id, start, stride, edges, (VOIDP)sub3_data);

/*
* Print what we have just read; the following numbers should be displayed:
*
*            3 7
*            9 13
*            15 19
*/
for ( j = 0; j < SUB3_LENGTH2; j++ ) {
    for (i = 0; i < SUB3_LENGTH1; i++) printf ("%d ", sub3_data[j][i]);
    printf ("\n");
}
/*
* Terminate access to the data set.
*/
status = SDendaccess (sds_id);

/*
```

```
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  read_subsets
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      parameter   (FILE_NAME  = 'SDS.hdf')
      integer      DFACC_READ, DFNT_INT32
      parameter   (DFACC_READ = 1,
     +             DFNT_INT32 = 24)
      integer      SUB1_LENGTH, SUB2_LENGTH, SUB3_LENGTH1,
     +             SUB3_LENGTH2
      parameter   (SUB1_LENGTH  = 5,
     +             SUB2_LENGTH  = 4,
     +             SUB3_LENGTH1 = 2,
     +             SUB3_LENGTH2 = 3)

C
C     Function declaration.
C
      integer sfstart, sfselect, sfrdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer start(2), edges(2), stride(2)
      integer sub1_data(SUB1_LENGTH)
      integer sub2_data(SUB2_LENGTH)
      integer sub3_data(SUB3_LENGTH1,SUB3_LENGTH2)
      integer i, j
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
C
C     Select the first data set.
C
      sds_index = 0
      sds_id =sfselect(sd_id, sds_index)
C
C            Reading the first subset.
C
C     Set elements of start, stride, and edges arrays to read
C     every 3d element in in the 2nd row starting in the 4th column.
C
      start(1) = 1
      start(2) = 3
      edges(1) = 1
      edges(2) = SUB1_LENGTH
      stride(1) = 1
      stride(2) = 3
C
C     Read the data from sub1_data array.
```

```
C
      status = sfrdata(sds_id, start, stride, edges, sub1_data)

C
C     Print what we have just read, the following numbers should be displayed:
C
C           5 8 1000 14 17
C
      write(*,*) (sub1_data(j), j = 1, SUB1_LENGTH)
C
C            Reading the second subset.
C
C     Set elements of start, stride, and edges arrays to read
C     first 4 elements of 10th column.
C
      start(1) = 0
      start(2) = 9
      edges(1) = SUB2_LENGTH
      edges(2) = 1
      stride(1) = 1
      stride(2) = 1
C
C     Read the data into sub2_data array.
C
      status = sfrdata(sds_id, start, stride, edges, sub2_data)

C
C     Print what we have just read; the following numbers should be displayed:
C
C          10 1000 12 13
C
      write(*,*) (sub2_data(j), j = 1, SUB2_LENGTH)
C
C            Reading the third subset.
C
C     Set elements of start, stride and edges arrays to read
C     every 6th element in the row and every 4th element in the column
C     starting at 1st row, 3rd column.
C
      start(1) = 0
      start(2) = 2
      edges(1) = SUB3_LENGTH1
      edges(2) = SUB3_LENGTH2
      stride(1) = 4
      stride(2) = 6
C
C     Read the data from the file into sub3_data array.
C
      status = sfrdata(sds_id, start, stride, edges, sub3_data)

C
C     Print what we have just read; the following numbers should be displayed:
C
C         3 9 15
C         7 13 19
C
      do 50 i = 1, SUB3_LENGTH1
         write(*,*) (sub3_data(i,j), j = 1, SUB3_LENGTH2)
50    continue
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
```

```
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.7. Obtaining Information about SD Data Sets

The routines covered in this section provide methods for obtaining information about all scientific data sets in a file, for identifying the data sets that meet certain criteria, and for obtaining information about specific data sets.

SDfileinfo obtains the numbers of data sets and file attributes, set by SD interface routines, in a file. **SDgetinfo** provides information about an individual SDS. To retrieve information about all data sets in a file, a calling program can use **SDfileinfo** to determine the number of data sets, followed by repeated calls to **SDgetinfo** to obtain the information about a particular data set.

SDnametoindex, SDnametoindices, or **SDreftoindex** can be used to obtain the index of an SDS in a file knowing its name or reference number. Refer to Section "*Required SDS Components"* for a description of the data set index and reference number. **SDidtoref** is used when the reference number of an SDS is required by another routine and the SDS identifier is available.

These routines are described individually in the following subsections.

3.7.1. Obtaining Information about the Contents of a File: SDfileinfo

SDfileinfo determines the number of scientific data sets and the number of file attributes contained in a file. This information is often useful in index validation or sequential searches. The syntax of **SDfileinfo** is as follows:

```
C:          status = SDfileinfo(sd_id, &n_datasets, &n_file_attrs);

FORTRAN:    status = sffinfo(sd_id, n_datasets, n_file_attrs)
```

SDfileinfo stores the numbers of scientific data sets and file attributes in the parameters *n_datasets* and *n_file_attrs*, respectively. Note that the value returned by *n_datasets* will include the number of SDS arrays *and* the number of dimension scales. Refer to Section "*Dimension Scales"* and Section "*Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar"* for the description of dimension scales and its association with SDS arrays as well as how to distinguish between SDS arrays and dimension scales. The file attributes are those that are created by **SDsetattr** for an SD interface identifier instead of an SDS identifier. Refer to Section "*Creating or Writing User-defined Attributes: SDsetattr"* for the discussion of **SDsetattr**.

SDfileinfo returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDfileinfo** are specified in Table 3L.

3.7.2. Obtaining Information about a Specific SDS: SDgetinfo

SDgetinfo provides basic information about an SDS array. Often information about an SDS array is needed before reading and working with the array. For instance, the rank, dimension sizes, and/or number type of an array are needed to allocate the proper amount of memory to work with the array. **SDgetinfo** takes an SDS identifier as input, and retrieves the name, rank, dimension sizes, number type, and number of attributes for the corresponding SDS. The syntax of this routine is as follows:

```
C:          status = SDgetinfo(sds_id, sds_name, &rank, dim_sizes, &ntype,
                                    &n_attrs);

FORTRAN:    status = sfginfo(sds_id, sds_name, rank, dim_sizes, ntype, n_attrs)
```

SDgetinfo stores the name, rank, dimension sizes, number type, and number of attributes of the specified data set into the parameters *sds_name*, *rank*, *dim_sizes*, *ntype*, and *n_attrs*, respectively. The parameter *sds_name* is a character string. Note that, starting in HDF 4.2.2, the name of the SDS is no longer limited to `64` characters. Thus, it is recommended that the application use **SDgetnamlen** to obtain the length of the data set's name so that it can sufficiently allocate space for the name prior to calling **SDgetinfo**.

If the data set is created with an unlimited dimension, then in the C interface, the first element of the *dim_sizes* array (corresponding to the slowest-changing dimension) contains the number of records in the unlimited dimension; in the FORTRAN-77 interface, the last element of the array *dim_sizes* (corresponding to the slowest-changing dimension) contains this information.

The parameter *ntype* contains any type that HDF supports for the scientific data. Refer to (See Table 2F, for the list of supported number types and their corresponding defined values. The parameter *n_attrs* only reflects the number of attributes assigned to the data set specified by *sds_id*; file attributes are not included. Use **SDfileinfo** to get the number of file attributes.

SDgetinfo returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDgetinfo** are specified in Table 3L.

3.7.3. Obtaining Data Set Compression Information: SDgetcompinfo

SDgetcompinfo retrieves the compression information used to create or write an SDS data set. **SDgetcompinfo** replaces **SDgetcompress** because this function has flaws, causing failure for some chunked and chunked/compressed data.

The possible compression algorithms used in SDS include:

- Adaptive Huffman
- GZIP "deflation" (Lempel/Ziv-77 dictionary coder)
- Run-length encoding
- NBIT
- Szip

SDgetcompinfo takes one input parameter, *sds_id*, a data set identifier, and two return parameters, *comp_type*, identifying the type of compression used, and either *c_info* (in C) or *comp_prm* (in FORTRAN-77), containing further compression information.

The syntax of **SDgetcompinfo** is as follows:

```
C:          status = SDgetcompinfo(sds_id, comp_type, c_info);

FORTRAN:    status = sfgcompress(sds_id, comp_type, comp_prm)
```

See Section "*Compressing SDS Data: SDsetcompress"* for a discussion on *comp_type*, *c_info*, and *comp_prm*, and a list of supported compression modes.

The parameter *comp_type* specifies the compression type definition and is set to

`COMP_CODE_NONE` (or 0) for no compression
`COMP_CODE_RLE` (or 1) for run-length encoding (RLE)
`COMP_CODE_NBIT` (or 2) for NBIT compression
`COMP_CODE_SKPHUFF` (or 3) for Skipping Huffman
`COMP_CODE_DEFLATE` (or 4) for GZIP compression
`COMP_CODE_SZIP` (or 5) for Szip compression

Compression information is returned by the parameter *c_info* in C, and by the parameter *comp_prm* in FORTRAN-77. The parameter *c_info* is a pointer to a union structure of type *comp_info.* Refer to the **SDsetcompress** entry in the *HDF Reference Manual* for the description of the *comp_info* structure.

When *comp_type* is `COMP_CODE_NONE` or `COMP_CODE_RLE`, the parameters *c_info* and *comp_prm* are unchanged.

When *comp_type* is `COMP_CODE_SKPHUFF`, then the structure *skphuff* in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) will store the size, in bytes, of the data elements.

When *comp_type* is `COMP_CODE_DEFLATE`, then the deflate structure in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) will store the information about the compression effort.

When *comp_type* is `COMP_CODE_SZIP`, then the Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset will be specified in `c_info.szip.options_mask` and `c_info.szip.pixels_per_block` in C, and *comp_prm(1)* and *comp_prm(2)* in Fortran, respectively.

SDgetcompinfo returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDgetcompinfo** are specified in Table 3L.

TABLE 3L **SDfileinfo, SDgetinfo, and SDgetcompinfo Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDfileinfo [intn] **(sffinfo)**	sd_id	int32	integer	SD interface identifier
	n_datasets	int32 *	integer	Number of data sets in the file
	n_file_attrs	int32 *	integer	Number of global attributes in the file
SDgetinfo [intn] **(sfginfo)**	sds_id	int32	integer	Data set identifier
	sds_name	char*	character*(*)	Name of the data set
	rank	int32 *	integer	Number of dimensions in the data set
	dim_sizes	int32 []	integer (*)	Size of each dimension in the data set
	ntype	int32 *	integer	Number type of the data in the data set
	n_attrs	int32 *	integer	Number of attributes in the data set
SDgetcompinfo [intn] **(sfgcompress)**	sds_id	int32	integer	Data set identifier
	comp_type	comp_coder_t	integer	Type of compression
	c_info	comp_info	N/A	Pointer to compression information structure
	comp_prm(1)	N/A	integer	Compression parameter in array format

EXAMPLE 10. **Getting Information about a File and an SDSs.**

This example illustrates the use of the routine **SDfileinfo/sffinfo** to obtain the number of data sets in the file SDS.hdf and the routine **SDgetinfo/sfginfo** to retrieve the name, rank, dimension sizes, data type and number of attributes of the selected data set.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"

main( )
{
   /************************ Variable declaration *************************/

   int32 sd_id, sds_id;
   intn  status;
   int32 n_datasets, n_file_attrs, index;
   int32 dim_sizes[MAX_VAR_DIMS];
   int32 rank, data_type, n_attrs;
   char  name[MAX_NC_NAME];
   int   i;

   /******************** End of variable declaration **********************/

   /*
   * Open the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_READ);

   /*
   * Determine the number of data sets in the file and the number
   * of file attributes.
   */
   status = SDfileinfo (sd_id, &n_datasets, &n_file_attrs);

   /*
   * Access every data set and print its name, rank, dimension sizes,
   * data type, and number of attributes.
   * The following information should be displayed:
   *
   *               name = SDStemplate
   *               rank = 2
   *               dimension sizes are : 16  5
   *               data type is  24
   *               number of attributes is  0
   */
   for (index = 0; index < n_datasets; index++)
   {
       sds_id = SDselect (sd_id, index);
       status = SDgetinfo (sds_id, name, &rank, dim_sizes,
                           &data_type, &n_attrs);

       printf ("name = %s\n", name);
       printf ("rank = %d\n", rank);
       printf ("dimension sizes are : ");
       for (i=0; i< rank; i++) printf ("%d  ", dim_sizes[i]);
       printf ("\n");
       printf ("data type is  %d\n", data_type);
       printf ("number of attributes is  %d\n", n_attrs);

       /*
       * Terminate access to the data set.
       */
       status = SDendaccess (sds_id);
   }

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program get_data_set_info
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      parameter   (FILE_NAME = ’SDS.hdf’)
      integer      DFACC_READ, DFNT_INT32
      parameter   (DFACC_READ = 1,
     +             DFNT_INT32 = 24)
      integer      MAX_NC_NAME, MAX_VAR_DIMS
      parameter   (MAX_NC_NAME  = 256,
     +             MAX_VAR_DIMS = 32)
C
C     Function declaration.
C
      integer sfstart, sffinfo, sfselect, sfginfo
      integer sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id
      integer n_datasets, n_file_attrs, index
      integer status, n_attrs
      integer rank, data_type
      integer dim_sizes(MAX_VAR_DIMS)
      character name *(MAX_NC_NAME)
      integer i
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
C
C     Determine the number of data sets in the file and the number of
C     file attributes.
C
      status = sffinfo(sd_id, n_datasets, n_file_attrs)
C
C     Access every data set in the file and print its name, rank,
C     dimension sizes, data type, and number of attributes.
C     The following information should be displayed:
C
C                name = SDStemplate
C                rank =   2
C                dimension sizes are :   5  16
C                data type is   24
C                number of attributes is   0
C
      do 10 index = 0, n_datasets - 1
         sds_id = sfselect(sd_id, index)
         status = sfginfo(sds_id, name, rank, dim_sizes, data_type,
     .                    n_attrs)
         write(*,*)  "name = ", name(1:15)
         write(*,*)  "rank = ", rank
         write(*,*)  "dimension sizes are : ", (dim_sizes(i), i=1, rank)
         write(*,*)  "data type is ", data_type
         write(*,*)  "number of attributes is ", n_attrs
C
```

```
C     Terminate access to the current data set.
C
         status = sfendacc(sds_id)
10    continue
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.7.4. Locating an SDS by Name: SDnametoindex

SDnametoindex determines and returns the index of a data set in a file given the data set's name. The syntax of this routine is as follows:

```
C:          sds_index = SDnametoindex(sd_id, sds_name);

FORTRAN:    sds_index = sfn2index(sd_id, sds_name)
```

The parameter *sds_name* is a character string. Note that, starting in HDF 4.2.2, the name of the SDS is no longer limited to `64` characters, which was the limit prior to 4.2.2.

If more than one data set has the name specified by *sds_name*, **SDnametoindex** will return the index of the first data set, which could be an SDS or a coordinate variable (also called dimension scale.) Note that if there are more than one data set with the same name in the file, writing to a data set returned by this function without verifying that it is the desired data set could cause data corruption. Refer to the *Important Note in Chapter 3* for more details regarding the problem and how to handle it.

SDgetnumvars_byname can be used to get the number of data sets (or variables, which includes both data sets and coordinate variables) with the same name. **SDnametoindices** can be used to get a list of structures containing the indices and the types of all the variables of that same name.

An index obtained by **SDnametoindex** or **SDnametoindices** can then be used by **SDselect** to obtain an SDS identifier for the specified data set. The **SDnametoindex** routine is case-sensitive to the name specified by *sds_name* and does not accept wildcards as part of that name. The name must exactly match the name of the SDS being searched for.

SDnametoindex returns the index of a data set or `FAIL` (or `-1`). The parameters of **SDnametoindex** are specified in Table 3M.

3.7.5. Locating More Than One SDS by the Same Name: SDnametoindices

SDnametoindices returns indices of all data sets having the same name. The data sets can be either SDSs or coordinate variables. The syntax of this routine is as follows:

```
C:          status = SDnametoindices(sd_id, sds_name, var_list);

FORTRAN:    status = sfn2indices(sd_id, sds_name, var_list, type_list,
                               n_vars)
```

The parameter *sds_name* is a character string. Note that, starting in HDF 4.2.2, the name of the SDS is no longer limited to `64` characters, which was the limit prior to 4.2.2.

SDnametoindices retrieves a list of structures `varlist_t`, containing the indices and the types of all variables of the same name *sds_name*. The structure `varlist_t` is defined as:

```
typedef struct varlist
```

```
{
    int32 var_index;    /* index of a variable */
    vartype_t var_type; /* type of a variable */
} varlist_t;
```

The type of a variable `vartype_t` is defined as:

```
IS_SDSVAR=0 : variable is an actual SDS
IS_CRDVAR=1 : variable is a coordinate variable
UNKNOWN=2 : variable is created before HDF 4.2.2, unknown type
```

Prior to calling **SDnametoindices**, **SDgetnumvars_byname** can be used to get the number of data sets, with which the application can allocate *var_list* appropriately. Also, when the number of data sets returned is 1, the application can call **SDnametoindex** instead of **SDnametoindices** for simplicity.

An index obtained by **SDnametoindex** or **SDnametoindices** can then be used by **SDselect** to obtain an SDS identifier for the specified data set.

The **SDnametoindices** routine is case-sensitive to the name specified by *sds_name* and does not accept wildcards as part of that name. The name must match exactly the name of the SDS being searched for.

SDnametoindices returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDname-toindices** are specified in Table 3M.

3.7.6. Getting Number of Data Sets Given a Name: SDgetnumvars_byname

SDgetnumvars_byname determines and returns the number of variables in a file having the same name. The variables may include both data sets and coordinate variables. The syntax of this routine is as follows:

```
C:          status = SDgetnumvars_byname(sd_id, sds_name, n_vars);

FORTRAN:    status = sfgnvars_byname(sd_id, sds_name, n_vars);
```

The parameter *sds_name* is a character string. Note that, starting in HDF 4.2.2, the name of the SDS is no longer limited to `64` characters, which was the limit prior to 4.2.2.

SDgetnumvars_byname returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDgetnumvars_byname** are specified in Table 3M.

TABLE 3M **SDnametoindex, SDnametoindices, and SDgetnumvars_byname Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDnametoindex [int32] **(sfn2index)**	sd_id	int32	integer	SD interface identifier
	sds_name	char *	character*(*)	Name of the data set
SDnametoindices **[intn]** **(sfn2indices**	sd_id	int32	integer	SD interface identifier
	sds_name	char *	character*(*)	Name of the data set
	var_list	varlist_t *	integer*	List of variables having name sds_name; Fortran: list of ?
	type_list (only Fortran)	N/A	integer*	Fortran: list of types of variables
	n_vars (only Fortran)	N/A	integer	Fortran: number of variables found
SDgetnumvars_byname [intn] **(sfgnvars_byname)**	sds_id	int32	integer	SDS identifier
	sds_name	char *	character*(*)	Name of the data set
	n_vars	unsigned*	integer	Number of variables having name sds_name

3.7.7. Locating an SDS by Reference Number: SDreftoindex

SDreftoindex determines and returns the index of a data set in a file given the data set's reference number. The syntax of this routine is as follows:

```
C:          sds_index = SDreftoindex(sd_id, ref);

FORTRAN:    sds_index = sfref2index(sd_id, ref)
```

The reference number can be obtained using **SDidtoref** if the SDS identifier is available. Remember that reference numbers do not necessarily adhere to any ordering scheme.

SDreftoindex returns either the index of an SDS or `FAIL` (or `-1`). The parameters of this routine are specified in Table 3N.

3.7.8. Obtaining the Reference Number Assigned to the Specified SDS: SDidtoref

SDidtoref returns the reference number of the data set identified by the parameter *sds_id* if the data set is found, or `FAIL` (or `-1`) otherwise. The syntax of this routine is as follows:

```
C:          sds_ref = SDidtoref(sds_id);

FORTRAN:    sds_ref = sfid2ref(sds_id)
```

This reference number is often used by **Vaddtagref** to add the data set to a vgroup. Refer to Chapter 5, *Vgroups (V API)*, for more information.

The parameter of **SDidtoref** is specified in Table 3N.

3.7.9. Obtaining the Type of an HDF4 Object: SDidtype

SDidtype returns the type of an object, given the object's identifier, *obj_id*. The syntax of this routine is as follows:

```
C:          obj_type = SDidtype(obj_id);
```

```
FORTRAN:  obj_type = sfidtype(obj_id, obj_type)
```

SDidtype returns a value of type *hdf_idtype_t*, which can be one of the following:

`NOT_SDAPI_ID (or -1)`	`not an SD API identifier`
`SD_ID  (or 0)`	`SD identifier`
`SDS_ID (or 1)`	`SDS identifier`
`DIM_ID (or 2)`	`Dimension identifier`

SDidtype returns `NOT_SDAPI_ID` for either when *obj_id* is not a valid HDF identifier, or is a valid HDF identifier, but not one of the identifier types in the SD interface, which are SD identifier, SDS identifier, and dimension identifier.

The parameter of **SDidtype** is specified in Table 3N.

3.7.10. Determining whether an SDS is empty: SDcheckempty

SDcheckempty takes an SDS identifier, *sds_id*, as input, and returns a single parameter indicating whether the SDS is empty. The syntax of this routine is as follows:

```
C:         status = SDcheckempty(sds_id, emptySDS);

FORTRAN:   status = sfchempty(sds_id, emptySDS)
```

The output parameter, *emptySDS*, indicates whether the SDS is empty or non-empty.

SDcheckempty returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDcheckempty** are specified in Table 3N.

TABLE 3N **SDreftoindex, SDidtoref, SDidtype, and SDcheckempty Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDreftoindex [int32] **(sfref2index)**	sd_id	int32	integer	SD interface identifier
	sds_ref	int32	integer	SDS reference number
SDidtoref [int32] **(sfid2ref)**	sds_id	int32	integer	SDS identifier
SDidtype [hdf_idtype_t] **(sfidtype)**	obj_id	int32	integer	An object identifier
SDcheckempty [int32] **(sfchempty)**	sds_id	int32	integer	SDS identifier
	emptySDS	intn *	integer	SDS status indicator (empty, not empty)

EXAMPLE 11. **Locating an SDS by Its Name.**

This example uses the routine **SDnametoindex/sfn2index** to locate the SDS with the specified name and then reads the data from it.

C:

```
#include "mfhdf.h"
```

```
#define FILE_NAME     "SDS.hdf"
#define SDS_NAME      "SDStemplate"
#define WRONG_NAME    "WrongName"
#define X_LENGTH      5
#define Y_LENGTH      16

main( )
{
   /************************* Variable declaration *************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 start[2], edges[2];
   int32 data[Y_LENGTH][X_LENGTH];
   int   i, j;

   /******************** End of variable declaration **********************/

   /*
   * Open the file for reading and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_READ);

   /*
   * Find index of the data set with the name specified in WRONG_NAME.
   * Error condition occurs, since the data set with that name does not exist
   * in the file.
   */
   sds_index = SDnametoindex (sd_id, WRONG_NAME);
   if (sds_index == FAIL)
   printf ("Data set with the name \"WrongName\" does not exist\n");

   /*
   * Find index of the data set with the name specified in SDS_NAME and use
   * the index to select the data set.
   */
   sds_index = SDnametoindex (sd_id, SDS_NAME);
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Set elements of the array start to 0, elements of the array edges to
   * SDS dimensions, and use NULL for stride argument in SDreaddata to read
   * the entire data.
   */
   start[0] = 0;
   start[1] = 0;
   edges[0] = Y_LENGTH;
   edges[1] = X_LENGTH;

   /*
   * Read the entire data into the buffer named data.
   */
   status = SDreaddata (sds_id, start, NULL, edges, (VOIDP)data);

   /*
   * Print 10th row; the following numbers should be displayed:
   *
   *            10 1000 12 13 14
   */
   for (j = 0; j < X_LENGTH; j++) printf ("%d ", data[9][j]);
   printf ("\n");
   /*
   * Terminate access to the data set.
   */
   status = SDendaccess (sds_id);

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  locate_by_name
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*11 SDS_NAME
      character*9  WRONG_NAME
      integer      X_LENGTH, Y_LENGTH
      parameter   (FILE_NAME  = 'SDS.hdf',
     +             SDS_NAME   = 'SDStemplate',
     +             WRONG_NAME = 'WrongName',
     +             X_LENGTH = 5,
     +             Y_LENGTH = 16)
      integer      DFACC_READ, DFNT_INT32
      parameter   (DFACC_READ = 1,
     +             DFNT_INT32 = 24)

C
C     Function declaration.
C
      integer sfstart, sfn2index, sfselect, sfrdata, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer start(2), edges(2), stride(2)
      integer data(X_LENGTH, Y_LENGTH)
      integer j
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
C
C     Find index of the data set with the name specified in WRONG_NAME.
C     Error condition occurs, since a data set with this name
C     does not exist in the file.
C
      sds_index = sfn2index(sd_id, WRONG_NAME)
      if (sds_index .eq. -1) then
        write(*,*) "Data set with the name ", WRONG_NAME,
     +             " does not exist"
      endif
C
C     Find index of the data set with the name specified in SDS_NAME
C     and use the index to attach to the data set.
C
```

```
      sds_index = sfn2index(sd_id, SDS_NAME)
      sds_id    = sfselect(sd_id, sds_index)
C
C     Set elements of start array to 0, elements of edges array
C    to SDS dimensions, and elements of stride array to 1 to read entire data.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Read entire data into array named data.
C
      status = sfrdata(sds_id, start, stride, edges, data)
C
C     Print 10th column; the following numbers should be displayed:
C
C           10 1000 12 13 14
C
      write(*,*) (data(j,10), j = 1, X_LENGTH)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.7.11. Creating SDS Arrays Containing Non-standard Length Data: SDsetnbitdataset

In version 4.0r1, HDF provided the routine **SDsetnbitdataset**, allowing HDF application to specify that an SDS array contains data of a non-standard length.

SDsetnbitdataset specifies that the data set identified by *sds_id* will contain data of a non-standard length, defined by the parameters *start_bit* and *bit_len*. Additional information about the non-standard bit length decoding are specified in the parameters *sign_ext* and *fill_one*. The syntax of **SDsetnbitdataset** is as follows:

```
C:          status = SDsetnbitdataset(sds_id, start_bit, bit_len, sign_ext,
                               fill_one);

FORTRAN:    status = sfsnbit(sds_id, start_bit, bit_len, sign_ext, fill_one)
```

Any length between 1 and 32 bits can be specified. After **SDsetnbitdataset** has been called for an SDS array, any read or write operations will convert between the new data length of the SDS array and the data length of the read or write buffer.

Bit lengths of all data types are counted from the right of the bit field starting with 0. In a bit field containing the values `01111011`, bits 2 and 7 are set to `0` and all the other bits are set to `1`.

The parameter *start_bit* specifies the left-most position of the variable-length bit field to be written. For example, in the bit field described in the preceding paragraph a parameter *start_bit* set to `4` would correspond to the fourth bit value of `1` from the right.

The parameter *bit_len* specifies the number of bits of the variable-length bit field to be written. This number includes the starting bit and the count proceeds toward the right end of the bit field - toward the lower-bit numbers. For example, starting at bit 5 and writing 4 bits of the bit field described in the preceding paragraph would result in the bit field `1110` being written to the data set. This would correspond to a *start_bit* value of `5` and a *bit_len* value of `4`.

The parameter *sign_ext* specifies whether to use the left-most bit of the variable-length bit field to sign-extend to the left-most bit of the data set data. For example, if 9-bit signed integer data is extracted from bits 17-25 and the bit in position 25 is `1`, then when the data is read back from disk, bits 26-31 will be set to `1`. Otherwise bit 25 will be `0` and bits 26-31 will be set to `0`. The *sign_ext* parameter can be set to `TRUE` (or `1`) or `FALSE` (or `0`); specify `TRUE` to sign-extend.

The parameter *fill_one* specifies whether to fill the "background" bits with the value `1` or `0`. This parameter is also set to either `TRUE` (or `1`) or `FALSE` (or `0`).

The "background" bits of a non-standard length data set are the bits that fall outside of the non-standard length bit field stored on disk. For example, if five bits of an unsigned 16-bit integer data set located in bits 5 to 9 are written to disk with the parameter *fill_one* set to `TRUE` (or `1`), then when the data is reread into memory bits 0 to 4 and 10 to 15 would be set to `1`. If the same 5-bit data was written with a *fill_one* value of `FALSE` (or `0`), then bits 0 to 4 and 10 to 15 would be set to `0`.

The operation on *fill_one* is performed before the operation on *sign_ext*. For example, using the *sign_ext* example above, bits 0 to 16 and 26 to 31 will first be set to the background bit value, and then bits 26 to 31 will be set to `1` or `0` based on the value of the 25th bit.

SDsetnbitdataset returns a positive value or `FAIL` (or `-1`). The parameters for **SDsetnbitdataset** are specified in Table 3O.

TABLE 3O **SDsetnbitdataset Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDsetnbitdataset [intn] **(sfsnbit)**	sds_id	int32	integer	Data set identifier
	start_bit	intn	integer	Leftmost bit of the field to be written
	bit_len	intn	integer	Length of the bit field to be written
	sign_ext	intn	integer	Sign-extend specifier
	fill_one	intn	integer	Background bit specifier

3.8. SDS Dimension and Dimension Scale Operations

The concept of dimensions is introduced in Section "*Required SDS Components"*. This section describes SD interface routines which store and retrieve information on dimensions and dimension scales. When a dimension scale is set for a dimension, the library stores the dimension and its associated information as an SDS array. In the following discussion, we will refer to that array (recall NetCDF) as a ***coordinate variable*** or ***dimension record***. The section concludes with consideration of related data sets and sharable dimensions.

3.8.1. Selecting a Dimension: SDgetdimid

SDS dimensions are uniquely identified by ***dimension identifiers***, which are assigned when a dimension is created. These dimension identifiers are used within a program to refer to a particular dimension, its scale, and its attributes. Before working with a dimension, a program must first obtain a dimension identifier by calling the **SDgetdimid** routine as follows:

```
C:          dim_id = SDgetdimid(sds_id, dim_index);

FORTRAN:    dim_id = sfdimid(sds_id, dim_index)
```

SDgetdimid takes two arguments, *sds_id* and *dim_index*, and returns a dimension identifier, *dim_id*. The argument *dim_index* is an integer from 0 to the number of dimensions minus 1. The number of dimensions in a data set is specified at the time the data set is created. Specifying a dimension index equal to or larger than the number of dimensions in the data set causes **SDgetdimid** to return a value of `FAIL` (or `-1`).

SDgetdimid returns a dimension identifier or `FAIL` (or `-1`). The parameters of **SDgetdimid** are specified in Table 3P.

Unlike file and data set identifiers, dimension identifiers cannot be explicitly closed.

3.8.2. Naming a Dimension: SDsetdimname

SDsetdimname assigns a name to a dimension. If two dimensions have the same name, they will be represented in the file by only one SDS. Therefore changes to one dimension will be reflected in the other. Naming dimensions is optional but encouraged. Dimensions that are not explicitly named by the user will have names generated by the HDF library. Use **SDdiminfo** to read existing dimension names. The syntax of **SDsetdimname** is as follows:

```
C:          status = SDsetdimname(dim_id, dim_name);

FORTRAN:    status = sfsdmname(dim_id, dim_name)
```

The argument *dim_id* in **SDsetdimname** is the dimension identifier returned by **SDgetdimid**. The parameter *dim_name* is a string of alphanumeric characters representing the name for the selected dimension. An attempt to rename a dimension using **SDsetdimname** will cause the old name to be deleted and a new one to be assigned.

Note that when naming dimensions the name of a particular dimension *must* be set before attributes are assigned; once the attributes have been set, the name must not be changed. In other words, **SDsetdimname** must only be called before any calls to **SDsetdimscale** (described in Section "*Writing Dimension Scales: SDsetdimscale"*), **SDsetattr** (described in Section "*Creating or Writing User-defined Attributes: SDsetattr"*) or **SDsetdimstrs** (described in Section "*Writing String Attributes of an SDS: SDsetdatastrs"*).

If the file being worked on was created by a pre-4.2.2 version of HDF, please refer to the *Important Note in Chapter 3* for information regarding a data corruption which might occur when a dimension is named the same as a one-dimensional data set.

SDsetdimname returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDsetdimname** are described in Table 3P.

TABLE 3P **SDgetdimid and SDsetdimname Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDgetdimid [int32] **(sfdimid)**	sds_id	int32	integer	Data set identifier
	dim_index	intn	integer	Dimension index
SDsetdimname [intn] **(sfsdmname)**	dim_id	int32	integer	Dimension identifier
	dim_name	char *	character*(*)	Dimension name

3.8.3. Old and New Dimension Implementations

Up to and including HDF version 4.0 beta1, dimensions were vgroup objects (described in Chapter 5, *Vgroups (V API)*, containing a single field vdata (described in Chapter 4, *Vdatas (VS API)*, with a class name of *DimVal0.0*. The vdata had the same number of records as the size of the dimension, which consisted of the values 0, 1, 2, . . . n - 1, where n is the size of the dimension. These values were not strictly necessary. Consider the case of applications that create large one dimensional data sets: the disk space taken by these unnecessary values nearly doubles the size of the HDF file. To avoid these situations, a new representation of dimensions was implemented for HDF version 4.0 beta 2 and later versions.

Dimensions are still vgroups in the new representation, but the vdata has only one record with a value of <*dimension size*> and the class name of the vdata has been changed to *DimVal0.1* to distinguish it from the old version.

Between HDF versions 4.0 beta1 and 4.1, the old and new dimension representations were written by default for each dimension created, and both representations were recognized by routines that operate on dimensions. From HDF version 4.1 forward, SD interface routines recognize only the new representation. Two compatibility mode routines, **SDsetdimval_comp** and **SDisdimval_bwcomp**, are provided to allow HDF programs to distinguish between the two dimension representations, or *compatibility modes*.

3.8.3.1. Setting the Future Compatibility Mode of a Dimension: SDsetdimval_comp

SDsetdimval_comp sets the compatibility mode for the dimension identified by the parameter *dim_id*. This operation determines whether the dimension will have the old and new representations or the new representation only. The syntax of **SDsetdimval_comp** is as follows:

```
C:          status = SDsetdimval_comp(dim_id, comp_mode);

FORTRAN:    status = sfsdmvc(dim_id, comp_mode)
```

The parameter *comp_mode* specifies the compatibility mode. It can be set to either `SD_DIMVAL_BW_COMP` (or `1`), which specifies compatible mode and that the old and new dimension representations will be written to the file, or `SD_DIMVAL_BW_INCOMP` (or `0`), which specifies incompatible mode and that only the new dimension representation will be written to file. As of HDF version 4.1r1, the default mode is backward-incompatible. Subsequent calls to **SDsetdimval_comp** will override the settings established in previous calls to the routine.

Unlimited dimensions are always backward compatible. Therefore **SDsetdimval_comp** takes no action when the dimension identified by *dim_id* is unlimited.

SDsetdimval_comp returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDsetdimval_comp** are specified in Table 3Q.

3.8.3.2. Determining the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp

SDisdimval_bwcomp determines whether the specified dimension has the old and new representations or the new representation only. The syntax of **SDisdimval_bwcomp** is as follows:

```
C:          comp_mode = SDisdimval_bwcomp(dim_id);

FORTRAN:    comp_mode = sfisdmvc(dim_id)
```

SDisdimval_bwcomp returns one of the three values: `SD_DIMVAL_BW_COMP` (or `1`), `SD_DIMVAL_BW_INCOMP` (or `0`), and `FAIL` (or `-1`). The interpretation of `SD_DIMVAL_BW_COMP` and `SD_DIMVAL_BW_INCOMP` are as that in the routine **SDsetdimval_comp**.

The parameters of **SDisdimval_bwcomp** are specified in Table 3Q.

TABLE 3Q **SDsetdimval_comp and SDisdimval_bwcomp Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetdimval_comp [intn] **(sfsdmvc)**	dim_id	int32	integer	Dimension identifier
	comp_mode	intn	integer	Compatibility mode
SDisdimval_bwcomp [intn] **(sfisdmvc)**	dim_id	int32	integer	Dimension identifier

3.8.4. Dimension Scales

A ***dimension scale*** can be thought of as a series of numbers demarcating intervals along a dimension. One scale is assigned per dimension. Users of netCDF can think of them as analogous to ***coordinate variables***. In the SDS data model, each dimension scale is a one-dimensional array with name and size equal to its assigned dimension name and size.

For example, if a dimension of length 6 named "depth" is assigned a dimension scale, its scale is a one-dimensional array of length 6 and is also assigned the name "depth". The name of the dimension will also appear as the name of the dimension scale.

Recall that when dimension scale is assigned to a dimension, the dimension is implemented as an SDS array with data being the data scale. Although dimension scales are conceptually different from SDS arrays, they are implemented as SDS arrays by the SD interface and are treated similarly by the routines in the interface. For example, when the **SDfileinfo** routine returns the number of data sets in a file, it includes dimension scales in that number. The **SDiscoordvar** routine (described in Section "*Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar"*) distinguishes SDS data sets from dimension scales.

3.8.4.1. Writing Dimension Scales: SDsetdimscale

SDsetdimscale stores scale information for the dimension identified by the parameter *dim_id*. The syntax of this routine is as follows:

```
C:          status = SDsetdimscale(dim_id, n_values, ntype, data);

FORTRAN:    status = sfsdscale(dim_id, n_values, ntype, data)
```

The argument *n_values* specifies the number of scale values along the specified dimension. For a fixed size dimension, *n_values* must be equal to the size of the dimension. The parameter *ntype* specifies the data type for the scale values and *data* is an array containing the scale values.

If the file being worked on was created by a pre-4.2.2 version of HDF, please refer to the *Important Note in Chapter 3* for information regarding a data corruption which might occur when a dimension is named the same as a one-dimensional data set.

SDsetdimscale returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are specified in Table 3R.

3.8.4.2. Obtaining Dimension Scale and Other Dimension Information: SDdiminfo

Before working with an existing dimension scale, it is often necessary to determine its characteristics. For instance, to allocate the proper amount of memory for a scale requires knowledge of its size and data type. **SDdiminfo** provides this basic information, as well as the name and the number of attributes for a specified dimension.

The syntax of this routine is as follows:

```
C:          status = SDdiminfo(dim_id, dim_name, &dim_size, &ntype, &n_attrs);

FORTRAN:    status = sfgdinfo(dim_id, dim_name, dim_size, ntype, n_attrs)
```

SDdiminfo retrieves and stores the dimension’s name, size, data type, and number of attributes into the parameters *dim_name*, *dim_size*, *ntype*, and *n_attrs*, respectively.

The parameter *dim_name* will contain the dimension name set by **SDsetdimname** or the default dimension name, *fakeDim[x]*, if **SDsetdimname** has not been called, where [*x*] denotes the dimension index. If the name is not desired, the parameter *dim_name* can be set to `NULL` in C or an empty string in FORTRAN-77.

An output value of 0 for the parameter *dim_size* indicates that the dimension specified by the parameter *dim_id* is unlimited. Use **SDgetinfo** to get the number of elements of the unlimited dimension.

If scale information is available for the specified dimension, i.e., **SDsetdimscale** has been called, the parameter *ntype* will contain the data type of the scale values; otherwise, *ntype* will contain `0`.

SDdiminfo returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are specified in Table 3R.

3.8.4.3. Reading Dimension Scales: SDgetdimscale

SDgetdimscale retrieves the scale values of a dimension. These values have previously been stored by **SDsetdimscale**. The syntax of this routine is as follows:

```
C:          status = SDgetdimscale(dim_id, data);

FORTRAN:    status = sfgdscale(dim_id, data)
```

SDgetdimscale reads all the scale values and stores them in the buffer *data* which is assumed to be sufficiently allocated to hold all the values. **SDdiminfo** should be used to determine whether the scale has been set for the dimension and to obtain the data type and the number of scale values for space allocation before calling **SDgetdimscale**. Refer to Section "*Obtaining Dimension Scale and Other Dimension Information: SDdiminfo"* for a discussion of **SDdiminfo**.

Note that it is not possible to read a subset of the scale values. **SDgetdimscale** returns all of the scale values stored with the given dimension.

The fact that **SDgetdimscale** returns `SUCCEED` should not be interpreted as meaning that scale values have been defined for the data set. This function should always be used with **SDdiminfo**, which is used first to determine whether a scale has been set, the number of scale values, their data type, etc. If **SDdiminfo** indicates that no scale values have been set, the values returned by **SDgetdimscale** in *data* should be ignored.

SDgetdimscale returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are specified in Table 3R.

TABLE 3R **SDsetdimscale, SDdiminfo, and SDgetdimscale Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDsetdimscale [intn] **(sfsdscale)**	dim_id	int32	integer	Dimension identifier
	n_values	int32	integer	Number of scale values
	ntype	int32	integer	Data type to be set for the scale values
	data	VOIDP	<valid data type>(*)	Buffer containing the scale values to be set
SDdiminfo [intn] **(sfgdinfo)**	dim_id	int32	integer	Dimension identifier
	dim_name	char *	character*(*)	Buffer for the dimension name
	n_values	int32 *	integer	Buffer for the dimension size
	ntype	int32 *	integer	Buffer for the scale data type
	n_attrs	int32 *	integer	Buffer for the attribute count
SDgetdimscale [intn] **(sfgdscale)**	dim_id	int32	integer	Dimension identifier
	data	VOIDP	<valid data type>(*)	Buffer for the scale values

EXAMPLE 12. **Setting and Retrieving Dimension Information.**

This example illustrates the use of the routines **SDgetdimid/sfdimid**, **SDsetdimname/sfsdmname**, **SDsetdimscale/sfsdscale**, **SDdiminfo/sfgdinfo**, and **SDgetdimscale/sfgdscale** to set and retrieve the dimensions names and dimension scales of the SDS created in Example 2 and modified in Examples 4 and 7.

C:

```
#include "mfhdf.h"


#define FILE_NAME     "SDS.hdf"
#define SDS_NAME      "SDStemplate"
#define DIM_NAME_X     "X_Axis"
#define DIM_NAME_Y     "Y_Axis"
#define NAME_LENGTH   6
#define X_LENGTH      5
#define Y_LENGTH      16
#define RANK          2

main( )
{
   /************************* Variable declaration **************************/

   int32   sd_id, sds_id, sds_index;
   intn    status;
   int32   dim_index, dim_id;
   int32   n_values, data_type, n_attrs;
   int16   data_X[X_LENGTH];    /* X dimension dimension scale */
   int16   data_X_out[X_LENGTH];
   float64 data_Y[Y_LENGTH];  /* Y dimension dimension scale */
```

```
float64 data_Y_out[Y_LENGTH];
char    dim_name[NAME_LENGTH];
int     i, j, nrow;

/******************** End of variable declaration **********************/

/*
* Initialize dimension scales.
*/
for (i=0; i < X_LENGTH; i++) data_X[i] = i;
for (i=0; i < Y_LENGTH; i++) data_Y[i] = 0.1 * i;

/*
* Open the file and initialize SD interface.
*/
sd_id = SDstart (FILE_NAME, DFACC_WRITE);

/*
* Get the index of the data set specified in SDS_NAME.
*/
sds_index = SDnametoindex (sd_id, SDS_NAME);

/*
* Select the data set corresponding to the returned index.
*/
sds_id = SDselect (sd_id, sds_index);

/* For each dimension of the data set specified in SDS_NAME,
*  get its dimension identifier and set dimension name
*  and dimension scale. Note that data type of dimension scale
*  can be different between dimensions and can be different from
*  SDS data type.
*/
for (dim_index = 0; dim_index < RANK; dim_index++)
{
    /*
    * Select the dimension at position dim_index.
    */
    dim_id = SDgetdimid (sds_id, dim_index);

    /*
    * Assign name and dimension scale to selected dimension.
    */
    switch (dim_index)
    {
 case 0: status = SDsetdimname (dim_id, DIM_NAME_Y);
             n_values = Y_LENGTH;
             status = SDsetdimscale (dim_id,n_values,DFNT_FLOAT64, \
                                     (VOIDP)data_Y);
         break;
 case 1: status = SDsetdimname (dim_id, DIM_NAME_X);
             n_values = X_LENGTH;
             status = SDsetdimscale (dim_id,n_values,DFNT_INT16, \
                                     (VOIDP)data_X);
         break;
 default: break;
    }

    /*
    * Get and display info about the dimension and its scale values.
    * The following information is displayed:
    *
    *         Information about 1 dimension:
        *        dimension name is Y_Axis
        *        number of scale values is 16
        *        dimension scale data type is float64
        *        number of dimension attributes is 0
        *
        *        Scale values are :
        *              0.000    0.100    0.200    0.300
        *              0.400    0.500    0.600    0.700
        *              0.800    0.900    1.000    1.100
        *              1.200    1.300    1.400    1.500
        *
        *        Information about 2 dimension:
        *        dimension name is X_Axis
        *        number of scale values is 5
        *        dimension scale data type is int16
        *        number of dimension attributes is 0
        *
        *        Scale values are :
        *              0  1  2  3  4
        */

        status = SDdiminfo (dim_id, dim_name, &n_values, &data_type, &n_attrs);
        printf ("Information about %d dimension:\n", dim_index+1);
        printf ("dimension name is %s\n", dim_name);
        printf ("number of scale values is %d\n", n_values);
        if( data_type == DFNT_FLOAT64)
        printf ("dimension scale data type is float64\n");
        if( data_type == DFNT_INT16)
        printf ("dimension scale data type is int16\n");
        printf ("number of dimension attributes is %d\n", n_attrs);
        printf ("\n");
        printf ("Scale values are :\n");
        switch (dim_index)
        {
          case 0:  status = SDgetdimscale (dim_id, (VOIDP)data_Y_out);
                   nrow = 4;
                   for (i=0; i<n_values/nrow; i++ )
                   {
                       for (j=0; j<nrow; j++)
                           printf ("  %-6.3f", data_Y_out[i*nrow + j]);
                           printf ("\n");
                   }
                   break;
          case 1:  status = SDgetdimscale (dim_id, (VOIDP)data_X_out);
                   for (i=0; i<n_values; i++) printf ("  %d", data_X_out[i]);
                   break;
          default: break;
         }
         printf ("\n");
    } /*for dim_index */

    /*
    * Terminate access to the data set.
    */
    status = SDendaccess (sds_id);

    /*
    * Terminate access to the SD interface and close the file.
    */
    status = SDend (sd_id);
}
```

FORTRAN:

```
      program  dimension_info
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*11 SDS_NAME
      character*6  DIM_NAME_X
      character*6  DIM_NAME_Y
      integer      X_LENGTH, Y_LENGTH, RANK
      parameter   (FILE_NAME  = ’SDS.hdf’,
     +             SDS_NAME   = ’SDStemplate’,
     +             DIM_NAME_X  = ’X_Axis’,
     +             DIM_NAME_Y  = ’Y_Axis’,
     +             X_LENGTH = 5,
     +             Y_LENGTH = 16,
     +             RANK     = 2)
      integer      DFACC_WRITE, DFNT_INT16, DFNT_FLOAT64
      parameter   (DFACC_WRITE   = 2,
     +             DFNT_INT16   = 22,
     +             DFNT_FLOAT64 = 6)

C
C     Function declaration.
C
      integer sfstart, sfn2index, sfdimid, sfgdinfo
      integer sfsdscale, sfgdscale, sfsdmname, sfendacc
      integer sfend, sfselect
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer dim_index, dim_id
      integer n_values, n_attrs, data_type
      integer*2 data_X(X_LENGTH)
      integer*2 data_X_out(X_LENGTH)
      real*8    data_Y(Y_LENGTH)
      real*8    data_Y_out(Y_LENGTH)
      character*6 dim_name
      integer   i
C
C**** End of variable declaration ************************************
C
C
C     Initialize dimension scales.
C
      do 10 i = 1, X_LENGTH
         data_X(i) = i - 1
10     continue

      do 20 i = 1, Y_LENGTH
         data_Y(i) = 0.1 * (i - 1)
20     continue
C
C     Open the file and initialize SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Get the index of the data set with the name specified in SDS_NAME.
C
      sds_index = sfn2index(sd_id, SDS_NAME)
```

```
C
C     Select the data set corresponding to the returned index.
C
      sds_id = sfselect(sd_id, sds_index)
C
C     For each dimension of the data set,
C     get its dimension identifier and set dimension name
C     and dimension scales. Note that data type of dimension scale can
C     be different between dimensions and can be different from SDS data type.
C
      do 30 dim_index = 0, RANK - 1
C
C        Select the dimension at position dim_index.
C
         dim_id = sfdimid(sds_id, dim_index)
C
C        Assign name and dimension scale to the dimension.
C
         if (dim_index .eq. 0) then
            status = sfsdmname(dim_id, DIM_NAME_X)
            n_values = X_LENGTH
            status = sfsdscale(dim_id, n_values, DFNT_INT16, data_X)
         end if
         if (dim_index .eq. 1) then
            status = sfsdmname(dim_id, DIM_NAME_Y)
            n_values = Y_LENGTH
            status = sfsdscale(dim_id, n_values, DFNT_FLOAT64, data_Y)
         end if
C
C      Get and display information about dimension and its scale values.
C      The following information is displayed:
C
C                     Information about 1 dimension :
C                     dimension name is X_Axis
C                     number of scale values is  5
C                     dimension scale data type is int16
C
C                     number of dimension attributes is   0
C                     Scale values are:
C                         0  1  2  3  4
C
C                     Information about 2 dimension :
C                     dimension name is Y_Axis
C                     number of scale values is  16
C                     dimension scale data type is float64
C                     number of dimension attributes is   0
C
C                     Scale values are:
C                         0.000      0.100      0.200      0.300
C                         0.400      0.500      0.600      0.700
C                         0.800      0.900      1.000      1.100
C                         1.200      1.300      1.400      1.500
C
       status = sfgdinfo(dim_id, dim_name, n_values, data_type, n_attrs)
C
       write(*,*) "Information about ", dim_index+1," dimension :"
       write(*,*) "dimension name is ", dim_name
       write(*,*) "number of scale values is", n_values
       if (data_type. eq. 22) then
           write(*,*) "dimension scale data type is int16"
       endif
       if (data_type. eq. 6) then
           write(*,*) "dimension scale data type is float64"
```

```
             endif
             write(*,*) "number of dimension attributes is ", n_attrs
      C
             write(*,*) "Scale values are:"
             if (dim_index .eq. 0) then
                status = sfgdscale(dim_id, data_X_out)
                write(*,*) (data_X_out(i), i= 1, X_LENGTH)
             endif
             if (dim_index .eq. 1)  then
                status = sfgdscale(dim_id, data_Y_out)
                write(*,100) (data_Y_out(i), i= 1, Y_LENGTH)
      100       format(4(1x,f10.3)/)
             endif
      30      continue
      C
      C     Terminate access to the data set.
      C
            status = sfendacc(sds_id)
      C
      C     Terminate access to the SD interface and close the file.
      C
            status = sfend(sd_id)
            end
```

3.8.4.4. Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar

The HDF library stores SDS dimensions as data sets. HDF therefore provides the routine **SDiscoordvar** to determine whether a particular data set contains the data of an SDS or an SDS dimension with dimension scale or attribute assigned to it. The syntax of **SDiscoordvar** this routine is as follows:

```
C:          status = SDiscoordvar(sds_id);

FORTRAN:    status = sfiscvar(sds_id)
```

If the data set, identified by the parameter *sds_id*, contains the dimension data, a subsequent call to **SDgetinfo** will fill the specified arguments with information about a dimension, rather than a data set.

If the file being worked on was created by a pre-4.2.2 version of HDF, please refer to the *Important Note in Chapter 3* for information regarding a data corruption which might occur when a dimension is named the same as a one-dimensional SDS.

SDiscoordvar returns `TRUE` (or `1`) if the specified data set represents a dimension scale and `FALSE` (or `0`), otherwise. This routine is further defined in Table 3S.

TABLE 3S

SDiscoordvar Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDiscoordvar [intn] **(sfiscvar)**	sds_id	int32	integer	Data set identifier

EXAMPLE 13.

Distinguishing a Dimension Scale from a Data Set in a File.

This example illustrates the use of the routine **SDiscoordvar/sfiscvar** to determine whether the selected SDS array is a data set or a dimension stored as an SDS array (coordinate variable) (see discussion in Section 3.8.4) and displays the name of the data set or dimension.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDS.hdf"

main( )
{
   /************************* Variable declaration **************************/

   int32 sd_id, sds_id, sds_index;
   intn  status;
   int32 rank, data_type, dim_sizes[MAX_VAR_DIMS];
   int32 n_datasets, n_file_attr, n_attrs;
   char  sds_name[MAX_NC_NAME];

   /********************* End of variable declaration **********************/

   /*
   * Open the file and initialize the SD interface.
   */
   sd_id = SDstart(FILE_NAME, DFACC_READ);

   /*
   * Obtain information about the file.
   */
   status = SDfileinfo(sd_id, &n_datasets, &n_file_attr);

   /* Get information about each SDS in the file.
   *  Check whether it is a coordinate variable, then display retrieved
   *  information.
   *  Output displayed:
   *
   *            SDS array with the name SDStemplate
   *            Coordinate variable with the name Y_Axis
   *            Coordinate variable with the name X_Axis
   *
   */
   for (sds_index=0; sds_index< n_datasets; sds_index++)
   {
       sds_id = SDselect (sd_id, sds_index);
       status = SDgetinfo(sds_id, sds_name, &rank, dim_sizes, &data_type,
&n_attrs);
       if (SDiscoordvar(sds_id))
          printf(" Coordinate variable with the name %s\n", sds_name);
```

```
            else
               printf(" SDS array with the name %s\n", sds_name);

         /*
         * Terminate access to the selected data set.
         */
         status = SDendaccess(sds_id);

         }

         /*
         * Terminate access to the SD interface and close the file.
         */
         status = SDend(sd_id);
      }
```

FORTRAN:

```
      program  sds_vrs_coordvar
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      parameter   (FILE_NAME = 'SDS.hdf')
      integer      DFACC_READ, DFNT_INT32
      parameter   (DFACC_READ = 1,
     +             DFNT_INT32 = 24)
      integer      MAX_VAR_DIMS
      parameter   (MAX_VAR_DIMS = 32)
C
C     Function declaration.
C
      integer sfstart, sfselect, sfiscvar, sffinfo, sfginfo
      integer sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer       sd_id, sds_id, sds_index, status
      integer       rank, data_type
      integer       n_datasets, n_file_attrs, n_attrs
      integer       dim_sizes(MAX_VAR_DIMS)
      character*256 sds_name
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
C
C     Obtain information about the file.
C
      status = sffinfo(sd_id, n_datasets, n_file_attrs)
C
C     Get information about each SDS in the file.
C     Check whether it is a coordinate variable, then display retrieved
C     information.
C     Output displayed:
C
C           SDS array with the name SDStemplate
C           Coordinate variable with the name X_Axis
C           Coordinate variable with the name Y_Axis
```

```
C
       do 10 sds_index = 0, n_datasets-1
          sds_id = sfselect(sd_id, sds_index)
          status = sfginfo(sds_id, sds_name, rank, dim_sizes,
     +                     data_type, n_attrs)
          status = sfiscvar(sds_id)
          if (status .eq. 1) then
              write(*,*) "Coordinate variable with the name ",
     +        sds_name(1:6)
          else
              write(*,*) "SDS array with the name ",
     +        sds_name(1:11)
          endif
C
C         Terminate access to the data set.
C
          status = sfendacc(sds_id)
10     continue
C
C      Terminate access to the SD interface and close the file.
C
       status = sfend(sd_id)
       end
```

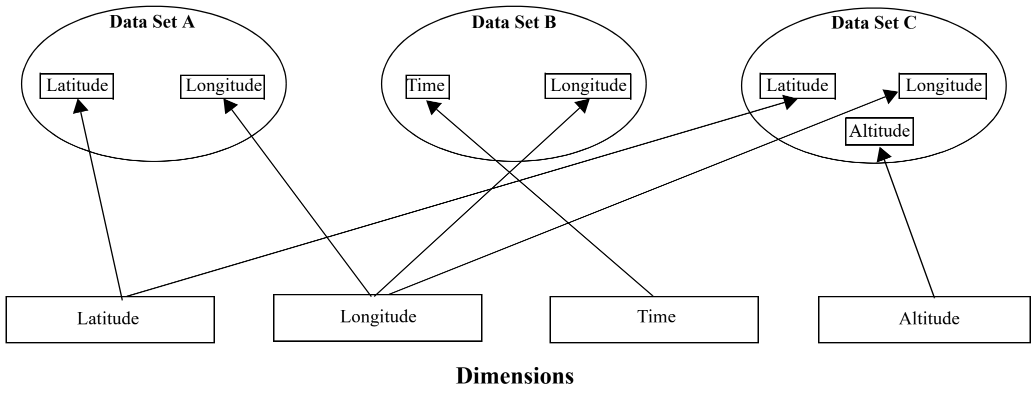
3.8.5. Related Data Sets

SD data sets with one or more dimensions with the same name and size are considered to be related. Examples of related data sets are cross-sections from the same simulation, frames in an animation, or images collected from the same apparatus. HDF attempts to preserve this relationship by unifying their dimension scales and attributes. To understand how related data sets are handled, it is necessary to understand what dimension records are and how they are created.

In the SD interface, dimension records are only created for dimensions of a unique name and size. To illustrate this, consider a case where there are three scientific data sets, each representing a unique variable, in an HDF file. (See Figure 3c) The first two data sets have two dimensions each and the third data set has three dimensions. There are a total of four dimensions in the file and the name mapping between the data sets and the dimensions are shown in the figure. Note that if, for example, the creation of a second dimension named "Altitude" is attempted and the size of the dimension is different from the existing dimension named "Altitude", an error condition will be generated.

As expected, assigning a dimension attribute to dimension 1 of either data set will create the required dimension scale and assign the appropriate attribute. However, because related data sets share dimension records, they also share dimension attributes. Therefore, it is impossible to assign an attribute to a dimension without assigning the same attribute to all dimensions of identical name and size, either within one data set or related data sets.

FIGURE 3c **Dimension Records and Attributes Shared Between Related Data Sets**



3.9. User-defined Attributes

User-defined attributes are defined by the calling program and contain auxiliary information about a file, SDS array, or dimension. This auxiliary information is sometimes called ***metadata*** because it is data about data. There are two ways to store metadata: as user-defined attributes or as predefined attributes.

Attributes take the form *label=value*, where *label* is a character string containing `H4_MAX_NC_NAME` (or `256`) or fewer characters and *value* contains one or more entries of the same data type as defined at the time the attribute is created. Attributes can be attached to files, data sets, and dimensions. These are referred to, respectively, as ***file attributes***, ***data set attributes***, and ***dimension attributes***:

- ***File attributes*** describe an entire file. They generally contain information pertinent to all HDF data sets in the file and are sometimes referred to as ***global attributes***.
- ***Data set attributes*** describe individual SDSs. Because their scope is limited to an individual SDS, data set attributes are sometimes referred to as ***local attributes***.
- ***Dimension attributes*** provide information applicable to an individual SDS dimension. It is possible to assign a unit to one dimension in a data set without assigning a unit to the remaining dimensions.

For each attribute, an ***attribute count*** is maintained that identifies the number of values in the attribute. Each attribute has a unique ***attribute index***, the value of which ranges from 0 to the total number of attributes minus 1. The attribute index is used to locate an attribute in the object which the attribute is attached to. Once the attribute is identified, its values and information can be retrieved.

The data types permitted for attributes are the same as those allowed for SDS arrays. SDS arrays with general attributes of the same name can have different data types. For example, the attribute *valid_range* specifying the valid range of data values for an array of 16-bit integers might be of type 16-bit integer, whereas the attribute *valid_range* for an array of 32-bit floats could be of type 32-bit floating-point integer.

Attribute names follow the same rules as dimension names. Providing meaningful names for attributes is important, however using standardized names may be necessary if generic applications and utility programs are to be used. For example, every variable assigned a unit should have an attribute named "*units*" associated with it. Furthermore, if an HDF file is to be used with software that recognizes "units" attributes, the values of the "units" attributes should be expressed in a conventional form as a character string that can be interpreted by that software.

The SD interface uses the same functions to access all attributes regardless of the objects they are assigned to. The difference between accessing a file, array, or dimension attribute lies in the use of identifiers. File identifiers, SDS identifiers, and dimension identifiers are used to respectively access file attributes, SDS attributes, and dimension attributes.

3.9.1. Creating or Writing User-defined Attributes: SDsetattr

SDsetattr creates or modifies an attribute for one of the objects: the file, the data set, or the dimension. If the attribute with the specified name does not exist, **SDsetattr** creates a new one. If the named attribute already exists, **SDsetattr** resets all the values that are different from those provided in its argument list. The syntax of this routine is as follows:

```
C:          status = SDsetattr(obj_id, attr_name, ntype, n_values, values);

FORTRAN:    status = sfsnatt(obj_id, attr_name, ntype, n_values, values)
    OR      status = sfscatt(obj_id, attr_name, ntype, n_values, values)
```

The parameter *obj_id* is the identifier of the HDF data object to which the attribute is assigned and can be a file identifier, SDS identifier, or dimension identifier. If *obj_id* specifies an SD interface identifier (*sd_id*), a global attribute will be created which applies to all objects in the file. If *obj_id* specifies a data set identifier (*sds_id*), an attribute will be attached only to the specified data set. If *obj_id* specifies a dimension identifier (*dim_id*), an attribute will be attached only to the specified dimension.

The parameter *attr_name* is an ASCII character string containing the name of the attribute. It represents the label in the *label* = *value* equation and can be no more than `H4_MAX_NC_NAME` (or `256`) characters. If this is set to the name of an existing attribute, the value portion of the attribute will be overwritten. Do not use **SDsetattr** to assign a name to a dimension, use **SDsetdimname** instead.

The arguments *ntype*, *n_values*, and *values* describe the right side of the *label* = *value* equation. The argument *values* contains one or more values of the same data type. The argument *ntype* contains any HDF supported data type (see Table 2F). The parameter *n_values* specifies the total number of values in the attribute.

There are two FORTRAN-77 versions of this routine: **sfsnatt** and **sfscatt**. The routine **sfsnatt** writes numeric attribute data and **sfscatt** writes character attribute data.

SDsetattr returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDsetattr** are further described in Table 3T.

EXAMPLE 14. **Setting Attributes.**

This example shows how the routines **SDsetattr/sfscatt/sfsnatt** are used to set the attributes of the file, data set, and data set dimension created in the Examples 2, 4, and 12.

C:

```
#include "mfhdf.h"

#define FILE_NAME      "SDS.hdf"
#define FILE_ATTR_NAME "File_contents"
#define SDS_ATTR_NAME  "Valid_range"
#define DIM_ATTR_NAME  "Dim_metric"

main( )
```

```
{
   /************************* Variable declaration *************************/

   int32   sd_id, sds_id, sds_index;
   intn    status;
   int32   dim_id, dim_index;
   int32   n_values;                /* number of values of the file, SDS or
                                       dimension attribute         */
   char8   file_values[] = "Storm_track_data";
                                   /* values of the file attribute */
   float32 sds_values[2] = {2., 10.};
                                   /* values of the SDS attribute  */
   char8   dim_values[]  = "Seconds";
                                  /* values of the dimension attribute */

   /********************* End of variable declaration **********************/

   /*
   * Open the file and initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_WRITE);

   /*
   * Set an attribute that describes the file contents.
   */
   n_values = 16;
   status = SDsetattr (sd_id, FILE_ATTR_NAME, DFNT_CHAR8, n_values,
                      (VOIDP)file_values);

   /*
   * Select the first data set.
   */
   sds_index = 0;
   sds_id = SDselect (sd_id, sds_index);

   /*
   * Assign attribute to the first SDS. Note that attribute values
   * may have different data type than SDS data.
   */
   n_values  = 2;
   status = SDsetattr (sds_id, SDS_ATTR_NAME, DFNT_FLOAT32, n_values,
                      (VOIDP)sds_values);

   /*
   * Get the the second dimension identifier of the SDS.
   */
   dim_index = 1;
   dim_id = SDgetdimid (sds_id, dim_index);

   /*
   * Set an attribute of the dimension that specifies the dimension metric.
   */
   n_values = 7;
   status = SDsetattr (dim_id, DIM_ATTR_NAME, DFNT_CHAR8, n_values,
                      (VOIDP)dim_values);

   /*
   * Terminate access to the data set.
   */
   status = SDendaccess (sds_id);

   /*
   * Terminate access to the SD interface and close the file.
   */
   status = SDend (sd_id);
}
```

FORTRAN:

```
      program  set_attribs
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*13 FILE_ATTR_NAME
      character*11 SDS_ATTR_NAME
      character*10 DIM_ATTR_NAME
      parameter   (FILE_NAME = 'SDS.hdf',
     +             FILE_ATTR_NAME = 'File_contents',
     +             SDS_ATTR_NAME  = 'Valid_range',
     +             DIM_ATTR_NAME  = 'Dim_metric')
      integer      DFACC_WRITE, DFNT_CHAR8, DFNT_FLOAT32
      parameter   (DFACC_WRITE = 2,
     +             DFNT_CHAR8  = 4,
     +             DFNT_FLOAT32 = 5)
C
C     Function declaration.
C
      integer sfstart, sfscatt, sfsnatt, sfselect, sfdimid
      integer sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer sd_id, sds_id, sds_index, status
      integer dim_id, dim_index
      integer n_values
      character*16 file_values
      real         sds_values(2)
      character*7  dim_values
      file_values   = 'Storm_track_data'
      sds_values(1) = 2.
      sds_values(2) = 10.
      dim_values    = 'Seconds'
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize the SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Set an attribute that describes the file contents.
C
      n_values = 16
      status = sfscatt(sd_id, FILE_ATTR_NAME, DFNT_CHAR8, n_values,
     +                 file_values)
C
C     Select the first data set.
C
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)
C
C     Assign attribute to the first SDS. Note that attribute values
C     may have different data type than SDS data.
C
```

```
      n_values = 2
      status = sfsnatt(sds_id, SDS_ATTR_NAME, DFNT_FLOAT32, n_values,
     +                 sds_values)
C
C     Get the identifier for the first dimension.
C
      dim_index = 0
      dim_id = sfdimid(sds_id, dim_index)
C
C     Set an attribute to the dimension that specifies the
C     dimension metric.
C
      n_values = 7
      status = sfscatt(dim_id, DIM_ATTR_NAME, DFNT_CHAR8, n_values,
     +                 dim_values)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)

      end
```

3.9.2. Querying User-defined Attributes: SDfindattr and SDattrinfo

Given a file, SDS, or dimension identifier and an attribute name, **SDfindattr** returns a valid attribute index if the corresponding attribute exists. The attribute index can then be used to retrieve information about the attribute or its values. Given a file, SDS, or dimension identifier and a valid attribute index, **SDattrinfo** retrieves the information about the corresponding attribute if it exists.

The syntax for **SDfindattr** and **SDattrinfo** are as follows:

```
C:        attr_index = SDfindattr(obj_id, attr_name);
          status = SDattrinfo(obj_id, attr_index, attr_name, &ntype, &n_val-
                          ues);

FORTRAN:  attr_index = sffattr(obj_id, attr_name)
          status = sfgainfo(obj_id, attr_index, attr_name, ntype, n_values)
```

SDfindattr returns the index of the attribute, which belongs to the object identified by the parameter *obj_id*, and whose name is specified by the parameter *attr_name*.

The parameter *obj_id* can be either an SD interface identifier (*sd_id*), a data set identifier (*sds_id*), or a dimension identifier (*dim_id*). **SDfindattr** is case-sensitive in searching for the name specified by the parameter *attr_name* and does not accept wildcards as part of that name.

SDattrinfo retrieves the attribute's name, data type, and number of values into the parameters *attr_name*, *ntype*, and *n_values*, respectively.

The parameter *attr_index* specifies the relative position of the attribute within the specified object. An attribute index may also be determined by either keeping track of the number and order of attributes as they are written or dumping the contents of the file using the HDF dumping utility, **hdp**, which is described in Chapter 15, *HDF Command-line Utilities*.

SDfindattr returns an attribute index or a value of `FAIL` (or `-1`). **SDattrinfo** returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDfindattr** and **SDattrinfo** are further described in Table 3T.

3.9.3. Reading User-defined Attributes: SDreadattr

Given a file, SDS, or dimension identifier and an attribute index, **SDreadattr** reads the values of an attribute that belongs to either a file, an SDS, or a dimension. The syntax of this routine is as follows:

```
C:         status = SDreadattr(obj_id, attr_index, values);

FORTRAN:   status = sfrattr(obj_id, attr_index, values)
    OR     status = sfrnatt(obj_id, attr_index, values)
    OR     status = sfrcatt(obj_id, attr_index, values)
```

SDreadattr stores the attribute values in the buffer *values*, which is assumed to be sufficiently allocated. The size of the buffer must be at least *n_values*sizeof (ntype)* bytes long, where *n_values* and *ntype* are the number of attribute values and their type. The values of *n_values* and *ntype* can be retrieved using **SDattrinfo**. Note that the size of the data type must be determined at the local machine where the application is running. **SDreadattr** will also read attributes and annotations created by the DFSD interface.

The parameter *obj_id* can be either an SD interface identifier (*sd_id*), a data set identifier (*sds_id*), or a dimension identifier (*dim_id*).

The parameter *attr_index* specifies the relative position of the attribute within the specified object. An attribute index may also be determined by either keeping track of the number and order of attributes as they are written or dumping the contents of the file using the HDF dumping utility, **hdp**, which is described in Chapter 15, *HDF Command-line Utilities*.

There are three FORTRAN-77 versions of this routine: **sfrattr**, **sfrnatt**, and **sfrcatt**. The routine **sfrattr** reads data of all valid data types, **sfrnatt** reads numeric attribute data and **sfrcatt** reads character attribute data.

SDreadattr returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDreadattr** are further described in Table 3T.

TABLE 3T

SDsetattr, SDfindattr, SDattrinfo, and SDreadattr Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetattr [intn] **(sfsnatt/ sfscatt)**	sd_id, sds_id or dim_id	int32	integer	SD interface, data set, or dimension identifier
	attr_name	char *	character*(*)	Name of the attribute
	ntype	int32	integer	Data type of the attribute
	n_values	int32	integer	Number of values in the attribute
	values	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer containing the data to be written
SDfindattr [int32] **(sffattr)**	sd_id, sds_id or dim_id	int32	integer	SD interface, data set, or dimension identifier
	attr_name	char *	character*(*)	Attribute name
SDattrinfo [intn] **(sfgainfo)**	sd_id, sds_id or dim_id	int32	integer	SD interface, data set, or dimension identifier
	attr_index	int32	integer	Index of the attribute to be read
	attr_name	char *	character*(*)	Buffer for the name of the attribute
	ntype	int32 *	integer	Buffer for the data type of the values in the attribute
	n_values	int32 *	integer	Buffer for the total number of values in the attribute
SDreadattr [intn] **(sfrattr/ sfrnatt/ sfrcatt)**	sd_id, sds_id or dim_id	int32	integer	SD interface, data set, or dimension identifier
	attr_index	int32	integer	Index of the attribute to be read
	values	VOIDP	<valid data type>(*)/ <valid numeric data type>(*)/ character*(*)	Buffer for the attribute values

EXAMPLE 15.

Reading Attributes.

This example uses the routines **SDfindattr/sffattr**, **SDattrinfo/sfgainfo**, and **SDreadattr/sfrattr** to find and read attributes of the file, data set, and data set dimension created in the Example 14.

C:

```
#include "mfhdf.h"

#define FILE_NAME      "SDS.hdf"
#define FILE_ATTR_NAME "File_contents"
#define SDS_ATTR_NAME  "Valid_range"
#define DIM_ATTR_NAME  "Dim_metric"

main( )
{
   /************************* Variable declaration **************************/

   int32   sd_id, sds_id, dim_id;
   intn    status;
   int32   attr_index, data_type, n_values;
   char    attr_name[MAX_NC_NAME];
   int8    *file_data;
   int8    *dim_data;
   float32 *sds_data;
   int     i;
```

```
/********************* End of variable declaration ***********************/

/*
* Open the file and initialize SD interface.
*/
sd_id = SDstart (FILE_NAME, DFACC_READ);

/*
* Find the file attribute defined by FILE_ATTR_NAME.
*/
attr_index = SDfindattr (sd_id, FILE_ATTR_NAME);

/*
* Get information about the file attribute. Note that the first
* parameter is an SD interface identifier.
*/
status = SDattrinfo (sd_id, attr_index, attr_name, &data_type, &n_values);

/*
* Allocate a buffer to hold the attribute data.
*/
file_data = (int8 *)malloc (n_values * sizeof (data_type));

/*
* Read the file attribute data.
*/
status = SDreadattr (sd_id, attr_index, file_data);

/*
* Print out file attribute value.
*/
printf ("File attribute value is : %s\n", file_data);

/*
* Select the first data set.
*/
sds_id = SDselect (sd_id, 0);

/*
* Find the data set attribute defined by SDS_ATTR_NAME. Note that the
* first parameter is a data set identifier.
*/
attr_index = SDfindattr (sds_id, SDS_ATTR_NAME);

/*
* Get information about the data set attribute.
*/
status = SDattrinfo (sds_id, attr_index, attr_name, &data_type, &n_values);

/*
* Allocate a buffer to hold the data set attribute data.
*/
sds_data = (float32 *)malloc (n_values * sizeof (data_type));

/*
* Read the SDS attribute data.
*/
status = SDreadattr (sds_id, attr_index, sds_data);

/*
* Print out SDS attribute data type and values.
*/
if (data_type == DFNT_FLOAT32)
```

```
              printf ("SDS attribute data type is : float32\n");
   printf ("SDS attribute values are :  ");
   for (i=0; i<n_values; i++) printf (" %f", sds_data[i]);
   printf ("\n");

   /*
   * Get the identifier for the second dimension of the SDS.
   */
   dim_id = SDgetdimid (sds_id, 1);

   /*
   * Find dimension attribute defined by DIM_ATTR_NAME.
   */
   attr_index = SDfindattr (dim_id, DIM_ATTR_NAME);

   /*
   * Get information about the dimension attribute.
   */
   status = SDattrinfo (dim_id, attr_index, attr_name, &data_type, &n_values);

   /*
   * Allocate a buffer to hold the dimension attribute data.
   */
   dim_data = (int8 *)malloc (n_values * sizeof (data_type));

   /*
   * Read the dimension attribute data.
   */
   status = SDreadattr (dim_id, attr_index, dim_data);

   /*
   * Print out dimension attribute value.
   */
   printf ("Dimensional attribute values is : %s\n", dim_data);

   /*
   * Terminate access to the data set and to the SD interface and
   * close the file.
   */
   status = SDendaccess (sds_id);
   status = SDend (sd_id);

   /*
   * Free all buffers.
   */
   free (dim_data);
   free (sds_data);
   free (file_data);

   /*   Output of this program is :
   *
   *    File attribute value is : Storm_track_data
   *    SDS attribute data type is : float32
   *    SDS attribute values are :   2.000000 10.000000
   *    Dimensional attribute values is : Seconds
   */
}
```

FORTRAN:

```
      program  attr_info
      implicit none
C
C     Parameter declaration.
C
      character*7  FILE_NAME
      character*13 FILE_ATTR_NAME
      character*11 SDS_ATTR_NAME
      character*10 DIM_ATTR_NAME
      parameter   (FILE_NAME = 'SDS.hdf',
     +             FILE_ATTR_NAME = 'File_contents',
     +             SDS_ATTR_NAME  = 'Valid_range',
     +             DIM_ATTR_NAME  = 'Dim_metric')
      integer      DFACC_READ, DFNT_FLOAT32
      parameter   (DFACC_READ   = 1,
     +             DFNT_FLOAT32 = 5)

C
C     Function declaration.
C
      integer sfstart, sffattr, sfgainfo, sfrattr, sfselect
      integer sfdimid, sfendacc, sfend
C
C**** Variable declaration *******************************************
C
      integer      sd_id, sds_id, dim_id
      integer      attr_index, data_type, n_values, status
      real         sds_data(2)
      character*20 attr_name
      character*16 file_data
      character*7  dim_data
      integer      i
C
C**** End of variable declaration ************************************
C
C
C     Open the file and initialize SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
C
C     Find the file attribute defined by FILE_ATTR_NAME.
C     Note that the first parameter is an SD interface identifier.
C
      attr_index = sffattr(sd_id, FILE_ATTR_NAME)
C
C     Get information about the file attribute.
C
      status = sfgainfo(sd_id, attr_index, attr_name, data_type,
     +         n_values)
C
C     Read the file attribute data.
C
      status = sfrattr(sd_id, attr_index, file_data)
C
C     Print file attribute value.
C
      write(*,*) "File attribute value is : ", file_data
C
C     Select the first data set.
C
      sds_id = sfselect(sd_id, 0)
C
C     Find the data set attribute defined by SDS_ATTR_NAME.
C     Note that the first parameter is a data set identifier.
```

```
C
      attr_index = sffattr(sds_id, SDS_ATTR_NAME)
C
C     Get information about the data set attribute.
C
      status = sfgainfo(sds_id, attr_index, attr_name, data_type,
     +          n_values)
C
C     Read the SDS attribute data.
C
      status = sfrattr(sds_id, attr_index, sds_data)

C
C     Print SDS attribute data type and values.
C
      if (data_type .eq. DFNT_FLOAT32)  then
         write(*,*) "SDS attribute data type is : float32 "
      endif
      write(*,*) "SDS attribute values are  : "
      write(*,*)  (sds_data(i), i=1, n_values)
C
C     Get the identifier for the first dimension of the SDS.
C
      dim_id = sfdimid(sds_id, 0)
C
C     Find the dimensional attribute defined by DIM_ATTR_NAME.
C     Note that the first parameter is a dimension identifier.
C
      attr_index = sffattr(dim_id, DIM_ATTR_NAME)
C
C     Get information about dimension attribute.
C
      status = sfgainfo(dim_id, attr_index, attr_name, data_type,
     +          n_values)
C
C     Read the dimension attribute data.
C
      status = sfrattr(dim_id, attr_index, dim_data)
C
C     Print dimension attribute value.
C
      write(*,*) "Dimensional attribute value is : ", dim_data
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)
C
C     Output of this program is :
C
C
C     File attribute value is : Storm_track_data
C     SDS attribute data type is : float32
C     SDS attribute values are  :
C         2.00000   10.00000
C      Dimensional attribute value is : Seconds
C
      end
```

3.10. Predefined Attributes

Predefined attributes use reserved names and in some cases predefined data type names. Predefined attributes are categorized as follows:

- ***Labels*** can be thought of as variable names. They are often used as keys in searches to find a particular predefined attribute.
- ***Units*** are a means of declaring the units pertinent to a specific discipline. A freely-available library of routines is available to convert between character string and binary forms of unit specifications and to perform useful operations on the binary forms. This library is used in some netCDF applications and is recommended for use with HDF applications. For more information, refer to the *netCDF User's Guide for C* which can be obtained at `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/`.
- ***Formats*** describe the format in which numeric values will be printed and/or displayed. The recommended convention is to use standard FORTRAN-77 notation for describing the data format. For example, "F7.2" means to display seven digits with two digits to the right of the decimal point.
- ***Coordinate systems*** contain information that should be used when interpreting or displaying the data. For example, the text strings "cartesian", "polar" and "spherical" are recommended coordinate system descriptions.
- ***Ranges*** define the maximum and minimum values of a selected valid range. The range may cover the entire data set, values outside the data set, or a subset of values within a data set. Because the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate as long as they are of the same data type as the SDS array.
- ***Fill value*** is the value used to fill the areas between non-contiguous writes to SDS arrays. For more information about fill values, refer to Section "*Fill Values and Fill Mode"*.
- ***Calibration*** stores scale and offset values used to create calibrated data in SDS arrays. When data are calibrated, they are typically reduced from floats, double, or large integers into 8-bit or 16-bit integers and "packed" into an appropriately sized array. After the scale and offset values are applied, the packed array will return to its original form.

Predefined attributes are useful because they establish conventions that applications can depend on and because they are understood by the HDF library without users having to define them. Predefined attributes also ensure backward compatibility with earlier versions of the HDF library. They can be assigned only to data sets and dimensions. Table 3U lists the predefined attributes and the types of object each attribute can be assigned to.

TABLE 3U **Predefined Attributes List**

HDF Data Object Type	Attribute Category	Attribute Name	Description
SDS Array or Dimension	`Label`	`long_name`	Name of the array
	`Unit`	`units`	Units used for all dimensions and data
	`Format`	`format`	Format for displaying dim scales and array values
SDS Array Only	`Coordinate System`	`coordsys`	Coordinate system used to interpret the SDS array
	`Range`	`valid_range`	Maximum and minimum values within a selected data range
	`Fill Value`	`__FillValue`	Value used to fill empty locations in an SDS array
	`Calibration`	`scale_factor`	Value by which each array value is to be multiplied
		`scale_factor_err`	Error introduced by scaling SDS array data
		`add_offset`	Value to which each array value is to be added
		`add_offset_err`	Error introduced by offsetting the SDS array data
		`calibrated_nt`	Data type of the calibrated data

While the following netCDF naming conventions are not predefined in HDF, they are highly recommended to promote consistency of information-sharing among generic applications. Refer to the *netCDF User's Guide for C* for further information.

- ***missing_value***: An attribute containing a value used to fill areas of an array not intended to contain either valid data or a fill value. The scope of this attribute is local to the array. An example of this would be a region where information is unavailable, as in a geographical grid containing ocean data. The part of the grid where there is land might not have any data associated with it and in such a case the *missing_value* value could be supplied. The *missing_value* attribute is different from the *_FillValue* attribute in that fill values are intended to indicate data that was expected but did not appear, whereas missing values are used to indicate data that were never expected.
- ***title***: A global file attribute containing a description of the contents of a file.
- ***history***: A global file attribute containing the name of a program and the arguments used to derive the file. Well-behaved generic filters (programs that take HDF or netCDF files as input and produce HDF or netCDF files as output) would be expected to automatically append their name and the parameters with which they were invoked to the history attribute of an input file.

3.10.1. Accessing Predefined Attributes

The SD interface provides two methods for accessing predefined attributes. The first method uses the general attribute routines for user-defined attributes described in Section "*User-defined Attributes*"; the second employs routines specifically designed for each attribute and will be discussed in the following sections. Although the general attribute routines work well and are recommended in most cases, the specialized attribute routines are sometimes easier to use, especially when reading or writing related predefined attributes. This is true for two reasons. First, because predefined attributes are guaranteed unique names, the attribute index is unnecessary. Second, attributes with several components may be read as a group. For example, using the SD routine designed to read the predefined calibration attribute returns all five components with a single call, rather than five separate calls.

There is one exception: unlike predefined data set attributes, predefined dimension attributes should be read or written using the specialized attribute routines only.

The predefined attribute parameters are described in Table 3V. Creating a predefined attribute with parameters different from these will produce unpredictable results when the attribute is read using the corresponding predefined-attribute routine.

TABLE 3V **Predefined Attribute Definitions**

Category	Attribute Name	Data Type	Number of Values	Attribute Description
Label	long_name	DFNT_CHAR8	String length	String
Unit	units	DFNT_CHAR8	String length	String
Format	format	DFNT_CHAR8	String length	String
Coordinate System	coordsys	DFNT_CHAR8	String length	String
Range	valid_range	<valid data type>	2	Minimum and maximum values in 2-element array
Fill Value	_FillValue	<valid data type>	1	Fill value
Calibration	scale_factor	DFNT_FLOAT64	1	Scale
	scale_factor_err	DFNT_FLOAT64	1	Scale error
	add_offset	DFNT_FLOAT64	1	Offset
	add_offset_err	DFNT_FLOAT64	1	Offset error
	calibrated_nt	DFNT_INT32	1	Data type

In addition to **SDreadattr**, **SDfindattr** and **SDattrinfo** are also valid general attribute routines to use when reading a predefined attribute. **SDattrinfo** is always useful for determining the size of an attribute whose value contains a string.

3.10.2. SDS String Attributes

This section describes the predefined string attributes of the SDSs and the next section describes those of the dimensions. Predefined string attributes of an SDS include the ***label***, ***unit***, ***format***, and ***coordinate system***.

3.10.2.1. Writing String Attributes of an SDS: SDsetdatastrs

SDsetdatastrs assigns the predefined string attributes label, unit, format, and coordinate system to an SDS array. The syntax of this routine is as follows:

```
C:          status = SDsetdatastrs(sds_id, label, unit, format, coord_system);

FORTRAN:    status = sfsdtstr(sds_id, label, unit, format, coord_system)
```

If you do not wish to set an attribute, set the corresponding parameter to `NULL` in C and an empty string in FORTRAN-77. **SDsetdatastrs** returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its arguments are further described in Table 3W.

3.10.2.2. Reading String Attributes of an SDS: SDgetdatastrs

SDgetdatastrs reads the predefined string attributes label, unit, format, and coordinate system from an SDS. These string attributes have previously been set by the routine **SDsetdatastrs**. The syntax of **SDgetdatastrs** is as follows:

```
C:          status = SDgetdatastrs(sds_id, label, unit, format, coord_system,
                          len);

FORTRAN:    status = sfgdtstr(sds_id, label, unit, format, coord_system, len)
```

SDgetdatastrs stores the predefined attributes into the parameters *label*, *unit*, *format*, and *coord_system*, which are character string buffers. If a particular attribute has not been set by **SDsetdatastrs**, the first character of the corresponding returned string will be `NULL` for C and `0` for FORTRAN-77. Each string buffer is assumed to be at least *len* characters long, including the space to hold the `NULL` termination character. If you do not wish to get a predefined attribute of this SDS, set the corresponding parameter to `NULL` in C and an empty string in FORTRAN-77.

SDgetdatastrs returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3W.

TABLE 3W **SDsetdatastrs and SDgetdatastrs Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetdatastrs [intn] **(sfsdtstr)**	sds_id	int32	integer	Data set identifier
	label	char *	character*(*)	Label for the data
	unit	char *	character*(*)	Definition of the units
	format	char *	character*(*)	Description of the data format
	coord_system	char *	character*(*)	Description of the coordinate system
SDgetdatastrs [intn] **(sfgdtstr)**	sds_id	int32	integer	Data set identifier
	label	char *	character*(*)	Buffer for the label
	unit	char *	character*(*)	Buffer for the description of the units
	format	char *	character*(*)	Buffer for the description of the data format
	coord_system	char *	character*(*)	Buffer for the description of the coordinate system
	len	intn	integer	Minimum length of the string buffers

3.10.3. String Attributes of Dimensions

Predefined string attributes of a dimension include ***label***, ***unit***, and ***format***. They adhere to the same definitions as those of the label, unit, and format strings for SDS attributes.

3.10.3.1. Writing a String Attribute of a Dimension: SDsetdimstrs

SDsetdimstrs assigns the predefined string attributes label, unit, and format to an SDS dimension and its scales. The syntax of this routine is as follows:

```
C:          status = SDsetdimstrs(dim_id, label, unit, format);

FORTRAN:    status = sfsdmstr(dim_id, label, unit, format)
```

The argument *dim_id* is the dimension identifier, returned by **SDgetdimid**, and identifies the dimension to which the attributes will be assigned. If you do not wish to set an attribute, set the corresponding parameter to `NULL` in C and an empty string in FORTRAN-77.

SDsetdimstrs returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3X.

3.10.3.2. Reading a String Attribute of a Dimension: SDgetdimstrs

SDgetdimstrs reads the predefined string attributes label, unit, and format from an SDS dimension. These string attributes have previously been set by the routine **SDsetdimstrs**. The syntax of **SDgetdimstrs** is as follows:

```
C:          status = SDgetdimstrs(dim_id, label, unit, format, len);

FORTRAN:    status = sfgdmstr(dim_id, label, unit, format, len)
```

SDgetdimstrs stores the predefined attributes of the dimension into the arguments *label*, *unit*, and *format*, which are character string buffers. If a particular attribute has not been set by **SDsetdimstrs**, the first character of the corresponding returned string will be `NULL` for C and `0` for FORTRAN-77. Each string buffer is assumed to be at least *len* characters long, including the space to hold the `NULL` termination character. If you do not wish to get a predefined attribute of this dimension, set the corresponding parameter to `NULL` in C and an empty string in FORTRAN-77.

SDgetdimstrs returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3X.

TABLE 3X **SDsetdimstrs and SDgetdimstrs Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetdimstrs [intn] **(sfsdmstr)**	dim_id	int32	integer	Dimension identifier
	label	char *	character*(*)	Label describing the specified dimension
	unit	char *	character*(*)	Units to be used with the specified dimension
	format	char *	character*(*)	Format to use when displaying the scale values
SDgetdimstrs [intn] **(sfgdmstr)**	dim_id	int32	integer	Dimension identifier
	label	char *	character*(*)	Buffer for the dimension label
	unit	char *	character*(*)	Buffer for the dimension unit
	format	char *	character*(*)	Buffer for the dimension format
	len	intn	integer	Maximum length of the string attributes

3.10.4. Range Attributes

The attribute ***range*** contains user-defined maximum and minimum values in a selected range. Since the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate. Also, because the maximum and minimum values are supposed to relate to the data set, it is assumed that they are of the same data type as the data.

3.10.4.1. Writing a Range Attribute: SDsetrange

SDsetrange sets the maximum and minimum range values for the data set identified by *sds_id* to the values provided by the parameters *max* and *min*. The syntax of the routine is as follows:

```
C:          status = SDsetrange(sds_id, max, min);

FORTRAN:    status = sfsrange(sds_id, max, min)
```

SDsetrange does not compute the maximum and minimum range values, it only stores the values as given. As a result, the maximum and minimum range values may not always reflect the actual maximum and minimum range values in the data set data. Recall that the type of max and min is assumed to be the same as that of the data set data.

SDsetrange returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3Y.

3.10.4.2. Reading a Range Attribute: SDgetrange

SDgetrange reads the maximum and minimum valid values of a data set. The syntax of this routine is as follows:

```
C:          status = SDgetrange(sds_id, &max, &min);

FORTRAN:    status = sfgrange(sds_id, max, min)
```

The maximum and minimum range values are stored in the parameters *max* and *min*, respectively, and must have previously been set by **SDsetrange**. Recall that the type of *max* and *min* is assumed to be the same as that of the data set data.

SDgetrange returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3Y.

TABLE 3Y **SDsetrange and SDgetrange Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetrange [intn] **(sfsrange)**	sds_id	int32	integer	Data set identifier
	max	VOIDP	<valid data type>	Maximum value to be stored
	min	VOIDP	<valid data type>	Minimum value to be stored
SDgetrange [intn] **(sfgrange)**	sds_id	int32	integer	Data set identifier
	max	VOIDP	<valid data type>	Buffer for the maximum value
	min	VOIDP	<valid data type>	Buffer for the minimum value

3.10.5. Fill Values and Fill Mode

A fill value is the value used to fill the spaces between non-contiguous writes to SDS arrays; it can be set with **SDsetfillvalue**. If a fill value is set before writing data to an SDS, the entire array is initialized to the specified fill value. By default, any location not subsequently overwritten with SDS data will contain the fill value.

A fill value must be of the same data type as the array to which it is written. To avoid conversion errors, use data-specific fill values instead of special architecture-specific values, such as infinity and ***Not-a-Number*** or ***NaN***.

A ***fill mode*** specifies whether the fill value is to be written to all the SDSs in the file; it can be set with **SDsetfillmode**.

Writing fill values to an SDS can involve more I/O overhead than is necessary, particularly in situations where the data set is to be contiguously filled with data before any read operation is made. In other words, writing fill values is only necessary when there is a possibility that the data set will be read before all gaps between writes are filled with data, i.e., before all elements in the array have been assigned values. Thus, for a file that has only data sets containing contiguous data, the fill mode should be set to `SD_NOFILL` (or `256`). Avoiding unnecessary filling can substantially increase the application performance.

For a non-contiguous data set, the array elements that have no actual data values must be filled with a fill value before the data set is read. Thus, for a file that has a non-contiguous data set, the fill mode should be set to `SD_FILL` (or `0`) and a fill value will be written to the all data sets in the file.

Note that, currently, **SDsetfillmode** specifies the fill mode of all data sets in the file. Thus, either all data sets are in `SD_FILL` mode or all data sets are in `SD_NOFILL` mode. However, when a specific SDS needs to be written with a fill value while others in the file do not, the following procedure can be used: set the fill mode to `SD_FILL`, write data to the data set requiring fill values, then set the fill mode back to `SD_NOFILL`. This procedure will produce one data set with fill values while the remaining data sets have no fill values.

3.10.5.1. Writing a Fill Value Attribute: SDsetfillvalue

SDsetfillvalue assigns a new value to the fill value attribute for an SDS array. The syntax of this routine is as follows:

```
C:          status = SDsetfillvalue(sds_id, fill_val);

FORTRAN:    status = sfsfill(sds_id, fill_val)
    OR      status = sfscfill(sds_id, fill_val)
```

The argument *fill_val* is the new fill value. It is recommended that you set the fill value before writing data to an SDS array, as calling **SDsetfillvalue** after data is written to an SDS array only changes the fill value attribute — it does not update the existing fill values.

There are two FORTRAN-77 versions of this routine: **sfsfill** and **sfscfill**. **sfsfill** writes numeric fill value data and **sfscfill** writes character fill value data.

SDsetfillvalue returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3Z.

3.10.5.2. Reading a Fill Value Attribute: SDgetfillvalue

SDgetfillvalue reads in the fill value of an SDS array as specified by a **SDsetfillvalue** call or its equivalent. The syntax of this routine is as follows:

```
C:          status = SDgetfillvalue(sds_id, &fill_val);

FORTRAN:    status = sfgfill(sds_id, fill_val)
    OR      status = sfgcfill(sds_id, fill_val)
```

The fill value is stored in the argument *fill_val* which is previously allocated based on the data type of the SDS data.

There are two FORTRAN-77 versions of this routine: **sfgfill** and **sfgcfill**. The **sfgfill** routine reads numeric fill value data and **sfgcfill** reads character fill value data.

SDgetfillvalue returns a value of `SUCCEED` (or `0`) if a fill value is retrieved successfully, or `FAIL` (or `-1`) otherwise, including when the fill value has not been set. The parameters of **SDgetfillvalue** are further described in Table 3Z.

3.10.5.3. Setting the Fill Mode for all SDSs in the Specified File: SDsetfillmode

SDsetfillmode sets the fill mode for all data sets contained in the file identified by the parameter *sd_id*. The syntax of **SDsetfillmode** is as follows:

```
C:          old_fmode = SDsetfillmode(sd_id, fill_mode);
```

```
FORTRAN:   old_fmode = sfsflmd(sd_id, fill_mode)
```

The argument *fill_mode* is the fill mode to be applied and can be set to either `SD_FILL` (or `0`) or `SD_NOFILL` (or `256`). `SD_FILL` specifies that fill values will be written to all SDSs in the specified file by default. If **SDsetfillmode** is never called before **SDsetfillvalue**, `SD_FILL` is the default fill mode. `SD_NOFILL` specifies that, by default, fill values will not be written to all SDSs in the specified file. This can be overridden for a specific SDS by calling **SDsetfillmode** then writing data to this data set before closing the file.

Note that whenever a file has been newly opened, or has been closed and then re-opened, the default `SD_FILL` fill mode will be in effect until it is changed by a call to **SDsetfillmode**.

SDsetfillmode returns the fill mode value before it is reset or a value of `FAIL` (or `-1`). The parameters of this routine are further described in Table 3Z.

TABLE 3Z **SDsetfillvalue, SDgetfillvalue, and SDsetfillmode Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDsetfillvalue [intn] **(sfsfill/ sfscfill)**	sds_id	int32	integer	Data set identifier
	fill_val	VOIDP	<valid numeric data type>/ character*(*)	Fill value to be set
SDgetfillvalue [intn] **(sfgfill/ sfgcfill)**	sds_id	int32	integer	Data set identifier
	fill_val	VOIDP	<valid numeric data type>/ character*(*)	Buffer for the fill value
SDsetfillmode [intn] **(sfsflmd)**	sd_id	int32	integer	SD interface identifier
	fill_mode	intn	integer	Fill mode to be set

3.10.6. Calibration Attributes

The ***calibration attributes*** are designed to store calibration information associated with data set data. When data is calibrated, the values in an array can be represented using a smaller data type than the original. For instance, an array containing data of type *float* could be stored as an array containing data of type 8- or 16-bit integer. Note that neither function performs any operation on the data set.

3.10.6.1. Setting Calibration Information: SDsetcal

SDsetcal stores the scale factor, offset, scale factor error, offset error, and the data type of the uncalibrated data set for the specified data set. The syntax of this routine is as follows:

```
C:         status = SDsetcal(sds_id, cal, cal_error, offset, off_err, ntype);

FORTRAN:   status = sfscal(sds_id, cal, cal_error, offset, off_err, ntype)
```

SDsetcal has six arguments; *sds_id*, *cal*, *cal_error*, *offset*, *off_err*, and *ntype*. The argument *cal* represents a single value that when multiplied against every value in the calibrated data array reproduces the original data array (assuming an *offset* of 0). The argument *offset* represents a single value that when subtracted from every value in the calibrated array reproduces the original data (assuming a *cal* of 1). The values of the calibrated data array relate to the values of the original data array according to the following equation:

```
orig_value = cal * (cal_value - offset)
```

In addition to *cal* and *offset*, **SDsetcal** also includes the scale and offset errors. The argument *cal_err* contains the potential error of the calibrated data due to scaling; *offset_err* contains the potential error for the calibrated data due to the offset.

SDsetcal returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Its parameters are further described in Table 3AA.

3.10.6.2. Reading Calibrated Data: SDgetcal

SDgetcal reads calibration attributes for an SDS array as previously written by **SDsetcal**. The syntax of this routine is as follows:

```
C:          status = SDgetcal(sds_id, &cal, &cal_error, &offset, &offset_err,
                               &ntype);

FORTRAN:    status = sfgcal(sds_id, cal, cal_error, offset, offset_err, ntype)
```

Because the HDF library does not actually apply calibration information to the data, **SDgetcal** can be called anytime before or after the data is read. If a calibration record does not exist, **SDgetcal** returns `FAIL`. **SDgetcal** takes six arguments: *sds_id*, *cal*, *cal_error*, *offset*, *offset_err*, and *ntype*. Refer to Section 3.10.6.1. for the description of these arguments.

SDgetcal returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDgetcal** are described in Table 3AA.

TABLE 3AA **SDsetcal and SDgetcal Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDsetcal [intn] **(sfscal)**	sds_id	int32	integer	Data set identifier
	cal	float64	real*8	Calibration factor
	cal_error	float64	real*8	Calibration error
	offset	float64	real*8	Uncalibrated offset
	offset_err	float64	real*8	Uncalibrated offset error
	ntype	int32	integer	Data type of uncalibrated data
SDgetcal [intn] **(sfgcal)**	sds_id	int32	integer	Data set identifier
	cal	float64 *	real*8	Calibration factor
	cal_error	float64 *	real*8	Calibration error
	offset	float64 *	real*8	Uncalibrated offset
	offset_err	float64 *	real*8	Uncalibrated offset error
	ntype	int32 *	integer	Data type of uncalibrated data

EXAMPLE 16. **Calibrating Data.**

Suppose the values in the calibrated array *cal_val* are the following integers:

```
cal_val[6] = {2, 4, 5, 11, 26, 81}
```

By applying the calibration equation *orig = cal * (cal_val - offset)* with *cal = 0.50* and *offset = -2000.0*, the calibrated array *cal_val[]* returns to its original floating-point form:

```
original_val[6] = {1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5}
```

3.11. Convenient Operations Related to File and Environment

The routines covered in this section provide methods for obtaining file name, object's type, length of object's name, and number of opened files allowed.

SDgetfilename retrieves the name of the file. **SDgetnamelen** retrieves the length of an object's name. **SDreset_maxopenfiles** resets the maximum number of files that can be opened at a time. **SDget_maxopenfiles** retrieves current limits on opened files. **SDget_numopenfiles** returns the number of files currently open.

These routines are described individually in the following subsections.

3.11.1. Obtaining the Name of a File: SDgetfilename

Given an identifier to a file, **SDgetfilename** returns its name via parameter *filename*. The user is repsonsible for allocating sufficient space to hold the file name. It can be at most `H4_MAX-_NC_NAME` characters in length. **SDgetnamelen** can be used to obtain the actual length of the name. The syntax of **SDgetfilename** is as follows:

```
C:          status = SDgetfilename(sd_id, filename);

FORTRAN:    status = sfgetfname(sd_id, filename)
```

SDgetfilename returns the length of the file name, without `'\0'`, or `FAIL` (or `-1`). The parameters of **SDgetfilename** are specified in Table 3AB.

3.11.2. Obtaining the Length of an HDF4 Object's Name: SDgetnamelen

SDgetnamelen retrieves the length of an object's name, given the object's identifier, *obj_id*. The object can be a file, a dataset, or a dimension. **SDgetnamelen** stores the length in the parameter *name_len*. The length does not include the '`\0`' character. The syntax of this routine is as follows:

```
C:          status = SDgetnamelen(obj_id, name_len);

FORTRAN:    status = sfgetnamelen(obj_id, name_len)
```

SDgetnamelen returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDgetname-len** are specified in Table 3AB.

TABLE 3AB **SDgetfilename and SDgetnamelen Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDgetfilename [intn] **(sfgetfname)**	sd_id	int32	integer	SD interface identifier
	filename	char*	character*(*)	Name of the file
SDgetnamelen [intn] **(sfgetnamelen)**	obj_id	int32	integer	HDF4 object identifier
	name_len	uint16*	integer	Length of the name

3.11.3. Resetting the Allowed Number of Opened Files:

SDreset_maxopenfiles

SDreset_maxopenfiles resets the maximum number of files can be opened at the same time. The syntax of the routine **SDsetcompress** is as follows:

```
C:          curr_max = SDreset_maxopenfiles(req_max);

FORTRAN:    curr_max = sfrmaxopenf(req_max)
```

Prior to release 4.2.2, the maximum number of files that can be opened at the same time was limited to 32. In HDF 4.2.2 and later versions, when this limit is reached, the library will increase it to the system limit minus 3 to account for stdin, stdout, and stderr.

This function can be called anytime to change the maximum number of open files allowed in HDF to *req_max*. If *req_max* is `0`, **SDreset_maxopenfiles** will simply return the current maximum number of open files allowed. If *req_max* exceeds system limit, **SDreset_maxopenfiles** will reset the maximum number of open files to the system limit, and return that value.

Furthermore, if the system maximum limit is reached, the library will push the error code `DFE_TOOMANY` onto the error stack. User applications can detect this after an **SDstart** fails.

SDreset_maxopenfiles returns the current maximum number of opened files allowed, or `FAIL` (or `-1`). The parameters of **SDreset_maxopenfiles** are specified in Table 3AC.

3.11.4. Obtaining Current Limits on Opened Files: SDget_maxopenfiles

SDget_maxopenfiles retrieves the current number of opened files allowed in HDF and the maximum number of opened files allowed on a system. The two parameters, curr_max and sys_limit, contain the two values, respectively. The syntax of this routine is as follows:

```
C:          status = SDget_maxopenfiles(curr_max, sys_limit);

FORTRAN:    status = sfgmaxopenf(cur_max, sys_limit)
```

SDget_maxopenfiles returns a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDget_maxopenfiles** are specified in Table 3AC.

3.11.5. Obtaining Number of Opened Files: SDget_numopenfiles

SDget_numopenfiles returns the number of files that are opened currently. The syntax of this routine is as follows:

```
C:          num_opened = SDget_numopenfiles();

FORTRAN:    num_opened = sfgnumopenf(cur_num)
```

SDget_numopenfiles returns the number of opened files or `FAIL` (or `-1`). The parameters of **SDget_numopenfiles** are specified in Table 3AC.

TABLE 3AC **SDreset_maxopenfiles, SDget_maxopenfiles, andSDget_numopenfiles Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDreset_maxopenfiles [intn] **(sfrmaxopenf)**	req_max	intn	integer	Requested maximum number of opened files
SDget_maxopenfiles [intn] **(sfgmaxopenf)**	curr_max	intn*	integer	Current number of open files allowed
	sys_limit	intn*	integer	Maximum number of open files allowed on a system
SDget_numopenfiles [intn] **(sfgnumopenf)**	curr_num	N/A	integer	Current number of open files. C function has no parameter

3.12. Chunked (or Tiled) Scientific Data Sets

NOTE: It is strongly encouraged that HDF users who wish to use the SD chunking routines first read the section on SD chunking in Chapter 14, *HDF Performance Issues*. In that section the concepts of chunking are explained, as well as their use in relation to HDF. As the ability to work with chunked data has been added to HDF functionality for the purpose of addressing specific performance-related issues, you should first have the necessary background knowledge to correctly determine how chunking will positively or adversely affect your application.

This section will refer to both "tiled" and "chunked" SDSs as simply ***chunked SDSs***, as tiled SDSs are the two-dimensional case of chunked SDSs.

3.12.1. Making an SDS a Chunked SDS: SDsetchunk

In HDF, an SDS must first be created as a generic SDS through the **SDcreate** routine, then **SDsetchunk** is called to make that generic SDS a chunked SDS. Note that there are two restrictions that apply to chunked SDSs. The maximum number of chunks in a single HDF file is 65,535 and a chunked SDS cannot contain an unlimited dimension. **SDsetchunk** sets the chunk size and the compression method for a data set. The syntax of **SDsetchunk** is as follows:

```
C:          status = SDsetchunk(sds_id, c_def, flag);

FORTRAN:    status = sfschnk(sds_id, dim_length, comp_type, comp_prm)
```

The chunking information is provided in the parameters *c_def* and *flag* in C, and the parameters *comp_type* and *comp_prm* in FORTRAN-77.

In C:

The parameter *c_def* has type `HDF_CHUNK_DEF` which is defined as follows:

```
typedef union hdf_chunk_def_u {
    int32 chunk_lengths[MAX_VAR_DIMS];
    struct {
        int32 chunk_lengths[MAX_VAR_DIMS];
        int32 comp_type;
        comp_info cinfo;
    } comp;
    struct {
        int32 chunk_lengths[MAX_VAR_DIMS];
        intn start_bit;
        intn bit_len;
        intn sign_ext;
```

```
            intn fill_one;
        } nbit;
    } HDF_CHUNK_DEF
```

Refer to the reference manual page for **SDsetcompress** for the definition of the structure *comp_info*.

The parameter *flag* specifies the type of the data set, i.e., if the data set is chunked or chunked and compressed with either RLE, Skipping Huffman, GZIP, Szip, or NBIT compression methods. Valid values of *flag* are `HDF_CHUNK` for a chunked data set, (`HDF_CHUNK | HDF_COMP`) for a chunked data set compressed with RLE, Skipping Huffman, GZIP, and Szip compression methods, and (`HDF_CHUNK | HDF_NBIT`) for a chunked NBIT-compressed data set.

There are three pieces of chunking and compression information which should be specified: chunking dimensions, compression type, and, if needed, compression parameters.

If the data set is chunked, i.e., *flag* value is `HDF_CHUNK`, then the elements of the array *chunk_lengths* in the union *c_def* (*c_def.chunk_lengths[]*) have to be initialized to the chunk dimension sizes.

If the data set is chunked and compressed using RLE, Skipping Huffman, GZIP, or Szip methods (i.e., *flag* value is set up to (`HDF_CHUNK | HDF_COMP`)), then the elements of the array `chunk_lengths` of the structure *comp* in the union *c_def* (`c_def.comp.chunk_lengths[]`) have to be initialized to the chunk dimension sizes.

If the data set is chunked and NBIT compression is applied (i.e., *flag* values is set up to (`HDF_CHUNK | HDF_NBIT`)), then the elements of the array `chunk_lengths` of the structure `nbit` in the union `c_def` (`c_def.nbit.chunk_lengths[]`) have to be initialized to the chunk dimension sizes.

The values of `HDF_CHUNK`, `HDF_COMP`, and `HDF_NBIT` are defined in the header file `hproto.h`.

Compression types are passed in the field *comp_type* of the structure *cinfo*, which is an element of the structure `comp` in the union *c_def* (`c_def.comp.cinfo.comp_type`). Valid compression types are: `COMP_CODE_RLE` for RLE, `COMP_CODE_SKPHUFF` for Skipping Huffman, `COMP_CODE_DEFLATE` for GZIP compression.

For Skipping Huffman, GZIP, and Szip compression methods, parameters are passed in corresponding fields of the structure *cinfo*. Specify skipping size for Skipping Huffman compression in the field `c_def.comp.cinfo.skphuff.skp_size`; this value cannot be less than 1. Specify deflate level for GZIP compression in the field `c_def.comp.cinfo.deflate_level`. Valid values of deflate levels are integers from 0 to 9 inclusive. Specify the Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset in the fields `c_info.szip.options_mask` and `c_info.szip.pixels_per_block`, respectively.

NBIT compression parameters are specified in the fields *start_bit*, *bit_len*, *sign_ext*, and *fill_one* in the structure *nbit* of the union *c_def*.

In FORTRAN-77:

The *dim_length* array specifies the chunk dimensions.

The *comp_type* parameter specifies the compression type. Valid compression types and their values are defined in the hdf.inc file, and are listed below.

`COMP_CODE_NONE` (or `0`) for uncompressed data
`COMP_CODE_RLE` (or `1`) for data compressed using the RLE compression algorithm
`COMP_CODE_NBIT` (or `2`) for data compressed using the NBIT compression algorithm
`COMP_CODE_SKPHUFF` (or `3`) for data compressed using the Skipping Huffman compression algorithm

`COMP_CODE_DEFLATE` (or `4`) for data compressed using the GZIP compression algorithm
`COMP_CODE_SZIP` (or `5`) for data compressed using the Szip compression algorithm

The parameter *comp_prm(1)* specifies the skipping size for the Skipping Huffman compression method and the deflate level for the GZIP compression method.

For Szip compression, the Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset must be specified in *comp_prm(1)* and *comp_prm(2)*, respectively.

comp_prm(1) =	`value of option_mask`
comp_prm(2) =	`value of pixels_per_-block`

For NBIT compression, the four elements of the array *comp_prm* correspond to the four NBIT compression parameters listed in the structure *nbit*. The array *comp_prm* should be initialized as follows:

comp_prm(1) =	`value of start_bit`
comp_prm(2) =	`value of bit_len`
comp_prm(3) =	`value of sign_ext`
comp_prm(4) =	`value of fill_one`

Refer to the description of the union `HDF_CHUNK_DEF` and of the routine **SDsetnbitdataset** for NBIT compression parameter definitions.

SDsetchunk returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Refer to Table 3AD and Table 3AE for the descriptions of the parameters of both versions.

TABLE 3AD **SDsetchunk Parameter List**

Routine Name [Return Type]	Parameter	Parameter Type	Description
		C	
SDsetchunk [intn]	sds_id	int32	Data set identifier
	c_def	HDF_CHUNK_DEF	Union containing information on how the chunks are to be defined
	flag	int32	Flag determining the behavior of the routine

TABLE 3AE **sfschnk Parameter List**

Routine Name	Parameter	Parameter Type	Description
		FORTRAN-77	
sfschnk	sds_id	integer	Data set identifier
	dim_length	integer(*)	Sizes of the chunk dimensions
	comp_type	integer	Compression type
	comp_prm	integer(*)	Array containing information needed by the compression algorithm

3.12.2. Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache

To maximize the performance of the HDF library routines when working with chunked SDSs, the library maintains a separate area of memory specifically for cached data chunks. **SDsetchunk-**

cache sets the maximum number of chunks of the specified SDS that are cached into this segment of memory. The syntax of **SDsetchunkcache** is as follows:

```
C:          status = SDsetchunkcache(sds_id, maxcache, flag);

FORTRAN:    status = sfscchnk(sds_id, maxcache, flag)
```

When the chunk cache has been filled, any additional chunks written to cache memory are cached according to the Least-Recently-Used (LRU) algorithm. This means that the chunk that has resided in the cache the longest without being reread or rewritten will be written over with the new chunk.

By default, when a generic SDS is made a chunked SDS, the parameter *maxcache* is set to the number of chunks along the fastest changing dimension. If needed, **SDsetchunkcache** can then be called again to reset the size of the chunk cache.

Essentially, the value of *maxcache* cannot be set to a value less than the number of chunks currently cached. If the chunk cache is *not* full, then the size of the chunk cache is reset to the new value of *maxcache* only if it is greater than the current number of chunks cached. If the chunk cache has been completely filled with cached data, **SDsetchunkcache** has already been called, and the value of the parameter *maxcache* in the current call to **SDsetchunkcache** is larger than the value of *maxcache* in the last call to **SDsetchunkcache**, then the value of *maxcache* is reset to the new value.

Currently the only allowed value of the parameter *flag* is `0`, which designates default operation. In the near future, the value `HDF_CACHEALL` will be provided to specify that the entire SDS array is to be cached.

SDsetchunkcache returns the maximum number of chunks that can be cached (the value of the parameter *maxcache*) if successful and `FAIL` (or `-1`) otherwise. The parameters of **SDsetchunkcache** are further described in Table 3AF.

TABLE 3AF **SDsetchunkcache Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDsetchunkcache [intn] **(sfscchnk)**	sds_id	int32	integer	Data set identifier
	maxcache	int32	integer	Maximum number of chunks to cache
	flag	int32	integer	Flag determining the default caching behavior

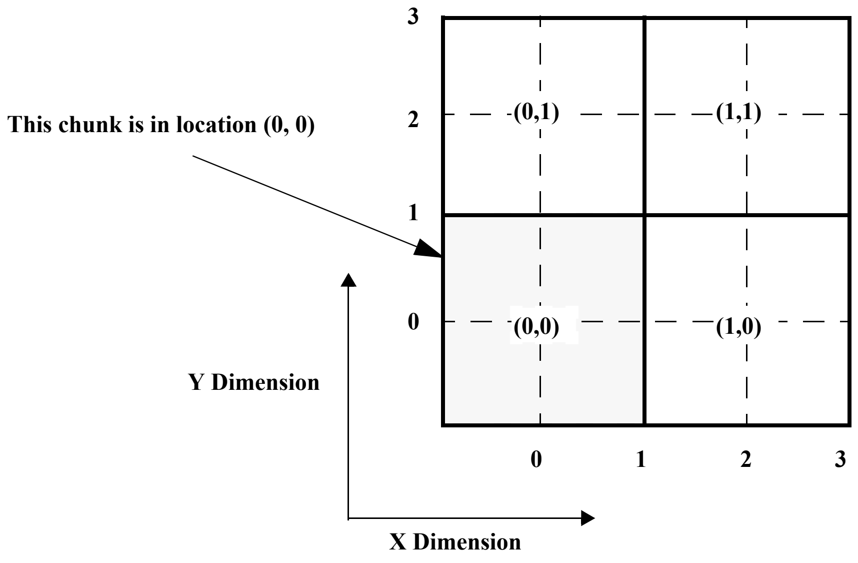
3.12.3. Writing Data to Chunked SDSs: SDwritechunk and SDwritedata

Both **SDwritedata** and **SDwritechunk** can be used to write to a chunked SDS. Later in this chapter, situations where **SDwritechunk** may be a more appropriate routine than **SDwritedata** will be discussed, but, for the most part, both routines achieve the same results. **SDwritedata** is discussed in Section "*Writing Data to an SDS Array: SDwritedata"*. The syntax of **SDwritechunk** is as follows:

```
C:          status = SDwritechunk(sds_id, origin, datap);

FORTRAN:    status = sfwchnk(sds_id, origin, datap)
    OR      status = sfwcchnk(sds_id, origin, datap)
```

The location of data in a chunked SDS can be specified in two ways. The first is the standard method used in the routine **SDwritedata** that access both chunked and non-chunked SDSs; this method refers to the starting location as an offset in elements from the origin of the SDS array itself. The second method is used by the routine **SDwritechunk** that only access chunked SDSs; this method refers to the origin of the chunk as an offset in chunks from the origin of the chunk array itself. The parameter *origin* specifies this offset; it also may be considered as chunk's coordinates in the chunk array. Figure 3d illustrates this method of chunk indexing in a 4-by-4 element SDS array with 2-by-2 element chunks.

FIGURE 3d **Chunk Indexing as an Offset in Chunks**



SDwritechunk is used when an entire chunk is to be written and requires the chunk offset to be known. **SDwritedata** is used when the write operation is to be done regardless of the chunking scheme used in the SDS. Also, as **SDwritechunk** is written specifically for chunked SDSs and does not have the overhead of the additional functionality supported by the **SDwritedata** routine, it is much faster than **SDwritedata**. Note that attempting to use **SDwritechunk** for writing to a non-chunked data set will return a `FAIL` (or `-1`).

The parameter *datap* must point to an array containing the entire chunk of data. In other words, the size of the array must be the same as the chunk size of the SDS to be written to, or an error condition will result.

There are two FORTRAN-77 versions of this routine: **sfwchnk** writes numeric data and **sfwcchnk** writes character data.

SDwritechunk returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDwritechunk** are in Table 3AG. The parameters of **SDwritedata** are listed in Table 3D.

TABLE 3AG **SDwritechunk Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDwritechunk [intn] **(sfwchnk/sfw-cchnk)**	sds_id	int32	integer	Data set identifier
	origin	int32 *	integer	Coordinates of the origin of the chunk to be written
	datap	VOIDP	<valid numeric data type>(*)/character*(*)	Buffer containing the data to be written

3.12.4. Reading Data from Chunked SDSs: SDreadchunk and SDreaddata

As both **SDwritedata** and **SDwritechunk** can be used to write data to a chunked SDS, both **SDreaddata** and **SDreadchunk** can be used to read data from a chunked SDS. **SDreaddata** is discussed in Section "*Reading Data from an SDS Array: SDreaddata*". The syntax of **SDreadchunk** is as follows:

```
C:          status = SDreadchunk(sds_id, origin, datap);

FORTRAN:    status = sfrchnk(sds_id, origin, datap)

    OR      status = sfrcchnk(sds_id, origin, datap)
```

SDreadchunk is used when an entire chunk of data is to be read. **SDreaddata** is used when the read operation is to be done regardless of the chunking scheme used in the SDS. Also, **SDreadchunk** is written specifically for chunked SDSs and does not have the overhead of the additional functionality supported by the **SDreaddata** routine. Therefore, it is much faster than **SDreaddata**. Note that **SDreadchunk** will return `FAIL` (or `-1`) when an attempt is made to read from a non-chunked data set.

As with **SDwritechunk**, the parameter *origin* specifies the coordinates of the chunk to be read, and the parameter *datap* must point to an array containing enough space for an entire chunk of data. In other words, the size of the array must be the same as or greater than the chunk size of the SDS to be read, or an error condition will result.

There are two FORTRAN-77 versions of this routine: **sfrchnk** reads numeric data and **sfrcchnk** reads character data.

SDreadchunk returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **SDreadchunk** are further described in Table 3AH. The parameters of **SDreaddata** are listed in (See Table 3K.

TABLE 3AH **SDreadchunk Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDreadchunk [intn] **(sfrchnk/sfrcchnk)**	sds_id	int32	integer	Data set identifier
	origin	int32 *	integer(*)	Coordinates of the origin of the chunk to be read
	datap	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer for the returned chunk data

3.12.5. Obtaining Information about a Chunked SDS: SDgetchunkinfo

SDgetchunkinfo is used to determine whether an SDS is chunked and how the chunk is defined. The syntax of this routine is as follows:

```
C:          status = SDgetchunkinfo(sds_id, c_def, flag);

FORTRAN:    status = sfgichnk(sds_id, dim_length, flag)
```

Currently, only information about chunk dimensions is retrieved into the corresponding structure element *c_def* for each type of compression in C, and into the array *dim_length* in Fortran. No information on compression parameters is available in the structure *comp* of the union `HDF_CHUNK_DEF`. For specific information on *c_def*, refer to Section "*Making an SDS a Chunked SDS: SDsetchunk"*.

The value returned in the parameter *flag* indicates the data set type (i.e., whether the data set is not chunked, chunked, or chunked and compressed).

If the data set is not chunked, the value of *flag* will be `HDF_NONE` (or `-1`). If the data set is chunked, the value of *flag* will be `HDF_CHUNK` (or `0`). If the data set is chunked and compressed with either RLE, Skipping Huffman, or GZIP compression algorithm, then the value of *flag* will be `HDF_CHUNK | HDF_COMP` (or `1`). If the data set is chunked and compressed with NBIT compression, then the value of *flag* will be `HDF_CHUNK | HDF_NBIT` (or `2`).

If the chunk length for each dimension is not needed, `NULL` can be passed in as the value of the parameter *c_def* in C.

Note that if the data set is empty, **SDgetchunkinfo** will fail. Thus, application must first verify that the data set has been written with data, before calling **SDgetchunkinfo**. **SDcheckempty** in Section "*Determining whether an SDS is empty: SDcheckempty"* determines whether the data set is empty.

SDgetchunkinfo returns either a value of `SUCCEED` (or `0`) or `FAIL` (or `-1`). Refer to Table 3AI and Table 3AJ for the description of the parameters of both versions.

TABLE 3AI

SDgetchunkinfo Parameter List

Routine Name [Return Type]	Parameter	Parameter Type	Description
		C	
SDgetchunkinfo [intn]	sds_id	int32	Data set identifier
	c_def	HDF_CHUNK_DEF *	Union structure containing information about the chunks in the SDS
	flag	int32 *	Flag determining the behavior of the routine

TABLE 3AJ

sfgichnk Parameter List

Routine Name	Parameter	Parameter Type	Description
		FORTRAN-77	
sfgichnk	sds_id	integer	Data set identifier
	dim_length	integer(*)	Sizes of the chunk dimensions
	comp_type	integer	Compression type

EXAMPLE 17.

Writing and Reading a Chunked SDS.

This example demonstrates the use of the routines **SDsetchunk/sfschnk**, **SDwritedata/sfwdata**, **SDwritechunk/sfwchnk**, **SDgetchunkinfo/sfgichnk**, **SDreaddata/sfrdata**, and **SDreadchunk/sfrchnk** to create a chunked data set, write data to it, get information about the data set, and read the data back. Note that the Fortran example uses transpose data to reflect the difference between C and Fortran internal storage.

C:

```
#include "mfhdf.h"

#define FILE_NAME     "SDSchunked.hdf"
#define SDS_NAME      "ChunkedData"
#define RANK          2

main()
{
   /************************ Variable declaration **************************/

   int32         sd_id, sds_id, sds_index;
   intn          status;
   int32         flag, maxcache, new_maxcache;
   int32         dim_sizes[2], origin[2];
   HDF_CHUNK_DEF c_def, c_def_out; /* Chunking definitions */
   int32         comp_flag, c_flags;
   int16         all_data[9][4];
   int32         start[2], edges[2];
   int16         chunk_out[3][2];
   int16         row[2] = { 5, 5 };
   int16         column[3] = { 4, 4, 4 };
   int16         fill_value = 0;   /* Fill value */
   int           i,j;
   /*
   * Declare chunks data type and initialize some of them.
   */
          int16 chunk1[3][2] = { 1, 1,
                                 1, 1,
                                 1, 1 };
```

```
      int16 chunk2[3][2] = { 2, 2,
                             2, 2,
                             2, 2 };

      int16 chunk3[3][2] = { 3, 3,
                             3, 3,
                             3, 3 };

      int16 chunk6[3][2] = { 6, 6,
                             6, 6,
                             6, 6 };

/******************** End of variable declaration **********************/
/*
* Define chunk's dimensions.
*
*         In this example we do not use compression.
*         To use chunking with RLE, Skipping Huffman, and GZIP
*         compression, initialize
*
*                c_def.comp.chunk_lengths[0] = 3;
*                c_def.comp.chunk_lengths[1] = 2;
*
*         To use chunking with NBIT, initialize
*
*                c_def.nbit.chunk_lengths[0] = 3;
*                c_def.nbit.chunk_lengths[1] = 2;
*
*/
c_def.chunk_lengths[0] = 3;
c_def.chunk_lengths[1] = 2;

/*
* Create the file and initialize SD interface.
*/
sd_id = SDstart (FILE_NAME, DFACC_CREATE);

/*
* Create 9x4 SDS.
*/
dim_sizes[0] = 9;
dim_sizes[1] = 4;
sds_id = SDcreate (sd_id, SDS_NAME,DFNT_INT16, RANK, dim_sizes);

/*
* Fill the SDS array with the fill value.
*/
status = SDsetfillvalue (sds_id, (VOIDP)&fill_value);

/*
* Create chunked SDS.
* In this example we do not use compression ( third
* parameter of SDsetchunk is set to HDF_CHUNK).
*
* To use RLE compresssion, set compression type and flag
*
*            c_def.comp.comp_type = COMP_CODE_RLE;
*            comp_flag = HDF_CHUNK | HDF_COMP;
*
* To use Skipping Huffman compression, set compression type, flag
* and skipping size skp_size
*
*            c_def.comp.comp_type = COMP_CODE_SKPHUFF;
*            c_def.comp.cinfo.skphuff.skp_size = value;
*            comp_flag = HDF_CHUNK | HDF_COMP;
*
* To use GZIP compression, set compression type, flag and
* deflate level
*
*            c_def.comp.comp_type = COMP_CODE_DEFLATE;
*            c_def.comp.cinfo.deflate.level = value;
*            comp_flag = HDF_CHUNK | HDF_COMP;
*
* To use NBIT compression, set compression flag and
* compression parameters
*
*            comp_flag = HDF_CHUNK | HDF_NBIT;
*            c_def.nbit.start_bit = value1;
*            c_def.nbit.bit_len   = value2;
*            c_def.nbit.sign_ext  = value3;
*            c_def.nbit.fill_one  = value4;
*/
comp_flag = HDF_CHUNK;
status = SDsetchunk (sds_id, c_def, comp_flag);

/*
* Set chunk cache to hold maximum of 3 chunks.
*/
maxcache = 3;
flag = 0;
new_maxcache = SDsetchunkcache (sds_id, maxcache, flag);

/*
* Write chunks using SDwritechunk function.
* Chunks can be written in any order.
*/

/*
* Write the chunk with the coordinates (0,0).
*/
origin[0] = 0;
origin[1] = 0;
status = SDwritechunk (sds_id, origin, (VOIDP) chunk1);

/*
* Write the chunk with the coordinates (1,0).
*/
origin[0] = 1;
origin[1] = 0;
status = SDwritechunk (sds_id, origin, (VOIDP) chunk3);

/*
* Write the chunk with the coordinates (0,1).
*/
origin[0] = 0;
origin[1] = 1;
status = SDwritechunk (sds_id, origin, (VOIDP) chunk2);

/*
* Write chunk with the coordinates (1,2) using
* SDwritedata function.
*/
start[0] = 6;
start[1] = 2;
edges[0] = 3;
edges[1] = 2;
```

```
status = SDwritedata (sds_id, start, NULL, edges, (VOIDP) chunk6);

/*
* Fill second column in the chunk with the coordinates (1,1)
* using SDwritedata function.
*/
start[0] = 3;
start[1] = 3;
edges[0] = 3;
edges[1] = 1;
status = SDwritedata (sds_id, start, NULL, edges, (VOIDP) column);

/*
* Fill second row in the chunk with the coordinates (0,2)
* using SDwritedata function.
*/
start[0] = 7;
start[1] = 0;
edges[0] = 1;
edges[1] = 2;
status = SDwritedata (sds_id, start, NULL, edges, (VOIDP) row);

/*
* Terminate access to the data set.
*/
status = SDendaccess (sds_id);

/*
* Terminate access to the SD interface and close the file.
*/
status = SDend (sd_id);

/*
* Reopen the file and access the first data set.
*/
sd_id = SDstart (FILE_NAME, DFACC_READ);
sds_index = 0;
sds_id = SDselect (sd_id, sds_index);

/*
* Get information about the SDS. Only chunk lengths and compression
* flag can be returned. Compression information is not available if
* NBIT, Skipping Huffman, or GZIP compression is used.
*/
status = SDgetchunkinfo (sds_id, &c_def_out, &c_flags);
if (c_flags == HDF_CHUNK )
   printf(" SDS is chunked\nChunk’s dimensions %dx%d\n",
          c_def_out.chunk_lengths[0],
          c_def_out.chunk_lengths[1]);
else if (c_flags == (HDF_CHUNK | HDF_COMP))
     printf("SDS is chunked and compressed\nChunk’s dimensions %dx%d\n",
              c_def_out.comp.chunk_lengths[0],
              c_def_out.comp.chunk_lengths[1]);
else if (c_flags == (HDF_CHUNK | HDF_NBIT))
     printf ("SDS is chunked (NBIT)\nChunk’s dimensions %dx%d\n",
              c_def_out.nbit.chunk_lengths[0],
              c_def_out.nbit.chunk_lengths[1]);

/*
* Read the entire data set using SDreaddata function.
*/
start[0] = 0;
start[1] = 0;
```

```
edges[0] = 9;
edges[1] = 4;
status = SDreaddata (sds_id, start, NULL, edges, (VOIDP)all_data);

/*
* Print out what we have read.
* The following information should be displayed:
*
* SDS is chunked
* Chunk's dimensions 3x2
*          1 1 2
*          1 1 2 2
*          1 1 2 2
*          3 3 0 4
*          3 3 0 4
*          3 3 0 4
*          0 0 6 6
*          5 5 6 6
*          0 0 6 6
*/
for (j=0; j<9; j++)
{
     for (i=0; i<4; i++) printf (" %d", all_data[j][i]);
     printf ("\n");
}

/*
* Read chunk with the coordinates (2,0) and display it.
*/
origin[0] = 2;
origin[1] = 0;
status = SDreadchunk (sds_id, origin, chunk_out);
printf (" Chunk (2,0) \n");
for (j=0; j<3; j++)
{
     for (i=0; i<2; i++) printf (" %d", chunk_out[j][i]);
     printf ("\n");
}

/*
* Read chunk with the coordinates (1,1) and display it.
*/
origin[0] = 1;
origin[1] = 1;
status = SDreadchunk (sds_id, origin, chunk_out);
printf (" Chunk (1,1) \n");
for (j=0; j<3; j++)
{
     for (i=0; i<2; i++) printf (" %d", chunk_out[j][i]);
     printf ("\n");
}

/*  The following information is displayed:
*
*   Chunk (2,0)
*   0 0
*   5 5
*   0 0
*   Chunk (1,1)
*   0 4
*   0 4
*   0 4
*/
```

```
    /*
    * Terminate access to the data set.
    */
    status = SDendaccess (sds_id);

    /*
    * Terminate access to the SD interface and close the file.
    */
    status = SDend (sd_id);
}
```

FORTRAN:

```
      program  chunk_examples
      implicit none
C
C     Parameter declaration.
C
      character*14 FILE_NAME
      character*11 SDS_NAME
      integer      RANK
      parameter   (FILE_NAME = 'SDSchunked.hdf',
     +             SDS_NAME  = 'ChunkedData',
     +             RANK      = 2)
      integer      DFACC_CREATE, DFACC_READ, DFNT_INT16
      parameter   (DFACC_CREATE = 4,
     +             DFACC_READ   = 1,
     +             DFNT_INT16   = 22)
      integer      COMP_CODE_NONE
      parameter   (COMP_CODE_NONE = 0)
C
C     This example does not use compression.
C
C     To use RLE compression, declare:
C
C     integer      COMP_CODE_RLE
C     parameter   (COMP_CODE_RLE = 1)
C
C     To use NBIT compression, declare:
C
C     integer      COMP_CODE_NBIT
C     parameter   (COMP_CODE_NBIT = 2)
C
C     To use Skipping Huffman compression, declare:
C
C     integer      COMP_CODE_SKPHUFF
C     parameter   (COMP_CODE_SKPHUFF = 3)
C
C     To use GZIP compression, declare:
C
C     integer      COMP_CODE_DEFLATE
C     parameter   (COMP_CODE_DEFLATE = 4)
C
C
C     Function declaration.
C
      integer sfstart, sfcreate, sfendacc, sfend,
     +        sfselect, sfsfill, sfschnk, sfwchnk,
     +        sfrchnk, sfgichnk, sfwdata, sfrdata,
     +        sfscchnk
C
C**** Variable declaration *******************************************
```

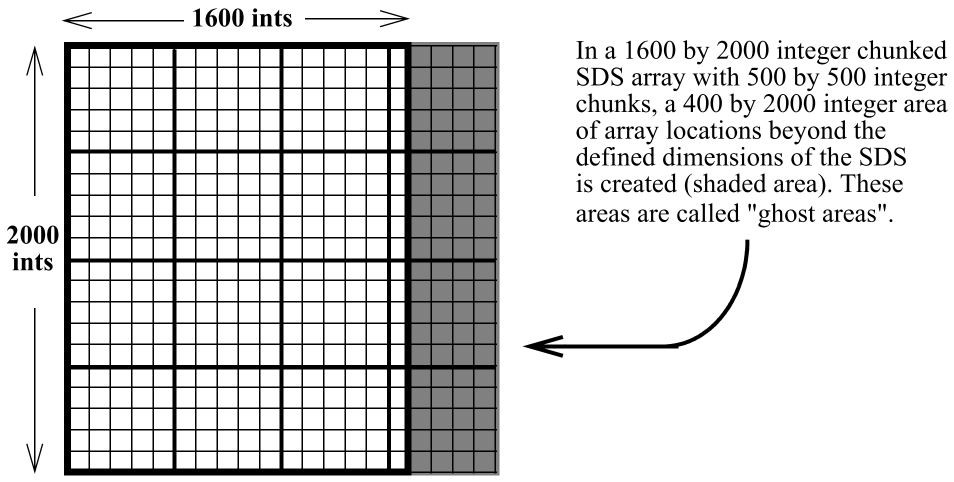
```
C
      integer   sd_id, sds_id, sds_index, status
      integer   dim_sizes(2), origin(2)
      integer   fill_value, maxcache, new_maxcache, flag
      integer   start(2), edges(2), stride(2)
      integer*2 all_data(4,9)
      integer*2 row(3), column(2)
      integer*2 chunk_out(2,3)
      integer*2 chunk1(2,3),
     +          chunk2(2,3),
     +          chunk3(2,3),
     +          chunk6(2,3)
      integer   i, j
C
C     Compression flag and parameters.
C
      integer comp_type, comp_flag, comp_prm(4)
C
C     Chunk’s dimensions.
C
      integer dim_length(2), dim_length_out(2)
C
C     Initialize four chunks
C
      data chunk1 /6*1/
      data chunk2 /6*2/
      data chunk3 /6*3/
      data chunk6 /6*6/
C
C     Initialize row and column arrays.
C
      data row /3*4/
      data column /2*5/
C
C**** End of variable declaration ************************************
C
C
C     Define chunk’s dimensions.
C
      dim_length(1) = 2
      dim_length(2) = 3
C
C     Create the file and initialize SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_CREATE)

C
C     Create 4x9 SDS
C
      dim_sizes(1) = 4
      dim_sizes(2) = 9
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT16,
     +                  RANK, dim_sizes)
C
C     Fill SDS array with the fill value.
C
      fill_value = 0
      status = sfsfill( sds_id, fill_value)
C
C     Create chunked SDS.
C
C     In this example we do not use compression.
C
```

```
C     To use RLE compression, initialize comp_type parameter
C     before the call to sfschnk function.
C              comp_type = COMP_CODE_RLE
C
C     To use NBIT, Skipping Huffman, or GZIP compression,
C     initialize comp_prm array and comp type parameter
C     before call to sfschnk function
C
C     NBIT:
C              comp_prm(1) = value_of(sign_ext)
C              comp_prm(2) = value_of(fill_one)
C              comp_prm(3) = value_of(start_bit)
C              comp_prm(4) = value_of(bit_len)
C              comp_type   = COMP_CODE_NBIT
C
C     Skipping Huffman:
C              comp_prm(1) = value_of(skp_size)
C              comp_type   = COMP_CODE_SKPHUFF
C
C     GZIP:
C              comp_prm(1) = value_of(deflate_level)
C              comp_type   = COMP_CODE_DEFLATE
C
C
      comp_type = COMP_CODE_NONE
      status = sfschnk(sds_id, dim_length, comp_type, comp_prm)
C
C     Set chunk cache to hold maximum 2 chunks.
C
      flag = 0
      maxcache = 2
      new_maxcache = sfscchnk(sds_id, maxcache, flag)
C
C     Write chunks using SDwritechunk function.
C     Chunks can be written in any order.
C
C     Write chunk with the coordinates (1,1).
C
      origin(1) = 1
      origin(2) = 1
      status = sfwchnk(sds_id, origin, chunk1)
C
C     Write chunk with the coordinates (1,2).
C
      origin(1) = 1
      origin(2) = 2
      status = sfwchnk(sds_id, origin, chunk3)
C
C     Write chunk with the coordinates (2,1).
C
      origin(1) = 2
      origin(2) = 1
      status = sfwchnk(sds_id, origin, chunk2)
C
C     Write chunk with the coordinates (2,3).
C
      origin(1) = 2
      origin(2) = 3
      status = sfwchnk(sds_id, origin, chunk6)
C
C     Fill second row in the chunk with the coordinates (2,2).
C
      start(1) = 3
      start(2) = 3
      edges(1) = 1
      edges(2) = 3
      stride(1) = 1
      stride(2) = 1
      status = sfwdata(sds_id, start, stride, edges, row)
C
C     Fill second column in the chunk with the coordinates (1,3).
C
      start(1) = 0
      start(2) = 7
      edges(1) = 2
      edges(2) = 1
      stride(1) = 1
      stride(2) = 1
      status = sfwdata(sds_id, start, stride, edges, column)
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)
C
C     Reopen the file and access the first data set.
C
      sd_id = sfstart(FILE_NAME, DFACC_READ)
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)
C
C     Get information about the SDS.
C
      status = sfgichnk(sds_id, dim_length_out, comp_flag)
      if (comp_flag .eq. 0) then
         write(*,*) 'SDS is chunked'
      endif
      if (comp_flag .eq. 1) then
         write(*,*) 'SDS is chunked and compressed'
      endif
      if (comp_flag .eq. 2) then
         write(*,*) 'SDS is chunked and NBIT compressed'
      endif
      write(*,*) 'Chunks dimensions are ', dim_length_out(1),
     + '  x' ,dim_length_out(2)
C
C     Read the whole SDS using sfrdata function and display
C     what we have read. The following information will be displayed:
C
C
C             SDS is chunked
C             Chunks dimensions are   2  x  3
C
C             1  1  1  3  3  3  0  5  0
C             1  1  1  3  3  3  0  5  0
C             2  2  2  0  0  0  6  6  6
C             2  2  2  4  4  4  6  6  6
C
      start(1) = 0
      start(2) = 0
      edges(1) = 4
      edges(2) = 9
      stride(1) = 1
      stride(2) = 1
      status = sfrdata(sds_id, start, stride, edges, all_data)
C
C     Display the SDS.
C
      write(*,*)
      do 10 i = 1,4
         write(*,*) (all_data(i,j), j=1,9)
10    continue
C
C     Read chunks with the coordinates (2,2) and (1,3) and display.
C     The following information will be shown:
C
C             Chunk (2,2)
C
C               0  0  0
C               4  4  4
C
C             Chunk (1,3)
C
C               0  5  0
C               0  5  0
C
      origin(1) = 2
      origin(2) = 2
      status = sfrchnk(sds_id, origin, chunk_out)
      write(*,*)
      write(*,*) 'Chunk (2,2)'
      write(*,*)
      do 20 i = 1,2
         write(*,*) (chunk_out(i,j), j=1,3)
20    continue
C
      origin(1) = 1
      origin(2) = 3
      status = sfrchnk(sds_id, origin, chunk_out)
      write(*,*)
      write(*,*) 'Chunk (1,3)'
      write(*,*)
      do 30 i = 1,2
         write(*,*) (chunk_out(i,j), j=1,3)
30    continue
C
C     Terminate access to the data set.
C
      status = sfendacc(sds_id)
C
C     Terminate access to the SD interface and close the file.
C
      status = sfend(sd_id)
      end
```

3.13. Ghost Areas

In cases where the size of the SDS array is not an even multiple of the chunk size, regions of excess array space beyond the defined dimensions of the SDS will be created. Refer to the following illustration.

FIGURE 3e **Array Locations Created Beyond the Defined Dimensions of an SDS**


1600 ints
2000 ints
In a 1600 by 2000 integer chunked SDS array with 500 by 500 integer chunks, a 400 by 2000 integer area of array locations beyond the defined dimensions of the SDS is created (shaded area). These areas are called "ghost areas".

These "ghost areas" can be accessed only by **SDreadchunk** and **SDwritechunk**; they cannot be accessed by either **SDreaddata** or **SDwritedata**. Therefore, storing data in these areas is not recommended. Future versions of the HDF library may not include the ability to write to these areas.

If the fill value has been set, the values in these array locations will be initialized to the fill value. It is highly recommended that users set the fill value before writing to chunked SDSs so that garbage values won't be read from these locations.

3.14. netCDF

HDF supports the netCDF data model and interface developed at the Unidata Program Center (UPC). Like HDF, netCDF is an interface to a library of data access programs that store and retrieve data. The file format developed at the UPC to support netCDF uses XDR (eXternal Data Representation), a non-proprietary external data representation developed by Sun Microsystems for describing and encoding data. Full documentation on netCDF and the Unidata netCDF interface is available at `http://www.unidata.ucar.edu/packages/netcdf/`.

The netCDF data model is interchangeable with the SDS data model in so far as it is possible to use the netCDF calling interface to place an SDS into an HDF file and conversely the SDS interface will read from an XDR-based netCDF file. Because the netCDF interface has not changed and netCDF files stored in XDR format are readable, existing netCDF programs and data are still usable, although programs will need to be relinked to the new library. However, there are important conceptual differences between the HDF and the netCDF data model that must be understood to effectively use HDF in working with netCDF data objects and to understand enhancements to the interface that will be included in the future to make the two APIs much more similar.

In the HDF model, when a multidimensional SDS is created by **SDcreate**, HDF data objects are also created that provide information about the individual dimensions — one for each dimension. Each SDS contains within its internal structure the array data as well as pointers to these dimensions. Each dimension is stored in a structure that is in the HDF file but separate from the SDS array.

If more than one SDS have the same dimension sizes, they may share dimensions by pointing to the same dimensions. This can be done in application programs by calling **SDsetdimname** to assign the same dimension name to all dimensions that are shared by several SDSs. For example, suppose you make the following sequence of calls for every SDS in a file:

```
dim_id = SDgetdimid(sds_id, 0);
ret = SDsetdimname(dim_id, "Lat");
dim_id = SDgetdimid(sds_id, 1);
ret = SDsetdimname(dim_id, "Long");
```

This will create a shared dimension named "*Lat*" that is associated with every SDS as the first dimension and a dimension named "*Long*" as the second dimension.

This same result is obtained differently in netCDF. Note that a netCDF "variable" is roughly the same as an HDF SDS. The netCDF interface requires application programs to define all dimensions, using *ncdimdef*, before defining variables. Those defined dimensions are then used to define variables in *ncvardef*. Each dimension is defined by a name and a size. All variables using the same dimension will have the same dimension name and dimension size.

Although the HDF SDS interface will read from and write to *existing* XDR-based netCDF files, HDF cannot be used to *create* XDR-based netCDF files.

There is currently no support for mixing HDF data objects that are not SDSs and netCDF data objects. For example, a raster image can exist in the same HDF file as a netCDF data object, but you must use one of the HDF raster image APIs to read the image and the HDF SD or netCDF interface to read the netCDF data object. The other HDF APIs are currently being modified to allow multifile access. Closer integration with the netCDF interface will probably be delayed until the end of that project.

3.14.1. HDF Interface vs. netCDF Interface

Existing netCDF applications can be used to read HDF files and existing HDF applications can be used to read XDR-based netCDF files. To read an HDF file using a netCDF application, the application must be recompiled using the HDF library. For example, recompiling the netCDF utility *ncdump* with HDF creates a utility that can dump scientific data sets from both HDF and XDR-based files. To read an XDR-based file using an HDF application, the application must be relinked to the HDF library.

The current version of HDF contains several APIs that support essentially the same data model:

- The multifile SD interface.
- The netCDF or NC interface.
- The single-file DFSD interface.
- The multifile GR interface.

The first three models can create, read, and write SDSs in HDF files. Both the SD and NC interfaces can read from and write to XDR-based netCDF files, but they cannot create them. This interoperability means that a single program may contain both SD and NC function calls and thus transparently read and write scientific data sets to HDF or XDR-based files.

The SD interface is the only HDF interface capable of accessing the XDR-based netCDF file format. The DFSD interface cannot access XDR-based files and can only access SDS arrays, dimension scales, and predefined attributes. A summary of file interoperability among the three interfaces is provided in Table 3AK.

TABLE 3AK **Summary of HDF and XDR File Compatibility for the HDF and netCDF APIs**

		Files Created by SD interface	**Files Written by NC Interface**	
		HDF	**HDF Library**	**Unidata netCDF Library**
Accessed by DFSD	Yes	Yes	Yes	No
Accessed by SD	Yes	Yes	Yes	Yes
Accessed by NC	Yes	Yes	Yes	Yes

A summary of NC function calls and their SD equivalents is presented in Table 3AL.

TABLE 3AL **NC Interface Routine Calls and their SD Equivalents**

Purpose	Routine Name		SD Equivalent	Description
	C	FORTRAN-77		
Operations	nccreate	NCCRE	SDstart	Creates a file
	ncopen	NCOPN	SDstart	Opens a file
	ncredef	NCREDF	Not Applicable	Sets open file into define mode
	ncendef	NCENDF	Not Applicable	Leaves define mode
	ncclose	NCCLOS	SDend	Closes an open file
	ncinquire	NCINQ	SDfileinfo	Inquires about an open file
	ncsync	NCSNC	Not Applicable	Synchronizes a file to disk
	ncabort	NCABOR	Not Applicable	Backs out of recent definitions
	ncsetfill	NCSFIL	Not Implemented	Sets fill mode for writes
Dimensions	ncdimdef	NCDDEF	SDsetdimname	Creates a dimension
	ncdimid	NCDID	SDgetdimid	Returns a dimension identifier from its name
	ncdiminq	NCDINQ	SDdiminfo	Inquires about a dimension
	ncdimrename	NCDREN	Not Implemented	Renames a dimension
Variables	ncvardef	NCVDEF	SDcreate	Creates a variable
	ncvarid	NCVID	SDnametoindex and SDselect	Returns a variable identifier from its name
	ncvarinq	NCVINQ	SDgetinfo	Returns information about a variable
	ncvarput1	NCVPT1	Not Implemented	Writes a single data value
	ncvarget1	NCVGT1	Not Implemented	Reads a single data value
	ncvarput	NCVPT	SDwritedata	Writes a hyperslab of values
	ncvarget	NCVGT/ NCVGTC	SDreaddata	Reads a hyperslab of values
	ncvarrename	NCVREN	Not Implemented	Renames a variable
	nctypelen	NCTLEN	DFKNTsize	Returns the number of bytes for a number type
Attributes	ncattput	NCAPT/ NCAPTC	SDsetattr	Creates an attribute
	ncattinq	NCAINQ	SDattrinfo	Returns information about an attribute
	ncattcopy	NCACPY	Not Implemented	Copies attribute from one file to another
	ncattget	NCAGT/ NCAGTC	SDreadattr	Returns attributes values
	ncattname	NCANAM	SDattrinfo	Returns name of attribute from its number
	ncattrename	NCAREN	Not Implemented	Renames an attribute
	ncattdel	NCADEL	Not Implemented	Deletes an attribute

3.14.2. ncdump and ncgen

The **ncdump** summary capability works on both HDF and netCDF files.

The **ncgen** summary capability works only on netCDF files.

3.14.2.1. Using ncdump on HDF Files

When used with an HDF file on some platforms (reported on SGI), **ncdump** may display signed 8-bit integer data (**int8**, with the intended signed range of *-127* through *128*) as unsigned 8-bit integer data (**uint8**, with the unsigned range *0* through *255*). This is due to the mapping of **int8** and **uint8** types in HDF to a common type, **NC_BYTE**, in netCDF.

3.14.2.2. New error code from ncdump

Prior to 4.2.11, ncdump did not report failure in reading corrupted data even though the internal reading function failed, thus, ncdump appeared to succeed when data corruption exists. Starting in version 4.2.11, when corrupted data is encountered, ncdump will display the following message for the variable with corrupted data and proceed to the next variable or exit if there are no more variables to read:

```
        "Reading failed for variable <Variable name>, the data is possibly corrupted."
```

CHAPTER 4 -- Vdatas (VS API)

4.1. Chapter Overview

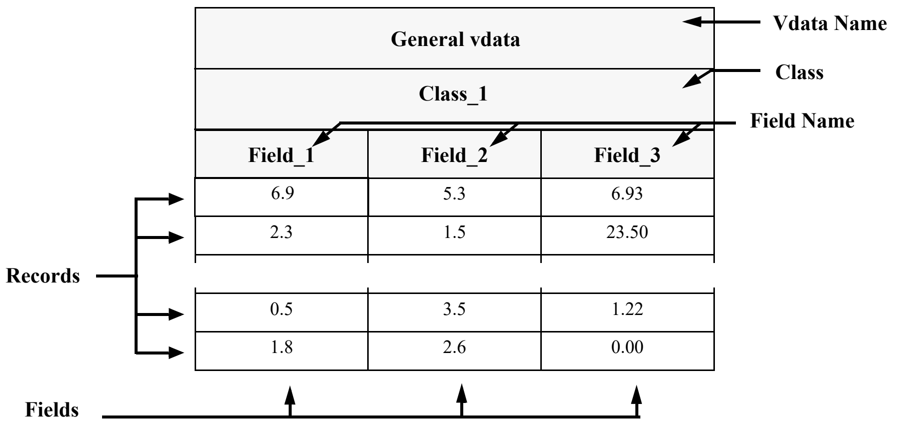
This chapter describes the vdata data model, the Vdata interface (also called the VS interface or the VS API), and the vdata programming model.

4.2. The Vdata Model

The HDF ***Vdata model*** provides a framework for storing customized tables, or ***vdatas***, in HDF files. The term "vdata" is an abbreviation of "vertex data", alluding to the fact that the object was first implemented in HDF to store the vertex and edge information of polygon sets. The vdata design has since been generalized to apply to a broader variety of applications.

A vdata is like a table that consists of a collection of ***records*** whose values are stored in fixed-length ***fields***. All records have the same structure and all values in each field have the same data type. (See Figure 4a) The library does not check for uniqueness in vdata's name, class, or field names. For example, when two vdatas having the same name, the first vdata will always be returned by VSfind().

FIGURE 4a **Vdata Table Structure**



A ***vdata name*** is a label typically assigned to describe the contents of a vdata. It often serves as a search key to locate a vdata in a file. A ***vdata class*** further distinguishes a particular vdata by identifying the purpose or the use of its data. Finally, ***vdata field names*** are labels assigned to the fields in the vdata.

4.2.1. Records and Fields

Each ***record*** in a vdata is composed of one or more fixed-length ***fields***. Vdata records and fields are identified by an index. The record and field indexes are zero-based and are separately incremented by one for each additional record and field in the vdata.

Every field in a vdata is assigned a data type when the vdata is created. The data type of a field may be any basic HDF data type: character, 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating point numbers. The maximum length of a vdata record is 65,535 bytes.

The Vdata model allows multiple entries per field as long as they have the same data type. The number of entries or ***components*** in a field is called the ***order*** of the field.

The organizational structure of a vdata is often determined by the data types of its data set or sets. For example, given a data set describing the location (“X,Y”) and temperature (“Temp”) of points in a plane, there are several ways to organize the data. (See Figure 4b) If the “X”, “Y” and “Temp” values are of the same data type, they could be stored as three single-component fields, as a two-component “X_Y” field and a single-component “Temp” field, or as a three-component “X_Y_-Temp” field. Generally, the “X,Y” data is stored in a single field, but HDF places no restrictions on the organization of field data and there are no significant HDF performance issues involved in choosing one organizational regime over another.

FIGURE 4b **Three Different Vdata Structures for Data of the Same Number Type**

Simulation Data 1		
2D_Temperature_Grid		
X	**Y**	**Temp**
2.30	1.50	23.50
3.40	5.70	8.03
0.50	3.50	1.22
1.80	2.60	0.00

3 Single-component Fields

Simulation Data 1	
2D_Temperature_Grid	
X_Y	**Temp**
2.30, 1.50	23.50
3.40, 5.70	8.03
0.50, 3.50	1.22
1.80, 2.60	0.00

1 Multi-component Field of Order 2
1 Single-component Field

Simulation Data 1
2D_Temperature_Grid
X_Y_Temp
2.30, 1.50, 23.50
3.40, 5.70, 8.03
0.50, 3.50, 1.22
1.80, 2.60, 0.00

1 Multi-component Field of Order 3

4.3. The Vdata Interface

The Vdata interface consists of routines that are used to store and retrieve information about vdatas and their contents.

4.3.1. Header Files Used by the Vdata Interface

The header file “hdf.h” must be included in programs that invoke Vdata interface routines.

4.3.2. Vdata Library Routines

Vdata routines begin with the prefixes “VS”, “VF”, “VSQ”, and “VH” in C, and “vsf”, “vf”, “vsq”, and “vh” in FORTRAN-77. Vdata routines perform most general vdata operations, VF routines query information about vdata fields, and VSQ routines query information about specific vdatas. VH routines are high-level procedures that write to single-field vdatas.

Vdata routines let you define, organize and manipulate vdatas. They are categorized as follows and are listed in Table 4A by their categories:

- ***Access routines*** control access to files and vdatas. Data transfer to and from a vdata can only occur after the access to the vdata has been initiated and before it is terminated. Some Vgroup interface routines are included since they are used interchangeably between the Vdata and Vgroup interfaces. Refer to Chapter 5, *Vgroups (V API)*, for a description of the Vgroup interface.
- ***Read and write routines*** store and retrieve the contents of and the information about a vdata.
- ***File inquiry routines*** provide information about how vdatas are stored in a file. They are useful for locating vdatas in the file.
- ***Vdata inquiry routines*** provide specific information about a given vdata, including the vdata's name, class, number of records, tag and reference number pairs, size, and interlace mode.
- ***Field inquiry routines*** provide specific information about the fields in a given vdata, including the field's size, name, order, and type, and the number of fields in the vdata.

TABLE 4A

Vdata Interface Routines

Category	Routine Names		Description
	C	FORTRAN-77	
Access/Create	`Vstart`	`vfstart`	Initializes the Vdata and the Vgroup interfaces (Section "*Accessing Files and Vdatas: Vstart and VSattach"*)
	`VSattach`	`vsfatch`	Establishes access to a specified vdata (Section "*Accessing Files and Vdatas: Vstart and VSattach"*)
	`VSdetach`	`vsfdtch`	Terminates access to a specified vdata (Section "*Terminating Access to Vdatas and Files: VSdetach and Vend"*)
	`Vend`	`vfend`	Terminates access to the Vdata and the Vgroup interfaces (Section "*Terminating Access to Vdatas and Files: VSdetach and Vend"*)
Read and Write	`VSfdefine`	`vsffdef`	Defines a new vdata field (Section "*Defining a Field within a Vdata: VSfdefine"*)
	`VSread`	`vsfrd/vsfrdc/ vsfread`	Reads one record from a vdata (Section "*Reading from the Current Vdata: VSread"*)
	`VSseek`	`vsfseek`	Seeks to a specified record in a vdata (Section "*Resetting the Current Position within Vdatas: VSseek"*)
	`VSsetattr`	`vsfsnat/vsfs-cat`	Sets the attribute of a vdata field or vdata (Section "*Setting the Attribute of a Vdata or Vdata Field: VSsetattr"*)
	`VSsetclass`	`vsfscls`	Assigns a class to a vdata (Section "*Assigning a Vdata Name and Class: VSsetname and VSsetclass"*)
	`VSsetfields`	`vsfsfld`	Specifies the vdata fields to be read or written (Section "*Initializing the Fields for Write Access: VSsetfields"* and Section "*Initializing the Fields for Read Access: VSsetfields"*)
	`VSsetinterlace`	`vsfsint`	Sets the interlace mode for a vdata (Section "*Specifying the Interlace Mode: VSsetinterlace"*)
	`VSsetname`	`vsfsnam`	Assigns a name to a vdata (Section "*Assigning a Vdata Name and Class: VSsetname and VSsetclass"*)
	`VHstoredata`	`vhfsd/vhfscd`	Writes data to a vdata with a single-component field (Section "*Creating and Writing to Single-Field Vdatas: VHstoredata and VHstoredatam"*)
	`VHstoredatam`	`vhfsdm/ vhfscdm`	Writes data to a vdata with a multi-component field (Section "*Creating and Writing to Single-Field Vdatas: VHstoredata and VHstoredatam"*)
	`VSgetblockinfo`	`vsfgetblinfo`	Retrieves the block size and the number of blocks for a linked-block vdata element (see *HDF Reference Manual*)
	`VSsetblocksize`	`vsfsetblsz`	Sets linked-block vdata element block size (see *HDF Reference Manual*)
	`VSsetnumblocks`	`vsfsetnmbl`	Sets the number of blocks for a linked-block vdata element (see *HDF Reference Manual*)
	`VSwrite`	`vsfwrt/vsf-wrtc/ vsfwrit`	Writes records to a vdata (Section "*Writing to a Vdata: VSwrite"*)

Vdata Inquiry	`VSattrinfo`	`vsfainf`	Retrieves information on a given attribute (Section "*Querying Information on a Vdata or Vdata Field Attribute: VSattrinfo"*)
	`VSelts`	`vsfelts`	Returns the number of records in the specified vdata (Section "*Other Vdata Information Retrieval Routines"*)
	`VSfexist`	`vsfex`	Locates a vdata given a list of field names (Section "*Searching for a Vdata by Field Name: VSfexist"*)
	`VSfindex`	`vsffidx`	Returns the index of a vdata field given the field name (Section "*Querying the Index of a Vdata Field Given the Field Name: VSfindex"*)
	`VSfnattrs`	`vsffnas`	Returns the number of attributes of a vdata or vdata field (Section "*Querying the Number of Attributes of a Vdata or a Vdata Field: VSfnattrs"*)
	`VSfindattr`	`vsffdat`	Retrieves the index of an attribute given the attribute name (Section "*Retrieving the Index of a Vdata or Vdata Field Attribute Given the Attribute Name: VSfindattr"*)
	`VSgetattr`	`vsfgnat/vsfg-cat`	Retrieves the values of a given attribute (Section "*Querying the Values of a Vdata or Vdata Field Attribute: VSgetattr"*)
	`VSgetclass`	`vsfgcls`	Returns the class name of the specified vdata (Section "*Other Vdata Information Retrieval Routines"*)
	`VSgetfields`	`vsfgfld`	Retrieves all field names within the specified vdata (Section "*Other Vdata Information Retrieval Routines"*)
	`VSgetinterlace`	`vsfgint`	Retrieves the interlace mode of the specified vdata (Section "*Other Vdata Information Retrieval Routines"*)
	`VSgetname`	`vsfgnam`	Retrieves the name of the specified vdata (Section "*Other Vdata Information Retrieval Routines"*)
	`VSinquire`	`vsfinq`	Returns information about the specified vdata (Section "*Obtaining Vdata Information: VSinquire"*)
	`VSisattr`	`vsfisat`	Determines whether the given vdata is an attribute (Section "*Determining whether a Vdata Is an Attribute: VSisattr"*)
	`VSnattrs`	`vsfnats`	Returns the total number of vdata attributes (Section "*Querying the Total Number of Vdata and Vdata Field Attributes: VSnattrs"*)
	`VSQuerycount`	`vsqfnelt`	Returns the number of records in the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQueryfields`	`vsqfflds`	Returns the field names of the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQueryinter-lace`	`vsqfintr`	Returns the interlace mode of the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQueryname`	`vsqfname`	Returns the name of the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQueryref`	`vsqref`	Retrieves the reference number of the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQuerytag`	`vsqtag`	Retrieves the tag of the specified vdata (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSQueryvsize`	`vsqfsiz`	Retrieves the local size in bytes of the specified vdata record (Section "*VSQuery Vdata Information Retrieval Routines"*)
	`VSsetattr`	`vsfsnat/vsfs-cat`	Sets the attribute of a vdata field or vdata (Section "*Setting the Attribute of a Vdata or Vdata Field: VSsetattr"*)
	`VSsizeof`	`vsfsiz`	Returns the size of the specified fields in a vdata (Section "*Other Vdata Information Retrieval Routines"*)

Field Inquiry	`VFfieldesize`	`vffesiz`	Returns the field size, as stored in a file, of a specified field (Section "*VF Field Information Retrieval Routines*")
	`VFfieldisize`	`vffisiz`	Returns the field size, as stored in memory, of a specified field (Section "*VF Field Information Retrieval Routines*")
	`VFfieldname`	`vffname`	Returns the name of the specified field in the given vdata (Section "*VF Field Information Retrieval Routines*")
	`VFfieldorder`	`vffordr`	Returns the order of the specified field in the given vdata (Section "*VF Field Information Retrieval Routines*")
	`VFfieldtype`	`vfftype`	Returns the data type for the specified field in the given vdata (Section "*VF Field Information Retrieval Routines*")
	`VFnfields`	`vfnflds`	Returns the total number of fields in the specified vdata (Section "*VF Field Information Retrieval Routines*")
File Inquiry	`VSfind`	`vsffnd`	Searches for a vdata in a file given the vdata's name (Section "*Determining a Reference Number from a Vdata Name: VSfind*")
	`VSgetid`	`vsfgid`	Returns the reference number of the next vdata in the file (Section "*Sequentially Searching for a Vdata: VSgetid*")
	`VSlone`	`vsflone`	Returns the reference number of vdatas that are not linked with any vgroups (Section "*Finding All Vdatas that are Not Members of a Vgroup: VSlone*")

4.3.3. Identifying Vdatas in the Vdata Interface

The Vdata interface identifies vdatas in several ways. Before an existing vdata is accessible, it is uniquely identified by its ***reference number***. The reference number of a vdata can be obtained from the name or the class of the vdata, or by sequentially traversing the file. The concept of reference number is discussed in Section "*Data Descriptor*".

When a vdata is attached, it is assigned with an identifier, called ***vdata id***, which is used by the Vdata interface routines in accessing the vdata.

4.3.4. Programming Model for the Vdata Interface

The programming model for accessing vdatas is as follows:

1. Open the file.
2. Initialize the Vdata interface.
3. Create a new vdata or open an existing one using its reference number.
4. Perform the desired operations on the vdata.
5. Terminate access to the vdata.
6. Terminate access to the Vdata interface.
7. Close the file.

To access a vdata, the calling program must contain the following calls, which are individually explained in the following subsections:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);
            vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
            <Optional operations>
            status = VSdetach(vdata_id);
            status = Vend(file_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            status = vfstart(file_id)
```

```
vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
<Optional operations>
status = vsfdtch(vdata_id)
status = vfend(file_id)
status = hclose(file_id)
```

4.3.5. Accessing Files and Vdatas: Vstart and VSattach

An HDF file must be opened by **Hopen** before it can be accessed using the Vdata interface. **Hopen** is described in Chapter 2, *HDF Fundamentals*.

Vstart must be called for every file to be accessed. This routine initializes the internal vdata structures used by the Vdata interface. **Vstart** has only one argument, the file identifier (*file_id*) returned by **Hopen**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). Note that the **Vstart** routine is used by both the Vdata and Vgroup interfaces.

VSattach initiates access to a vdata and must be called before any operations on the vdata may occur. **VSattach** takes three arguments: *file_id*, *vdata_ref*, and *vdata_access_mode*, and returns either a vdata identifier or `FAIL` (or `-1`).

The argument *file_id* is the file identifier returned by **Hopen** and *vdata_ref* is the reference number that identifies the vdata to be accessed. Specifying *vdata_ref* with a value of `-1` will create a new vdata; specifying *vdata_ref* with a nonexistent reference number will return an error code of `FAIL` (or `-1`); and specifying *vdata_ref* with a valid reference number will initiate access to the corresponding vdata.

If an existing vdata's reference number is unknown, it must be obtained prior to the **VSattach** call. (Refer to Chapter 2, *HDF Fundamentals*, for a description of reference numbers.) The HDF library provides two routines for this purpose, **VSfind** and **VSgetid**. **VSfind** can be used to obtain the reference number of a vdata when the vdata's name is known. **VSgetid** can be used to obtain the reference number when only the location of the vdata within the file is known; this is often discovered by sequentially traversing the file. These routines are discussed in Section "*Sequentially Searching for a Vdata: VSgetid*" and Section "*Determining a Reference Number from a Vdata Name: VSfind*".

The argument *vdata_access_mode* specifies the access mode ("*r*" for read-only access or "*w*" for read and write access) for subsequent operations on the specified vdata. Although several HDF user programs may simultaneously read from one vdata, only one write access is allowed at a time. The "*r*" access mode may only be used with existing vdatas; the "*w*" access mode is valid with both new vdatas (*vdata_ref = -1*) *and* existing vdatas.

Note that, although a vdata can be created without being written with data, either the routine **VSsetname** or **VSsetfields** must be called in order for the vdata to exist in the file.

The parameters for **Vstart** and **VSattach** are further defined in Table 4B.

4.3.6. Terminating Access to Vdatas and Files: VSdetach and Vend

VSdetach terminates access to a vdata by updating pertinent information and freeing all memory associated with the vdata and initialized by **VSattach**. Once access to the vdata is terminated, its identifier becomes invalid and any attempt to access it will result in an error condition. **VSdetach** takes only one argument, the vdata identifier that is returned by **VSattach**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`).

Vend releases all internal data structures allocated by **Vstart**. **Vend** must be called once for each call to **Vstart** and only after access to all vdatas have been terminated (i.e., all calls to **VSdetach** have been made). Attempts to call Vdata interface routines after calling **Vend** will result in an error condition. **Vend** takes one argument, the file identifier that is returned by **Hopen**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). Note that the **Vend** routine is used by both the Vdata and Vgroup interfaces.

In summary, successfully terminating access to a vdata requires one **VSdetach** call for each call to **VSattach** and one **Vend** call for each call to **Vstart**.

The parameters for **VSdetach** and **Vend** are further defined in Table 4B.

Hclose terminates access to a file and should only be called after all **Vend** calls have been made to close the Vdata interface. Refer to Chapter 2, *HDF Fundamentals*, for a description of **Hclose**.

TABLE 4B

Vstart, VSattach, VSdetach, and Vend Parameter Lists

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Vstart [intn] **(vfstart)**	file_id	int32	integer	File identifier
VSattach [int32] **(vsfatch)**	file_id	int32	integer	File identifier
	vdata_ref	int32	integer	Reference number of the vdata
	vdata_access_mode	char *	character*1	Vdata access mode
VSdetach [int32] **(vsfdtch)**	vdata_id	int32	integer	Vdata identifier
Vend [intn] **(vfend)**	file_id	int32	integer	File identifier

EXAMPLE 1.

Accessing a Vdata in an HDF File

This example illustrates the use of **Hopen/hopen**, **Vstart/vfstart**, **VSattach/vsfatch**, **VSdetach/vsfdtch**, **Vend/vfend**, and **Hclose/hclose** to create and to access different vdatas from different HDF files.

The program creates an HDF file, named "General_Vdatas.hdf", containing a vdata. The program also creates a second HDF file, named "Two_Vdatas.hdf", containing two vdatas. Note that, in this example, the program does not write data to these vdatas. Also note that before closing the file, the access to its vdatas and its corresponding Vdata interface must be terminated. These examples request information about a specific vdata.

C:

```
#include "hdf.h"
```

```
#define  FILE1_NAME     "General_Vdatas.hdf"
#define  FILE2_NAME     "Two_Vdatas.hdf"
#define  VDATA_NAME      "Vdata 1"
#define  VDATA_CLASS    "Empty Vdatas"
```

```
main( )
{
  /************************* Variable declaration **************************/

  intn  status_n;      /* returned status for functions returning an intn  */
  int32 status_32,     /* returned status for functions returning an int32 */
        file1_id, file2_id,
        vdata_id, vdata1_id, vdata2_id,
        vdata_ref = -1;     /* ref number of a vdata, set to -1 to create  */

  /********************* End of variable declaration *********************/

  /*
  * Create the first HDF file.
  */
  file1_id = Hopen (FILE1_NAME, DFACC_CREATE, 0);

  /*
  * Initialize the VS interface associated with the first HDF file.
  */
  status_n = Vstart (file1_id);

  /*
  * Create a vdata in the first HDF file.
  */
  vdata_id = VSattach (file1_id, vdata_ref, "w");

  /*
  * Assign a name to the vdata.
  */
  status_32 = VSsetname (vdata_id, VDATA_NAME);

  /*
  * Other operations on the vdata identified by vdata_id can be carried
  * out starting from this point.
```

```
*/

/*
* Create the second HDF file.
*/
file2_id = Hopen (FILE2_NAME, DFACC_CREATE, 0);

/*
* Initialize the VS interface associated with the second HDF file.
*/
status_n = Vstart (file2_id);

/*
* Create the first vdata in the second HDF file.
*/
vdata1_id = VSattach (file2_id, vdata_ref, "w");

/*
* Create the second vdata in the second HDF file.
*/
vdata2_id = VSattach (file2_id, vdata_ref, "w");

/*
* Assign a class name to these vdatas.
*/
status_32 = VSsetclass (vdata1_id, VDATA_CLASS);
status_32 = VSsetclass (vdata2_id, VDATA_CLASS);

/*
* Other operations on the vdatas identified by vdata1_id and vdata2_id
* can be carried out starting from this point.
*/

/*
* Terminate access to the first vdata in the second HDF file.
```

```
*/
status_32 = VSdetach (vdata1_id);

/*
* Terminate access to the second vdata in the second HDF file.
*/
status_32 = VSdetach (vdata2_id);

/*
* From this point on, any operations on the vdatas identified by vdata1_id
and vdata2_id are invalid but not on the vdata identified by vdata_id.
*/

/*
* Terminate access to the VS interface associated with the second HDF file.
*/
status_n = Vend (file2_id);

/*
* Close the second HDF file.
*/
status_n = Hclose (file2_id);

/*
* Terminate access to the vdata in the first HDF file.
*/
status_32 = VSdetach (vdata_id);

/*
* Terminate access to the VS interface associated with the first HDF file.
*/
status_n = Vend (file1_id);

/*
* Close the first HDF file.
```

```
  */

  status_n = Hclose (file1_id);

}
```

FORTRAN:

```
       program create_vdatas
           implicit none
      C
      C     Parameter declaration
      C
           character*18 FILE1_NAME
           character*14 FILE2_NAME
           character*7  VDATA_NAME
           character*12 VDATA_CLASS
      C
           parameter (FILE1_NAME  = 'General_Vdatas.hdf',
          +           FILE2_NAME  = 'Two_Vdatas.hdf',
          +           VDATA_NAME  = 'Vdata 1',
          +           VDATA_CLASS = 'Empty Vdatas')
           integer DFACC_CREATE
           parameter (DFACC_CREATE = 4)
      C
      C     Function declaration
      C
           integer hopen, hclose
           integer vfstart, vsfatch, vsfsnam, vsfscls, vsfdtch, vfend

      C
      C**** Variable declaration *******************************************
      C
           integer status
           integer file1_id, file2_id
           integer vdata_id, vdata1_id, vdata2_id
           integer vdata_ref
      C
      C**** End of variable declaration ************************************
      C
      C
      C     Create the first HDF file.
      C
           file1_id = hopen(FILE1_NAME, DFACC_CREATE, 0)
      C
      C     Initialize the VS interface associated with the first HDF file.
      C
           status = vfstart(file1_id)
      C
      C     Create a vdata in the first HDF file.
      C
           vdata_ref = -1
           vdata_id = vsfatch(file1_id, vdata_ref, 'w')
      C
      C     Assign a name to the vdata.
      C
           status = vsfsnam(vdata_id, VDATA_NAME)
      C
      C     Other operations on the vdata identified by vdata_id can be carried out
      C     starting from this point.
      C
      C     Create the second HDF file.
```

```
C
      file2_id = hopen(FILE2_NAME, DFACC_CREATE, 0)
C
C     Initialize the VS interface associated with the second HDF file.
C
      status = vfstart(file2_id)
C
C     Create the first vdata in the second HDF file.
C
      vdata1_id = vsfatch(file2_id, vdata_ref, 'w')
C
C     Create the second vdata in the second HDF file.
C
      vdata2_id = vsfatch(file2_id, vdata_ref, 'w')
C
C     Assign a class name to these vdatas.
C
      status = vsfscls(vdata1_id, VDATA_CLASS)
      status = vsfscls(vdata2_id, VDATA_CLASS)
C
C     Other operations on the vdatas identified by vdata1_id and vdata2_id
C     can be carried out starting from this point.
C
C
C     Terminate access to the first vdata in the second HDF file.
C
      status = vsfdtch(vdata1_id)
C
C     Terminate access to the second vdata in the second HDF file.
C
      status = vsfdtch(vdata2_id)
C
C     Terminate access to the VS interface associated with the second HDF file.
C
      status = vfend(file2_id)
C
C     Close the second HDF file.
C
      status = hclose(file2_id)
C
C     Terminate access to the vdata in the first HDF file.
C
      status = vsfdtch(vdata_id)
C
C     terminate access to the VS interface associated with the first HDF file.
C
      status = vfend(file1_id)
C
C     Close the first HDF file.
C
      status = hclose(file1_id)
      end
```

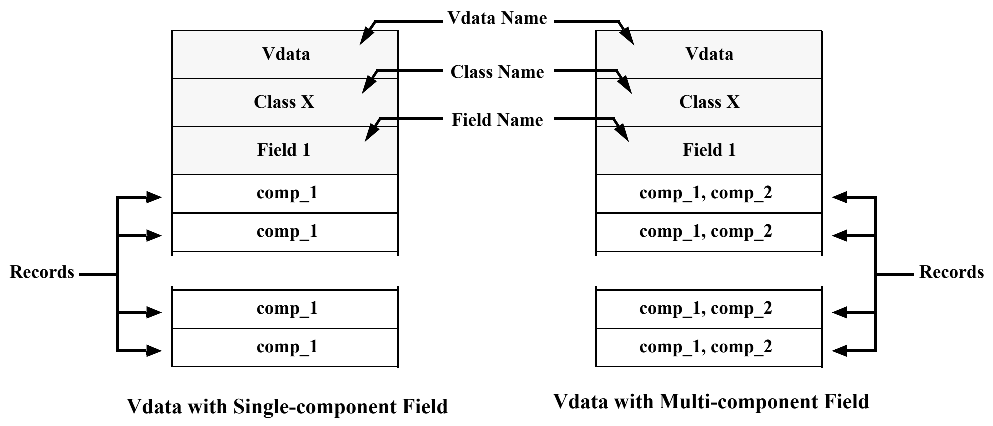
4.4. Creating and Writing to Single-Field Vdatas: VHstoredata and VHstoredatam

There are two methods of writing vdatas that contain one field per record. One requires the use of several VS routines and the other involves the use of **VHstoredata** or **VHstoredatam**, two high-level routines that encapsulate several VS routines into one.

The high-level VH routines are useful when writing one-field vdatas and complete information about each vdata is available. If you cannot provide full information about a vdata, you must use the VS routines described in the next section.

Figure 4c shows two examples of single-field vdatas. The fields can be single-component or multi-component fields. With a multi-component field, they may contain one or more values of the same data type.

FIGURE 4c **Single- and Multi-component Vdatas**



VHstoredata creates then writes a vdata with one single-component field. **VHstoredatam** creates and writes a vdata with one multi-component field. In both cases the following steps are involved:

1. Open the file.
2. Initialize the Vdata interface.
3. Store (create then write to) the vdata.
4. Terminate access to the Vdata interface.
5. Close the file.

These steps correspond to the following sequence of function calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);

            /* Either VHstoredata or VHstoredatam can be called here. */
            vdata_ref = VHstoredata(file_id, fieldname, buf, n_records, data_-
                             type, vdata_name, vdata_class);
    OR      vdata_ref = VHstoredatam(file_id, fieldname, buf, n_records, data_-
                             type, vdata_name, vdata_class, order);

            status = Vend(file_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            status = vfstart(file_id)

C           Either vhfsd/vhfscd or vhfsdm/vhfscdm can be called here.
            vdata_ref = vhfsd(file_id, fieldname, buf, n_records, data_type,
                             vdata_name, vdata_class)
    OR      vdata_ref = vhfscd(file_id, fieldname, buf, n_records, data_type,
                             vdata_name, vdata_class)
```

```
     OR

     vdata_ref = vhfsdm(file_id, fieldname, buf, n_records, data_type,
                        vdata_name, vdata_class, order)
OR   vdata_ref = vhfscdm(file_id, fieldname, buf, n_records, data_type,
                        vdata_name, vdata_class, order)

     status = vfend(file_id)
     status = hclose(file_id)
```

The first seven parameters of **VHstoredata** and **VHstoredatam** are the same. The parameter *file_id* is the file identifier returned by **Hopen**. The parameter *fieldname* specifies the name of the vdata field. The parameter *buf* contains the data to be stored into the vdata. In C, the data type of the parameter *buf* is *uint8*; in FORTRAN-77, it is the data type of the data to be stored. The parameters *n_records* and *data_type* contain the number of records in the vdata and the data type of the vdata data. The parameters *vdata_name* and *vdata_class* specify the name and class of the vdata. The parameter *order* of **VHstoredatam** specifies the order of the field. The maximum length of the vdata name is given by the `VSNAMELENMAX` (or `64`) as defined in the header file "hlimits.h".

Note that these two routines do not overwrite existing vdatas but only create new ones before storing the data.

The FORTRAN-77 version of **VHstoredata** has two routines: **vhfsd** for numeric data and **vhfscd** for character data; the FORTRAN-77 version of **VHstoredatam** has two routines: **vhfsdm** for numeric data and **vhfscdm** for character data.

Both routines return the reference number of the newly-created vdata or `FAIL` (or `-1`) if the operation is unsuccessful. The parameters for **VHstoredata** and **VHstoredatam** are further described in Table 4C.

TABLE 4C

VHstoredata and VHstoredatam Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VHstoredata [int32] **(vhfsd/vhfscd)**	file_id	int32	integer	File identifier
	fieldname	char *	character*(*)	String containing the name of the field
	buf	uint8 *	<valid numeric data type>(*)/character*(*)	Buffer containing the data to be stored
	n_records	int32	integer	Number of records to create in the vdata
	data_type	int32	integer	Data type of the stored data
	vdata_name	char *	character*(*)	Name of the vdata
	vdata_class	char *	character*(*)	Class name of the vdata
VHstoredatam [int32] **(vhfsdm/ vhfscdm)**	file_id	int32	integer	File identifier
	fieldname	char *	character*(*)	String containing the name of the field
	buf	uint8 *	<valid numeric data type>(*)/character*(*)	Buffer containing the data to be stored
	n_records	int32	integer	Number of records to create in the vdata
	data_type	int32	integer	Data type of the stored data
	vdata_name	char *	character*(*)	Name of the vdata
	vdata_class	char *	character*(*)	Class name of the vdata
	order	int32	integer	Number of field components

EXAMPLE 2.

Creating and Storing One-field Vdatas Using VHstoredata and VHstoredatam

This example illustrates the use of **VHstoredata/vhfscd** and **VHstoredatam/vhfsdm** to create single-field vdatas.

This example creates and writes two vdatas to the file "General_Vdatas.hdf". The first vdata is named "First Vdata", contains 5 records, and belongs to a class named "5x1 Array". The second vdata is named "Second Vdata", contains 6 records, and belongs to a class named "6x4 Array". The field of the first vdata is a single-component field, i.e., order of 1, and named "Single-component Field". The field of the second vdata has an order of 4 and is named "Multi-component Field".

In these examples two vdatas are created. The first vdata has five records with one field of order 1 and is created from a 5 x 1 array in memory. The second vdata has six records with one field of order 4 and is created from a 6 x 4 array in memory.

C:

```
#include "hdf.h"

#define  FILE_NAME      "General_Vdatas.hdf"
#define  CLASS1_NAME    "5x1 Array"
#define  CLASS2_NAME    "6x4 Array"
#define  VDATA1_NAME    "First Vdata"
#define  VDATA2_NAME    "Second Vdata"
#define  FIELD1_NAME    "Single-component Field"
#define  FIELD2_NAME    "Multi-component Field"
#define  N_RECORDS_1    5    /* number of records the first vdata contains  */
#define  N_RECORDS_2    6    /* number of records the second vdata contains */
#define  ORDER_2        4    /* order of the field in the second vdata      */
                /* Note that the order of the field in the first vdata is 1 */
```

```
main( )
{
   /************************* Variable declaration **************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32;     /* returned status for functions returning an int32 */
   int32 file_id, vdata1_ref, vdata2_ref;

   /*
   * Define an array to buffer the data of the first vdata.
   */
   char8 vdata1_buf [N_RECORDS_1] = {’V’, ’D’, ’A’, ’T’, ’A’};

   /*
   * Define an array to buffer the data of the second vdata.
   */
   int32 vdata2_buf [N_RECORDS_2][ORDER_2] = {{1, 2, 3, 4}, {2, 4, 6, 8},
                                               {3, 6, 9, 12}, {4, 8, 12, 16},
                                              {5, 10, 15, 20}, {6, 12, 18, 24}};

   /********************* End of variable declaration *********************/

   /*
   * Open the HDF file for writing.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);

   /*
   * Create the first vdata and populate it with data from the vdata1_buf
   * array. Note that the buffer vdata1_buf is cast to (uint8 *) for the
   * benefit of generic data type.
   */
   vdata1_ref = VHstoredata (file_id, FIELD1_NAME, (uint8 *)vdata1_buf,
                    N_RECORDS_1, DFNT_CHAR8, VDATA1_NAME, CLASS1_NAME);

   /*
   * Create the second vdata and populate it with data from the vdata2_buf
   * array.
   */
   vdata2_ref = VHstoredatam (file_id, FIELD2_NAME, (uint8 *)vdata2_buf,
            N_RECORDS_2, DFNT_INT32, VDATA2_NAME, CLASS2_NAME, ORDER_2);

   /*
   * Terminate access to the VS interface and close the HDF file.
   */
   status_n = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
 program create_onefield_vdatas
      implicit none
C
C     Parameter declaration
C
      character*18 FILE_NAME
      character*9  CLASS1_NAME
      character*9  CLASS2_NAME
      character*11 VDATA1_NAME
      character*12 VDATA2_NAME
      character*22 FIELD1_NAME
      character*21 FIELD2_NAME
      integer      N_RECORDS_1, N_RECORDS_2
      integer      ORDER_2
C
      parameter (FILE_NAME   = 'General_Vdatas.hdf',
     +           CLASS1_NAME = '5x1 Array',
     +           CLASS2_NAME = '6x4 Array',
     +           VDATA1_NAME = 'First Vdata',
     +           VDATA2_NAME = 'Second Vdata',
     +           FIELD1_NAME = 'Single-component Field',
     +           FIELD2_NAME = 'Multi-component Field')
      parameter (N_RECORDS_1 = 5,
     +           N_RECORDS_2 = 6,
     +           ORDER_2     = 4)

      integer DFACC_WRITE, DFNT_CHAR8, DFNT_INT32
      parameter (DFACC_WRITE = 2,
     +           DFNT_CHAR8  = 4,
     +           DFNT_INT32  = 24)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vhfscd, vhfsdm, vfend

C
C**** Variable declaration *******************************************
C
      integer   status
      integer   file_id
      integer   vdata1_ref, vdata2_ref
      character vdata1_buf(N_RECORDS_1)
      integer   vdata2_buf(ORDER_2, N_RECORDS_2)
      data vdata1_buf /'V','D','A','T','A'/
      data vdata2_buf / 1,  2,  3,  4,
     +                  2,  4,  6,  8,
     +                  3,  6,  9, 12,
     +                  4,  8, 12, 16,
     +                  5, 10, 15, 20,
     +                  6, 12, 18, 24/
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
C     Create the first vdata and populate it with data from vdata1_buf array.
C
      vdata1_ref = vhfscd(file_id, FIELD1_NAME, vdata1_buf, N_RECORDS_1,
     +                    DFNT_CHAR8, VDATA1_NAME, CLASS1_NAME)
C
C     Create the second vdata and populate it with data from vdata2_buf array.
C
```

```
       vdata2_ref = vhfsdm(file_id, FIELD2_NAME, vdata2_buf, N_RECORDS_2,
     +                     DFNT_INT32, VDATA2_NAME, CLASS2_NAME,
     +                     ORDER_2)
C
C      Terminate access to the VS interface and close the HDF file.
C
       status = vfend(file_id)
       status = hclose(file_id)
       end
```

4.5. Writing to Multi-Field Vdatas

There are several steps involved in creating ***general vdatas*** with more than one field: define the vdata, define the fields of the vdata, and write the vdata to the file. These steps are usually executed within a single program, although it is also possible to define an empty vdata in anticipation of writing data to it at a later time.

4.5.1. Creating Vdatas

Creating an empty vdata involves the following steps:

1. Open a file.
2. Initialize the Vdata interface.
3. Create the new vdata.
4. Assign a vdata name. (optional)
5. Assign a vdata class. (optional)
6. Define the fields.
7. Initialize fields for writing.
8. Set the interlace mode.
9. Dispose of the vdata identifier.
10. Terminate access to the Vdata interface.
11. Close the file.

Like the high-level VH interface, the Vdata interface does not retain default settings from one operation to the next or from one file to the next. Each time a vdata is created, its definitions must be explicitly reset.

To create a multi-field vdata, the calling program must contain the following:

```
C:        file_id = Hopen(filename, file_access_mode, num_dds_block);
          status = Vstart(file_id);
          vdata_id = VSattach(file_id, -1, vdata_access_mode);
          status = VSsetname(vdata_id, vdata_name);
          status = VSsetclass(vdata_id, vdata_class);
          status = VSfdefine(vdata_id, fieldname1, data_type1, order1);
           . . . . . . . . . .
          status = VSfdefine(vdata_id, fieldnameN, data_typeN, orderN);
          status = VSsetfields(vdata_id, fieldname_list);
          status = VSsetinterlace(vdata_id, interlace_mode);
          status = VSdetach(vdata_id);
          status = Vend(file_id);
          status = Hclose(file_id);
```

```
FORTRAN:  file_id = hopen(filename, file_access_mode, num_dds_block)
          status = vfstart(file_id)
          vdata_id = vsfatch(file_id, -1, vdata_access_mode)
          status = vsfsnam(vdata_id, vdata_name)
          status = vsfscls(vdata_id, vdata_class)
          status = vsffdef(vdata_id, fieldname1, data_type1, order1)
           . . . . . . . . . .
          status = vsffdef(vdata_id, fieldnameN, data_typeN, orderN)
          status = vsfsfld(vdata_id, fieldname_list)
          status = vsfsint(vdata_id, interlace_mode)
          status = vsfdtch(vdata_id)
          status = vfend(file_id)
          status = hclose(file_id)
```

In the routines that follow, *vdata_id* is the vdata identifier returned by **VSattach**.

4.5.1.1. Assigning a Vdata Name and Class: VSsetname and VSsetclass

VSsetname assigns a name to a vdata. If not explicitly named by a call to **VSsetname**, the name of the vdata is set by default to `NULL`. A name may be assigned and reassigned at any time after the vdata is created. The parameter *vdata_name* contains the name to be assigned to the vdata.

VSsetclass assigns a class to a vdata. If **VSsetclass** is not called, the vdata's class is set by default to `NULL`. As with the vdata name, the class may be assigned and reassigned any time after the vdata is created. The parameter *vdata_class* contains the class name to be assigned to the vdata.

VSsetname and **VSsetclass** return either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for these routines are further defined in Table 4E.

4.5.1.2. Defining a Field within a Vdata: VSfdefine

VSfdefine defines a field within a newly-created vdata. Each **VSfdefine** call assigns the name contained in the argument *fieldname*, the data type contained in the argument *data_type*, and the order contained in the argument *order* to one new field. Once data is written to a vdata, the name, data type and order of the field may not be modified or deleted.

The Vdata interface also provides certain ***predefined fields.*** A predefined field has a specific name, data type, and order, so there is no need to call **VSfdefine** to define a predefined field. Some applications may require the use of predefined fields in vdatas. Available predefined fields are discussed in Table 4D.

Note that **VSfdefine** does not allocate memory for the field, but simply introduces the field. The field definition must be completed by **VSsetfields**, which is discussed in Section "*Initializing the Fields for Write Access: VSsetfields*".

VSfdefine returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSfdefine** are further described in Table 4E.

TABLE 4D **Predefined Data Types and Field Names for Vdata Fields**

Data Type	Coordinate Point Field Names			Normal Component Field Names		
	x-coordinate	y-coordi-nate	z-coordi-nate	x-compo-nent	y-compo-nent	z-compo-nent
float	PX	PY	PZ	NX	NY	NZ
integer	IX	IY	IZ	None	None	None

4.5.1.3. Initializing the Fields for Write Access: VSsetfields

VSsetfields initializes read and write access to the fields in a vdata. It must be called prior to read or write operations. Initializing for read access is discussed in Section "*Initializing the Fields for Read Access: VSsetfields*". For writing, **VSsetfields** specifies the fields to be written and the order in which they are to be placed.

The parameter *fieldname_list* is a comma-separated list of the field names, with no white space included. The fields can be either the predefined fields or the fields that have been previously introduced by **VSfdefine**. **VSfdefine** allows a user to declare a field, along with its data type and order, but **VSsetfields** finalizes the definition by allowing the user to select the fields that are to be included in the vdata. Thus, any fields created by **VSfdefine** that are not in the parameter *field-name_list* of **VSsetfields** will be ignored. This feature was originally intended for interactive-mode users. The combined width of the fields in the parameter *fieldname_list* is also the length of the record and must be less than `MAX_FIELD_SIZE` (or `65535`). An attempt to create a larger record will cause **VSsetfields** to return `FAIL` (or `-1`).

VSsetfields returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSsetfields** are further defined in Table 4E.

4.5.1.4. Specifying the Interlace Mode: VSsetinterlace

The Vdata interface supports two types of interlacing: ***file interlacing*** and ***buffer interlacing***. ***File interlacing*** determines how data is stored in a file and ***buffer interlacing*** determines how data is stored in memory. The Vdata interface can write data from a buffer to a file in an interlaced or non-interlaced manner. It can also read data from a file in an interlaced or non-interlaced manner.

The **VSread** and **VSwrite** routines set the buffer's interlace mode. The **VSwrite** routine will be discussed in Section "*Writing to a Vdata: VSwrite*" and the **VSread** routine will be discussed in Section "*Reading from the Current Vdata: VSread*".

VSsetinterlace sets the file interlacing mode for a vdata. Setting the parameter *interlace_mode* to `FULL_INTERLACE` (or `0`) fills the vdata by record, whereas specifying `NO_INTERLACE` (or `1`) fills the vdata by field. (See Figure 4d) For multi-component fields, all components are treated as a single field.

As with file interlacing, the default buffer interlace mode is `FULL_INTERLACE` because it is more efficient to write complete records than it is to write fields if the file and buffer interlace modes are the same, although both require the same amount of disk space.

In Figure 4d, the illustrated vdata has four fields and three records.

FIGURE 4d **Interlaced and Non-Interlaced Vdata Contents**

Vdata			
Mixed_Data_Type			
Temp	**Height**	**Speed**	**Ident**
1.11	**1**	**11.11**	**A**
2.22	**2**	**22.22**	**B**
3.33	**3**	**33.33**	**C**

Interlacing Mode: FULL_INTERLACE

Vdata			
Mixed_Data_Type			
Temp	**1.11**	**2.22**	**3.33**
Height	**1**	**2**	**3**
Speed	**11.11**	**22.22**	**33.33**
Ident	**A**	**B**	**C**

Interlacing Mode: NO_INTERLACE

VSsetinterlace can only be used for operations on new vdatas as the interlacing cannot be changed once the data has been written to a vdata. Records in a fully interlaced vdata can be written record-by-record and, thus, can be appended; however, all records in a non-interlaced vdata must be written at the same time.

VSsetinterlace returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSsetinterlace** are further described in Table 4E.

4.5.1.5. Specifying External Storage Information: VSsetexternalfile

The HDF library allows applications to store vdata tables in an ***external file*** that is separate from the ***primary file*** containing the metadata for the vdata. The library keeps track of the beginning of the vdata and adds data at the appropriate position in the external file. When data is written or appended, the HDF library writes data to the external file and updates the appropriate metadata in the primary file.

VSsetexternalfile specifies that an external data file is to be used to store the data of the given vdata. The parameter *filename* is the name of the external data file and *offset* is the number of bytes from the beginning of the external file to the location where the first byte of data should be written. The syntax for **VSsetexternalfile** is as followed:

```
C:          status = VSsetexternalfile(vdata_id, filename, &offset)

FORTRAN:    status = vsfsextf(vdata_id, filename, offset)
```

If a file with the name specified by *filename* exists in the current directory search path, the function will access it as the external file. It is the user's responsibility to make sure that the external data file is kept with the primary HDF file.

VSsetexternalfile returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSsetexternalfile** are further described in Table 4E.

TABLE 4E **VSsetname, VSsetclass, VSfdefine, VSsetfields, and VSsetinterlace Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSsetname [int32] **(vsfsnam)**	vdata_id	int32	integer	Vdata identifier
	vdata_name	char *	character*(*)	Vdata name
VSsetclass [int32] **(vsfscls)**	vdata_id	int32	integer	Vdata identifier
	vdata_class	char *	character*(*)	Vdata name
VSfdefine [intn] **(vsffdef)**	vdata_id	int32	integer	Vdata identifier
	fieldname	char *	character*(*)	Name of the field to be defined
	data_type	int32	integer	Type of the field data
	order	int32	integer	Order of the new field
VSsetfields [intn] **(vsfsfld)**	vdata_id	int32	integer	Vdata identifier
	fieldname_list	char *	character*(*)	Names of the vdata fields to be accessed
VSsetinterlace [intn] **(vsfsint)**	vdata_id	int32	integer	Vdata identifier
	interlace_mode	int32	integer	Interlace mode
VSsetexternalfile **[intn]** **(vsfsextf)**	vdata_id	int32	integer	Vdata identifier
	filename	char *	character*(*)	External file name
	offset	int32	integer	Offset of external data

4.5.2. Writing Data to Vdatas

This section describes the vdata writing operation (**VSwrite**), random access to vdata (**VSseek**), and packing and unpacking mechanisms that allow storing vdata fields of different data types (**VSfpack**).

Writing to a vdata requires the following steps:

1. Open a file.
2. Initialize the Vdata interface.
3. Initialize fields for writing.
4. Initiate access to the vdata.
5. Seek to the target record.
6. Write the data.
7. Dispose of the vdata identifier.
8. Terminate access to the Vdata interface.
9. Close the file.

These steps correspond to the following sequence of function calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);
            vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
            status = VSsetfields(vdata_id, fieldname_list);
            record_pos = VSseek(vdata_id, record_index);
            num_of_recs = VSwrite(vdata_id, databuf, n_records, interlace_mode);
            status = VSdetach(vdata_id);
            status = Vend(file_id);
```

```
              status = Hclose(file_id);

FORTRAN:      file_id = hopen(filename, file_access_mode, num_dds_block)
              status = vfstart(file_id)
              vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
              status = vsfsfld(vdata_id, fieldname_list);
              record_pos = vsfseek(vdata_id, record_index);

              num_of_recs = vsfwrt(vdata_id, databuf, n_records, interlace_mode)
    OR        num_of_recs = vsfwrtc(vdata_id, databuf, n_records, interlace_mode)
    OR        num_of_recs = vsfwrit(vdata_id, databuf, n_records, interlace_mode)

              status = vsfdtch(vdata_id)
              status = vfend(file_id)
              status = hclose(file_id)
```

4.5.2.1. Resetting the Current Position within Vdatas: VSseek

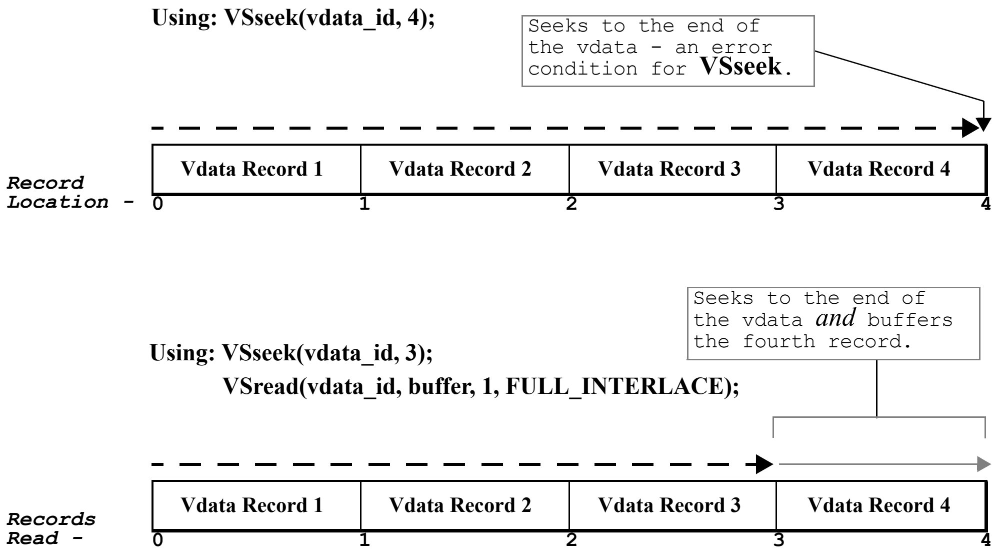
VSseek provides a mechanism for random access to fully-interlaced vdatas. Random-access for non-interlaced vdatas is not available. The parameter *record_index* is the position of the record to be written. The position of the first record in a vdata is specified by *record_index* = `0`. Any vdata operation will be performed on this record by default; vdata operations on other records require that **VSseek** be called first to specify the target record.

Note that **VSseek** has been designed for the purpose of *overwriting* data, not *appending* data. That means **VSseek** puts the current record pointer at the beginning of the sought record and the subsequent write will overwrite the record. To append data to a vdata, the current record pointer must be put at the end of the last record. Thus, you must seek to the last record then read this record so that the current record pointer will be put at the end of the record. A write operation will now start at the end of the last record in the vdata. Figure 4e illustrates a situation where **VSseek** can be misused while attempting to append data to the vdata and how **VSread** is called to correctly place the record pointer at the end of the vdata for appending.

Note that, because the record location numbering starts at 0, the record location and the value of the parameter *record_index* are off by 1. For example, reading the fourth record in the buffer requires *record_index* to be set to 3.

See the notes regarding the potential performance impact of appendable data sets in Section <><TextItalic>"Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance"

FIGURE 4e **Setting the Record Pointer to the End of a Vdata**


Using: VSseek(vdata_id, 4);
Seeks to the end of the vdata - an error condition for VSseek.
Vdata Record 1
Vdata Record 2
Vdata Record 3
Vdata Record 4
Record Location -
0
1
2
3
4
Seeks to the end of the vdata and buffers the fourth record.
Using: VSseek(vdata_id, 3);
VSread(vdata_id, buffer, 1, FULL_INTERLACE);
Records Read -

In this illustration, the vdata to which we plan to append data contains 4 records. Using **VSseek** to seek to the end of the fourth record by setting the parameter *record_index* to 4 results in an error condition. Setting the parameter *record_index* to 3 places the current record pointer at the beginning of the fourth record. We then use **VSread** to read the contents of the fourth record into a buffer; this moves the current record pointer to the end of the fourth record. The contents of the buffer can then be discarded and a write operation can be called to append data to the end of the vdata.

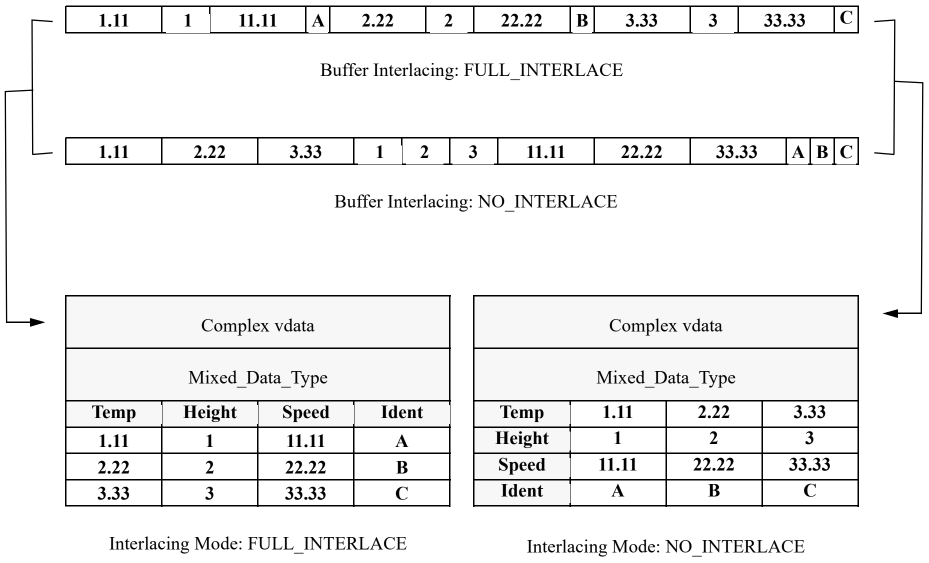
VSseek returns the sought record location or `FAIL` (or `-1`). Its parameters are further defined in Table 4F.

4.5.2.2. Writing to a Vdata: VSwrite

VSwrite writes buffered data to a specified vdata. The parameter *databuf* is a buffer containing the records to be stored in the vdata. The parameter *n_records* specifies the number of records to be stored.

Recall that the *file interlacing* is set by **VSsetinterlace** when the vdata is created, and the *buffer interlacing* is specified by the parameter *interlace_mode* in the call to **VSwrite** when data is written to the file. The array *databuf* is assumed to be organized in memory as specified by *interlace_mode*. Setting *interlace_mode* to `FULL_INTERLACE` (or `0`) indicates that the array in memory is organized by record, whereas to `NO_INTERLACE` (or `1`) indicates that the array is organized by field. (See Figure 4f) **VSwrite** will write interlaced or non-interlaced data to a vdata in a file: interlaced data in the buffer can be written to the vdata in the file as non-interlaced data and vice versa. If the data is to be stored with an interlace mode different from that of the buffer, **VSsetinterlace** (described in Section "*Specifying the Interlace Mode: VSsetinterlace*") must be called prior to **VSwrite**. Multiple write operations can only be used on fully-interlaced vdatas in the file.

FIGURE 4f **Writing Interlaced or Non-interlaced Buffers into Interlaced or Non-interlaced Vdatas**


1.11 | 1 | 11.11 | A | 2.22 | 2 | 22.22 | B | 3.33 | 3 | 33.33 | C
Buffer Interlacing: FULL_INTERLACE
1.11 | 2.22 | 3.33 | 1 | 2 | 3 | 11.11 | 22.22 | 33.33 | A | B | C
Buffer Interlacing: NO_INTERLACE
Complex vdata
Mixed_Data_Type
Temp Height Speed Ident
1.11 1 11.11 A
2.22 2 22.22 B
3.33 3 33.33 C
Interlacing Mode: FULL_INTERLACE
Complex vdata
Mixed_Data_Type
Temp 1.11 2.22 3.33
Height 1 2 3
Speed 11.11 22.22 33.33
Ident A B C
Interlacing Mode: NO_INTERLACE

The data in the array *databuf* is assumed to contain the exact amount of data in the order needed to fill the fields defined in the last call to **VSsetfields**. Because **VSwrite** writes the contents of *databuf* contiguously to the vdata, any “padding” due to record alignment must be removed before attempting to write from *databuf* to the vdata. For more information on alignment padding see Section "*Packing or Unpacking Field Data: VSfpack"*.

It should be remembered that **VSwrite** writes whole records, not individual fields. If a modification to one field within a previously-written record is needed, the contents of the record must first be preserved by reading it to a buffer with **VSread**, which will be described in Section "*Reading from the Current Vdata: VSread"*; the record must then be updated in the buffer and written back to the file with **VSwrite**.

To store a vdata to the file after being created, either **VSsetname**, **VSsetfields**, or **VSwrite** must be called before **VSdetach** for the vdata. If **VSwrite** is not called, the vdata created will be empty.

The FORTRAN-77 version of **VSwrite** has three routines: **vsfwrt** is for buffered numeric data, **vsfwrtc** is for buffered character data and **vsfwrit** is for generic packed data.

VSwrite returns the total number of records written or `FAIL` (or `-1`). Its parameters are further defined in Table 4F.

TABLE 4F

VSseek and VSwrite Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSseek [int32] **(vsfseek)**	vdata_id	int32	integer	Vdata identifier
	record_index	int32	integer	Index of the record to seek to
VSwrite [int32] **(vsfwrt/vsfwrtc/ vsfwrit)**	vdata_id	int32	integer	Vdata identifier
	databuf	uint8*	<valid numeric data type>(*) / character*(*) / integer	Buffer containing data to be written
	n_records	int32	integer	Number of records to be written
	interlace_mode	int32	integer	Interlace mode of the buffered data

EXAMPLE 3.

Writing a Vdata of Homogeneous Type

This example illustrates the use of **VSfdefine/vsffdef**, **VSsetname/vsfsnam**, **VSsetclass/vsfscls**, **VSsetfields/vsfsfld**, and **VSwrite/vsfwrt** to create and write a three-field vdata to the file "General_Vdatas.hdf". Although the fields have data of the same type, they have different orders.

To clarify the illustration, let us assume that the vdata is used to contain the data of some particles collected from an experiment. Each record of the data includes the position of a particle, its weight, and the minimum and maximum temperature the particle can endure. The vdata is named "Solid Particle", contains 10 records, and belongs to a class, named "Particle Data". The fields of the vdata include "Position", "Mass", and "Temperature". The field "Position" has an order of 3 for the x, y, and z values representing the position of a particle. The field "Mass" has an order of 1. The field "Temperature" has an order of 2 for the minimum and maximum temperature. The program creates the vdata, sets its name and class name, defines its fields, and then writes the data to it.

C:

```
#include "hdf.h"

#define  FILE_NAME        "General_Vdatas.hdf"
#define  N_RECORDS        10        /* number of records the vdata contains */
#define  ORDER_1          3         /* order of first field */
#define  ORDER_2          1         /* order of second field */
#define  ORDER_3          2         /* order of third field */
#define  CLASS_NAME       "Particle Data"
#define  VDATA_NAME       "Solid Particle"
#define  FIELD1_NAME      "Position"      /* contains x, y, z values */
#define  FIELD2_NAME      "Mass"          /* contains weight values */
#define  FIELD3_NAME      "Temperature"   /* contains min and max values */
#define  FIELDNAME_LIST   "Position,Mass,Temperature" /* No spaces b/w names */

/* number of values per record */
#define  N_VALS_PER_REC   (ORDER_1 + ORDER_2 + ORDER_3)

main( )
{
   /************************* Variable declaration *************************/

   intn   status_n;     /* returned status for functions returning an intn  */
   int32  status_32,    /* returned status for functions returning an int32 */
          file_id, vdata_id,
          vdata_ref = -1,    /* ref number of a vdata, set to -1 to create  */
          num_of_records;    /* number of records actually written to vdata */
   int16  rec_num;           /* current record number */
   float32  data_buf[N_RECORDS][N_VALS_PER_REC]; /* buffer for vdata values */

   /********************** End of variable declaration **********************/

   /*
   * Open the HDF file for writing.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);

   /*
   * Create a new vdata.
   */
   vdata_id = VSattach (file_id, vdata_ref, "w");

   /*
   * Set name and class name of the vdata.
   */
   status_32 = VSsetname (vdata_id, VDATA_NAME);
   status_32 = VSsetclass (vdata_id, CLASS_NAME);

   /*
   * Introduce each field’s name, data type, and order.  This is the first
   * part in defining a field.
   */
   status_n = VSfdefine (vdata_id, FIELD1_NAME, DFNT_FLOAT32, ORDER_1 );
   status_n = VSfdefine (vdata_id, FIELD2_NAME, DFNT_FLOAT32, ORDER_2 );
   status_n = VSfdefine (vdata_id, FIELD3_NAME, DFNT_FLOAT32, ORDER_3 );

   /*
   * Finalize the definition of the fields.
   */
   status_n = VSsetfields (vdata_id, FIELDNAME_LIST);

   /*
   * Buffer the data by the record for fully interlaced mode.  Note that the
   * first three elements contain the three values of the first field, the
   * fourth element contains the value of the second field, and the last two
   * elements contain the two values of the third field.
   */
   for (rec_num = 0; rec_num < N_RECORDS; rec_num++)
   {
      data_buf[rec_num][0] = 1.0 * rec_num;
      data_buf[rec_num][1] = 2.0 * rec_num;
      data_buf[rec_num][2] = 3.0 * rec_num;
      data_buf[rec_num][3] = 0.1 + rec_num;
      data_buf[rec_num][4] = 0.0;
      data_buf[rec_num][5] = 65.0;
   }

   /*
   * Write the data from data_buf to the vdata with full interlacing mode.
   */
   num_of_records = VSwrite (vdata_id, (uint8 *)data_buf, N_RECORDS,
                             FULL_INTERLACE);
   /*
   * Terminate access to the vdata and to the VS interface, then close
   * the HDF file.
   */
   status_32 = VSdetach (vdata_id);
   status_n  = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
 program write_to_vdata
      implicit none
C
C     Parameter declaration
C
      character*18 FILE_NAME
      character*13 CLASS_NAME
      character*14 VDATA_NAME
      character*8  FIELD1_NAME
      character*4  FIELD2_NAME
      character*11 FIELD3_NAME
      character*27 FIELDNAME_LIST
      integer      N_RECORDS
      integer      ORDER_1, ORDER_2, ORDER_3
      integer      N_VALS_PER_REC
C
      parameter (FILE_NAME       = 'General_Vdatas.hdf',
     +           CLASS_NAME      = 'Particle Data',
     +           VDATA_NAME      = 'Solid Particle',
     +           FIELD1_NAME     = 'Position',
     +           FIELD2_NAME     = 'Mass',
     +           FIELD3_NAME     = 'Temperature',
     +           FIELDNAME_LIST = 'Position,Mass,Temperature')
      parameter (N_RECORDS = 10,
     +           ORDER_1   = 3,
     +           ORDER_2   = 1,
     +           ORDER_3   = 2,
     +           N_VALS_PER_REC = ORDER_1 + ORDER_2 + ORDER_3)

      integer DFACC_WRITE, DFNT_FLOAT32, FULL_INTERLACE
      parameter (DFACC_WRITE    = 2,
     +           DFNT_FLOAT32   = 5,
     +           FULL_INTERLACE = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vsfatch, vsfsnam, vsfscls, vsffdef, vsfsfld,
     +        vsfwrt, vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, vdata_id
      integer vdata_ref, rec_num, num_of_records
      real    data_buf(N_VALS_PER_REC, N_RECORDS)
C
C**** End of variable declaration ***********************************
C
C
C     Open the HDF file for writing.
```

```
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
C     Create a new vdata.
C
      vdata_ref = -1
      vdata_id = vsfatch(file_id, vdata_ref, 'w')
C
C     Set name and class name of the vdata.
C
      status = vsfsnam(vdata_id, VDATA_NAME)
      status = vsfscls(vdata_id, CLASS_NAME)
C
C     Introduce each field's name, data type, and order. This is the
C     first part in defining a field.
C
      status = vsffdef(vdata_id, FIELD1_NAME, DFNT_FLOAT32, ORDER_1)
      status = vsffdef(vdata_id, FIELD2_NAME, DFNT_FLOAT32, ORDER_2)
      status = vsffdef(vdata_id, FIELD3_NAME, DFNT_FLOAT32, ORDER_3)
C
C     Finalize the definition of the fields.
C
      status = vsfsfld(vdata_id, FIELDNAME_LIST)
C
C     Buffer the data by the record for fully interlaced mode. Note that the
C     first three elements contain the three values of the first field,
C    the forth element contains the value of the second field, and the last two
C     elements contain the two values of the third field.
C
      do 10 rec_num = 1, N_RECORDS
         data_buf(1, rec_num) = 1.0 * rec_num
         data_buf(2, rec_num) = 2.0 * rec_num
         data_buf(3, rec_num) = 3.0 * rec_num
         data_buf(4, rec_num) = 0.1 + rec_num
         data_buf(5, rec_num) = 0.0
         data_buf(6, rec_num) = 65.0
10    continue
C
C     Write the data from data_buf to the vdata with the full interlacing mode.
C
      num_of_records = vsfwrt(vdata_id, data_buf, N_RECORDS,
     +                        FULL_INTERLACE)
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.5.2.3. Setting Up Linked Block Vdatas: VSsetblocksize and VSsetnumblocks

Unless otherwise specified, Vdata data sets stored in linked blocks employ a default size and number of linked blocks, as set in `HDF_APPENDABLE_BLOCK_LEN` and `HDF_APPENDABLE_BLOCK_NUM`, respectively. **VSsetblocksize** and **VSsetnumblocks** provide a mechanism for managing these values when the defaults are not appropriate.

VSsetblocksize and **VSsetnumblocks** can be called to change the default linked block settings. The parameter *vdata_id* identifies the vdata. The size of blocks is specified in bytes in *block_size* and number of blocks in *num_blocks*.

VSsetblocksize and **VSsetnumblocks** must be called before any data is written to a vdata; once a linked block element has been created, neither the block size nor the number blocks can be changed. Further note that **VSsetblocksize** sets the block size only for blocks following the first block.

See the notes regarding the potential performance impact of block size in Section <><TextI-talic>"Tuning Linked Block Size to Enhance Performance"

VSsetblocksize and **VSsetnumblocks** both return `SUCCESS` (or `0`) upon successful completion or `FAIL` (or `-1`). Their parameters are further defined in Table 4G.

TABLE 4G **VSsetblocksize and VSsetnumblocks Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSsetblocksize [intn] **(vsfsetblsz)**	vdata_id	int32	integer	Vdata identifier
	block_size	int32	integer	Size of each block, in bytes
VSsetnumblocks [intn] **(vsfsetnmbl)**	vdata_id	int32	integer	Vdata identifier
	num_blocks	int32	integer	Number of blocks to be used for the linked-block element

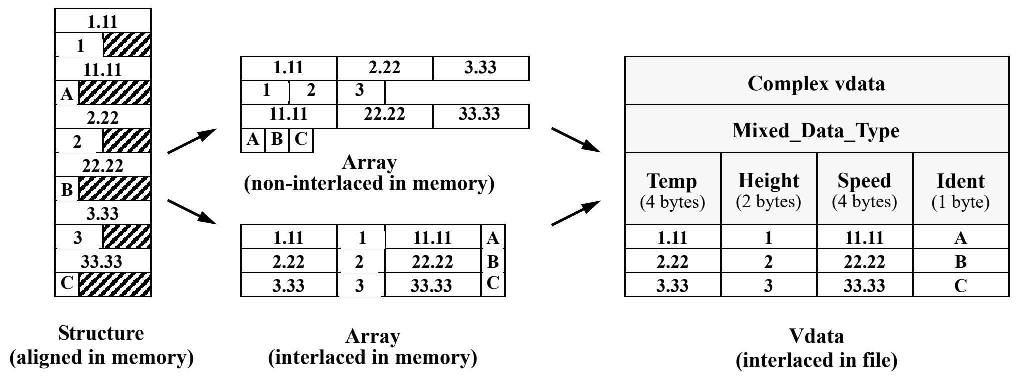
4.5.2.4. Packing or Unpacking Field Data: VSfpack

Storing fields of mixed data types is an efficient use of disk space and is useful in applications that use structures. However, while data structures in memory containing fields of variable lengths can contain alignment bytes, field data stored in a vdata cannot include them. This is true for both fully-interlaced and non-interlaced data. Because of this storing limitation, when variable-length field types are used, it is generally not possible to write data directly from a structure in memory into a vdata in a file with a **VSwrite** call or to read data directly into a buffer from the vdata with a call to **VSread**. Thus, when writing, **VSfpack** is used to pack field data into a temporary buffer by removing the padding, or alignment bytes, and when reading, to unpack field data into vdata fields by adding necessary alignment bytes. The syntax for **VSfpack** is as follows:

```
C:          status = VSfpack(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs);

FORTRAN:    status = vsfcpak(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs)

     OR     status = vsfnpak(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs)
```

The process of removing the alignment bytes is called "packing the array." An illustration of this process is provided in Figure 4g. The data provided by the user is stored in the structure in memory. The field values are aligned with padded bytes. **VSfpack** packs the data into the array in memory after removing the padded bytes. The packed data is then written to the vdata in the file by **VSwrite**.

FIGURE 4g **Removing Alignment Bytes When Writing Data From a C Structure to a Vdata**



The process illustrated in Figure 4g can be read in the reverse direction for "unpacking the array," that is when using **VSfpack** to fill a structure in memory with vdata field data. In this case, alignment bytes are added to the field data to make the data conform to the specific alignment requirements of the platform.

VSfpack performs both tasks, packing and unpacking, and the parameter *action* specifies the appropriate action for the routine. Valid values for the parameter *action* are `_HDF_VSPACK` (or `0`) for packing and `_HDF_VSUNPACK` (or `1`) for unpacking.

The calling program must allocate sufficient space for the buffer *buf* to hold all packed or unpacked fields. The parameter *buf_size* specifies the size of the buffer *buf* and should be at least *n_records* *(the total size of all fields specified in *fields_in_buf*).

When **VSfpack** is called to pack field values into *buf*, the parameter *fields_in_buf* must specify all fields of the vdata. This can be accomplished either by listing all of the field names in *fields_in_buf* or by setting *fields_in_buf* to `NULL` in C or to one blank character in FORTRAN-77.

When **VSfpack** is called to unpack field values, the parameter *fields_in_buf* may specify a subset of the vdata fields. The parameter *fields_in_buf* can be set to `NULL` in C or to one space character in FORTRAN-77 to specify all fields in the vdata.

The parameter *fieldname_list* specifies the field(s) to be packed or unpacked. The parameter *bufptrs* provides pointers to the buffers for each field to be packed or unpacked. The calling program is responsible for allocating sufficient space for each field buffer. Significant differences between the C and FORTRAN-77 functionality are described in the following paragraphs.

In C, *fieldname_list* can list either all of the fields specified by *fields_in_buf* or a subset of those fields. Only if *fields_in_buf* specifies all of the vdata fields, then *fields_in_buf* can be set to `NULL` to specify all vdata fields. The parameter *bufptrs* contains an array of pointers to the buffers where field data will be packed or unpacked.

The FORTRAN-77 routines can pack or unpack only one field at a time, so the parameter *fieldname_list* contains only the name of that field. The parameter *bufptrs* is the buffer for that field.

The FORTRAN-77 version of **VSfpack** has two routines: **vsfcpak** packs or unpacks character data and **vsfnpak** packs or unpacks numeric data. Refer to the FORTRAN-77 version in Example 4 for a more specific illustration.

VSfpack returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSfpack** are described in Table 4H.

TABLE 4H

VSfpack Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSfpack [intn] **(vsfcpak/vsfnpak)**	vdata_id	int32	integer	Vdata identifier
	action	intn	integer	Action to be performed
	fields_in_buf	char *	character*(*)	Fields in the buffer `buf` to write or read from the vdata
	buf	VOIDP	integer	Buffer for the vdata values
	buf_size	intn	integer	Buffer size in bytes
	n_records	intn	integer	Number of records to pack or unpack
	fieldname_list	char *	character*(*)	Names of the fields to be packed or unpacked
	bufptrs	VOIDP	<valid numeric data type>(*)/ character*(*)	Array of pointers to the field buffers in C and field buffer in FORTRAN-77

EXAMPLE 4.

Writing a Multi-field and Mixed-type Vdata with Packing

This example illustrates the use of **VSfpack/vsfnpak/vsfcpak** and **VSwrite/vsfwrit** to write a vdata with data of different types. Note that the approach used in Example 3 makes it difficult for the vdata to have mixed-type data.

In this example, the program creates an HDF file, named "Packed_Vdata.hdf", then defines a vdata which is named "Mixed Data Vdata" and belongs to class "General Data Class". The vdata contains four order-1 fields, "Temp", "Height", "Speed", and "Ident" of type float32, int16, float32, and char8, respectively. The program then packs the data in fully interlaced mode into a databuf and writes the packed data to the vdata. Note that, in the C example, a VSfpack call packs all N_RECORDS and a VSwrite call writes out all N_RECORDS records. In the Fortran example, N_RECORDS of each field are packed using separate calls to vsfnpak and vsfcpak; vsfwrit writes packed data to the vdata.

C:

```
#include "hdf.h"

#define  FILE_NAME        "Packed_Vdata.hdf"
#define  VDATA_NAME       "Mixed Data Vdata"
#define  CLASS_NAME       "General Data Class"
#define  FIELD1_NAME      "Temp"
#define  FIELD2_NAME      "Height"
#define  FIELD3_NAME      "Speed"
#define  FIELD4_NAME      "Ident"
#define  ORDER            1        /* number of values in the field         */
#define  N_RECORDS        20       /* number of records the vdata contains  */
#define  N_FIELDS         4        /* number of fields in the vdata         */
#define  FIELDNAME_LIST   "Temp,Height,Speed,Ident"  /* No spaces b/w names */

/* number of bytes of the data to be written, i.e., the size of all the
   field values combined times the number of records */
#define BUF_SIZE (2*sizeof(float32) + sizeof(int16) + sizeof(char)) * N_RECORDS

main( )
{
   /************************* Variable declaration *************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id, vdata_id,
         vdata_ref = -1,   /* vdata's reference number, set to -1 to create */
         num_of_records; /* number of records actually written to the vdata */
   float32 temp[N_RECORDS];       /* buffer to hold values of first field   */
   int16   height[N_RECORDS];     /* buffer to hold values of second field  */
   float32 speed[N_RECORDS];      /* buffer to hold values of third field   */
   char8   ident[N_RECORDS];      /* buffer to hold values of fourth field  */
   VOIDP   fldbufptrs[N_FIELDS];/*pointers to be pointing to the field buf-
fers*/
   uint16  databuf[BUF_SIZE]; /* buffer to hold the data after being packed*/
   int     i;

   /********************** End of variable declaration *********************/

   /*
   * Create an HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);

   /*
   * Create a new vdata.
   */
   vdata_id = VSattach (file_id, vdata_ref, "w");

   /*
   * Set name and class name of the vdata.
   */
   status_32 = VSsetname (vdata_id, VDATA_NAME);
   status_32 = VSsetclass (vdata_id, CLASS_NAME);

   /*
   * Introduce each field's name, data type, and order.  This is the first
   * part in defining a vdata field.
   */
   status_n = VSfdefine (vdata_id, FIELD1_NAME, DFNT_FLOAT32, ORDER);
   status_n = VSfdefine (vdata_id, FIELD2_NAME, DFNT_INT16, ORDER);
   status_n = VSfdefine (vdata_id, FIELD3_NAME, DFNT_FLOAT32, ORDER);
   status_n = VSfdefine (vdata_id, FIELD4_NAME, DFNT_CHAR8, ORDER);

   /*
   * Finalize the definition of the fields of the vdata.
   */
   status_n = VSsetfields (vdata_id, FIELDNAME_LIST);

   /*
   * Enter data values into the field buffers by the records.
   */
   for (i = 0; i < N_RECORDS; i++)
   {
      temp[i] = 1.11 * (i+1);
      height[i] = i;
      speed[i] = 1.11 * (i+1);
      ident[i] = 'A' + i;
   }

   /*
   * Build an array of pointers each of which points to a field buffer that
```

```
    * holds all values of the field.
    */
    fldbufptrs[0] = &temp[0];
    fldbufptrs[1] = &height[0];
    fldbufptrs[2] = &speed[0];
    fldbufptrs[3] = &ident[0];

    /*
    * Pack all data in the field buffers that are pointed to by the set of
    * pointers fldbufptrs, and store the packed data into the buffer
    * databuf.  Note that the second parameter is _HDF_VSPACK for packing.
    */
    status_n = VSfpack (vdata_id,_HDF_VSPACK, NULL, (VOIDP)databuf,
            BUF_SIZE, N_RECORDS, NULL, (VOIDP)fldbufptrs);

    /*
    * Write all records of the packed data to the vdata.
    */
    num_of_records = VSwrite (vdata_id, (uint8 *)databuf, N_RECORDS,
                              FULL_INTERLACE);

    /*
    * Terminate access to the vdata and the VS interface, then close
    * the HDF file.
    */
    status_32 = VSdetach (vdata_id);
    status_n = Vend (file_id);
    status_32 = Hclose (file_id);
}
```

FORTRAN:

```
      program write_mixed_vdata
      implicit none
C
C     Parameter declaration
C
      character*16 FILE_NAME
      character*18 CLASS_NAME
      character*16 VDATA_NAME
      character*4  FIELD1_NAME
      character*6  FIELD2_NAME
      character*5  FIELD3_NAME
      character*5  FIELD4_NAME
      character*23 FIELDNAME_LIST
      integer      N_RECORDS, N_FIELDS, ORDER
      integer      BUF_SIZE
C
      parameter (FILE_NAME       = 'Packed_Vdata.hdf',
     +           CLASS_NAME      = 'General Data Class',
     +           VDATA_NAME      = 'Mixed Data Vdata',
     +           FIELD1_NAME     = 'Temp',
     +           FIELD2_NAME     = 'Height',
     +           FIELD3_NAME     = 'Speed',
     +           FIELD4_NAME     = 'Ident',
     +           FIELDNAME_LIST = 'Temp,Height,Speed,Ident')
      parameter (N_RECORDS = 20,
     +           N_FIELDS  = 4,
     +           ORDER     = 1,
     +           BUF_SIZE = (4 + 2 + 4 + 1)*N_RECORDS)

      integer DFACC_WRITE, DFNT_FLOAT32, DFNT_INT16, DFNT_CHAR8,
     +        FULL_INTERLACE, HDF_VSPACK
```

```
      parameter (DFACC_WRITE    = 2,
     +           DFNT_FLOAT32   = 5,
     +           DFNT_INT16     = 22,
     +           DFNT_CHAR8     = 4,
     +           FULL_INTERLACE = 0,
     +           HDF_VSPACK     = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vsfatch, vsfsnam, vsfscls, vsffdef, vsfsfld,
     +        vsfnpak, vsfcpak, vsfwrit, vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer   status
      integer   file_id, vdata_id
      integer   vdata_ref, num_of_records
      real      temp(N_RECORDS)
      integer*2 height(N_RECORDS)
      real      speed(N_RECORDS)
      character ident(N_RECORDS)
      integer   i
C
C     Buffer for packed data should be big enough to hold N_RECORDS.
C
      integer   databuf(BUF_SIZE/4 + 1)
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
C     Create a new vdata.
C
      vdata_ref = -1
      vdata_id = vsfatch(file_id, vdata_ref, 'w')
C
C     Set name and class name of the vdata.
C
      status = vsfsnam(vdata_id, VDATA_NAME)
      status = vsfscls(vdata_id, CLASS_NAME)
C
C     Introduce each field's name, data type, and order. This is the
C     first part in defining a field.
C
      status = vsffdef(vdata_id, FIELD1_NAME, DFNT_FLOAT32, ORDER)
      status = vsffdef(vdata_id, FIELD2_NAME, DFNT_INT16, ORDER)
      status = vsffdef(vdata_id, FIELD3_NAME, DFNT_FLOAT32, ORDER)
      status = vsffdef(vdata_id, FIELD4_NAME, DFNT_CHAR8, ORDER)
C
C     Finalize the definition of the fields.
C
      status = vsfsfld(vdata_id, FIELDNAME_LIST)
C
C     Enter data values into the field databufs by the records.
```

```
C
      do 10 i = 1, N_RECORDS
         temp(i)   = 1.11 * i
         height(i) = i - 1
         speed(i)  = 1.11 * i
         ident(i)  = char(64+i)
10    continue
C
C     Pack N_RECORDS of data into databuf. In Fortran, each field is packed
C     using separate calls to vsfnpak or vsfcpak.
C
      status = vsfnpak(vdata_id, HDF_VSPACK, ’ ’, databuf, BUF_SIZE,
     +                 N_RECORDS, FIELD1_NAME, temp)
      status = vsfnpak(vdata_id, HDF_VSPACK, ’ ’, databuf, BUF_SIZE,
     +                 N_RECORDS, FIELD2_NAME, height)
      status = vsfnpak(vdata_id, HDF_VSPACK, ’ ’, databuf, BUF_SIZE,
     +                 N_RECORDS, FIELD3_NAME, speed)
      status = vsfcpak(vdata_id, HDF_VSPACK, ’ ’, databuf, BUF_SIZE,
     +                 N_RECORDS, FIELD4_NAME, ident)
C
C     Write all the records of the packed data to the vdata.
C
      num_of_records = vsfwrit(vdata_id, databuf, N_RECORDS,
     +                         FULL_INTERLACE)
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.6. Reading from Vdatas

Reading from vdatas is more complicated than writing to vdatas, as it usually involves searching for a particular vdata, then searching *within* that vdata, before actually reading data. The process of reading from vdatas can be summarized as follows:

10. Identify the appropriate vdata in the file.
11. Obtain information about the vdata.
12. Read in the desired data.

Only Step 3 will be covered in this section assuming that the vdata of interest and its data information is known. Step 1 is covered in Section "*Searching for Vdatas in a File"* and Step 2 is covered in Section "*Obtaining Information about a Specific Vdata"*.

Step 3 can be expanded into the following:

1. Open the file.
2. Initialize the Vdata interface.
3. Initiate access to the vdata.
4. Optionally seek to the appropriate record.
5. Initialize the fields to be read.
6. Read the data.
7. If the fields have different data types, unpack the field data.

8. Terminate access to the vdata.
9. Terminate access to the Vdata interface.
10. Close the file.

The following sequence of function calls corresponds to the above steps:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);
            vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
            record_pos = VSseek(vdata_id, record_index);
            status = VSsetfields(vdata_id, fieldname_list);
            records_read = VSread(vdata_id, databuf, n_records, interlace_mode);
            status = VSfpack(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs);
            status = VSdetach(vdata_id);
            status = Vend(file_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            status = vfstart(file_id)
            vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
            record_pos = vsfseek(vdata_id, record_index)
            status = vsfsfld(vdata_id, fieldname_list)

            records_read = vsfrd(vdata_id, databuf, n_records, interlace_mode)
    OR      records_read = vsfrdc(vdata_id, databuf, n_records, interlace_mode)

            status = vsfcpak(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs)
    OR      status = vsfnpak(vdata_id, action, fields_in_buf, buf, buf_size,
                             n_records, fieldname_list, bufptrs)

            status = vsfdtch(vdata_id)
            status = vfend(file_id)
            status = hclose(file_id)
```

4.6.1. Initializing the Fields for Read Access: VSsetfields

VSsetfields is used to establish access to the fields to be read by the next read operation. The argument *fieldname_list* is a comma-separated string of the field names with no white space. The order the field names occur in *fieldname_list* is the order in which the fields will be read. For example, assume that a vdata contains fields named A, B, C, D, E, F in that order. The following declarations demonstrate how to use *fieldname_list* to read a single field, a collection of random fields, and all the fields in reverse order:

- Single field: *fieldname_list* = "`B`"
- Collection of fields: *fieldname_list* = "`A,E`"
- Reverse order: *fieldname_list* = "`F,E,D,C,B,A`"

VSsetfields returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSsetfields** are further defined in Table 4E.

4.6.2. Reading from the Current Vdata: VSread

VSread sequentially retrieves data from the records in a vdata. The parameter *databuf* is the buffer to store the retrieved data, *n_records* specifies the number of records to retrieve, and *inter-*

lace_mode specifies the interlace mode, `FULL_INTERLACE` (or `0`) or `NO_INTERLACE` (or `1`), to be used in the contents of *databuf*.

Prior to the first **VSread** call, **VSsetfields** must be called.

If a **VSread** call is successful, the data returned in *databuf* is formatted according to the interlace mode specified by the parameter *interlace_mode* and the data fields appear in the order specified in the last call to **VSsetfields** for that vdata.

By default, **VSread** reads from the first vdata record. To retrieve an arbitrary record from a vdata, use **VSseek** to specify the record position before calling **VSread**. **VSseek** is described in Section "*Resetting the Current Position within Vdatas: VSseek*".

The FORTRAN-77 version of **VSread** has three routines: **vsfrd** reads buffered numeric data, **vsfrdc** reads buffered character data and **vsfread** reads generic packed data.

VSread returns the total number of records read if successful and `FAIL` (or `-1`) otherwise. The parameters for **VSread** are further defined in Table 4I.

TABLE 4I **VSread Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSread [int32] **(vsfrd/vsfrdc/ vsfread)**	vdata_id	int32	integer	Vdata identifier
	databuf	uint8*	<valid numeric data type>(*) / character*(*) / integer	Buffer for the retrieved data
	n_records	int32	integer	Number of records to be retrieved
	interlace_mode	int32	integer	Interlace mode of the buffered data

VSsetfields and **VSread** may be called several times to read from the same vdata. However, note that **VSread** operations are sequential. Thus, in the following code segment, the first call to **VSread** returns ten "A" data values from the first ten elements in the vdata, while the second call to **VSread** returns ten "B" data values from the second ten elements (elements 10 to 19) in the vdata.

```
status = VSsetfields(vdata_id, "A");
records_read = VSread(vdata_id, bufferA, 10, interlace_mode);

status = VSsetfields(vdata_id, "B");
records_read = VSread(vdata_id, bufferB, 10, interlace_mode);
```

To read the first ten "B" data values, the access routine **VSseek** must be called to explicitly position the read pointer back to the position of the first record. The following code segment reads the first ten "A" and "B" values into two separate float arrays *bufferA* and *bufferB*.

```
status = VSsetfields(vdata_id, "A");
records_read = VSread(vdata_id, bufferA, 10, interlace_mode);

record_pos = VSseek(vdata_id, 0); /* seeks to first record */
status = VSsetfields(vdata_id, "B");
records_read = VSread(vdata_id, bufferB, 10, interlace_mode);
```

EXAMPLE 5.

Reading a Vdata of Homogeneous Type

This example illustrates the use of **VSfind/vsffnd** to locate a vdata given its name, **VSseek/vsfseek** to move the current position to a desired record, and **VSread/vsfrd** to read the data of several records. The function **VSfind** will be discussed in Section 4.7.3. The approach used in this example can only read data written by a program such as that in Example 3, i.e., without packing. Reading mixed data vdatas must use the approach illustrated in Example 6.

The program reads 5 records starting from the fourth record of the two fields "Position" and "Temperature" in the vdata "Solid Particle" from the file "General_Vdatas.hdf". After the program uses VSfind/vsffnd to obtain the reference number of the vdata, it uses **VSseek/vsfseek** to place the current position at the fourth record, then starts reading 5 records, and displays the data.

C:

```
#include "hdf.h"

#define  FILE_NAME       "General_Vdatas.hdf"
#define  VDATA_NAME      "Solid Particle"
#define  N_RECORDS       5    /* number of records the vdata contains */
#define  RECORD_INDEX    3    /* position where reading starts - 4th record */
#define  ORDER_1         3    /* order of first field to be read */
#define  ORDER_2         2    /* order of second field to be read */
#define  FIELDNAME_LIST  "Position,Temperature" /* only two fields are read */
#define  N_VALS_PER_REC  (ORDER_1 + ORDER_2)
                         /* number of values per record */

main( )
{
   /************************* Variable declaration *************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id, vdata_id,
         vdata_ref,     /* vdata's reference number */
         num_of_records, /* number of records actually written to the vdata */
         record_pos;    /* position of the current record */
   int16 i, rec_num;    /* current record number in the vdata */
   float32 databuf[N_RECORDS][N_VALS_PER_REC];   /* buffer for vdata values */

   /********************* End of variable declaration *********************/

   /*
   * Open the HDF file for reading.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);

   /*
   * Get the reference number of the vdata, whose name is specified in
   * VDATA_NAME, using VSfind, which will be discussed in Section 4.7.3.
   */
   vdata_ref = VSfind (file_id, VDATA_NAME);

   /*
   * Attach to the vdata for reading if it is found, otherwise
   * exit the program.
   */
   if (vdata_ref == 0) exit;
   vdata_id = VSattach (file_id, vdata_ref, "r");

   /*
   * Specify the fields that will be read.
   */
   status_n = VSsetfields (vdata_id, FIELDNAME_LIST);

   /*
   * Place the current point to the position specified in RECORD_INDEX.
   */
   record_pos = VSseek (vdata_id, RECORD_INDEX);

   /*
   * Read the next N_RECORDS records from the vdata and store the data
   * in the buffer databuf with fully interlaced mode.
   */
   num_of_records = VSread (vdata_id, (uint8 *)databuf, N_RECORDS,
                            FULL_INTERLACE);

   /*
   * Display the read data as many records as the number of records
   * returned by VSread.
   */
   printf ("\n       Particle Position        Temperature Range\n\n");
   for (rec_num = 0; rec_num < num_of_records; rec_num++)
   {
      printf ("   %6.2f, %6.2f, %6.2f        %6.2f, %6.2f\n",
        databuf[rec_num][0], databuf[rec_num][1], databuf[rec_num][2],
        databuf[rec_num][3], databuf[rec_num][4]);
   }

   /*
   * Terminate access to the vdata and to the VS interface, then close
   * the HDF file.
   */
   status_32 = VSdetach (vdata_id);
   status_n = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
  program read_from_vdata
       implicit none
C
C      Parameter declaration
C
       character*18 FILE_NAME
       character*14 VDATA_NAME
       character*20 FIELDNAME_LIST
       integer      N_RECORDS, RECORD_INDEX
       integer      ORDER_1, ORDER_2
       integer      N_VALS_PER_REC
C
       parameter (FILE_NAME       = 'General_Vdatas.hdf',
      +           VDATA_NAME      = 'Solid Particle',
      +           FIELDNAME_LIST = 'Position,Temperature')
       parameter (N_RECORDS  = 5,
      +           RECORD_INDEX = 3,
      +           ORDER_1    = 3,
      +           ORDER_2    = 2,
      +           N_VALS_PER_REC = ORDER_1 + ORDER_2 )
       integer DFACC_READ, FULL_INTERLACE
       parameter (DFACC_READ     = 1,
      +           FULL_INTERLACE = 0)
 C
 C     Function declaration
 C
       integer hopen, hclose
       integer vfstart, vsffnd, vsfatch, vsfsfld, vsfrd, vsfseek,
      +        vsfdtch, vfend

 C
 C**** Variable declaration *******************************************
 C
       integer status
       integer file_id, vdata_id
       integer vdata_ref, rec_num, num_of_records, rec_pos
       real    databuf(N_VALS_PER_REC, N_RECORDS)
       integer i
 C
 C**** End of variable declaration ************************************
 C
 C
 C     Open the HDF file for reading.
 C
       file_id = hopen(FILE_NAME, DFACC_READ, 0)
 C
 C     Initialize the VS interface.
 C
       status = vfstart(file_id)
 C
 C     Get the reference number of the vdata, whose name is specified in
 C     VDATA_NAME, using vsffnd, which will be discussed in Section 4.7.3.
 C
       vdata_ref = vsffnd(file_id, VDATA_NAME)
 C
 C     Attach to the vdata for reading if it is found,
 C     otherwise exit the program.
 C
       if (vdata_ref .eq. 0) stop
       vdata_id = vsfatch(file_id, vdata_ref, 'r')
 C
 C     Specify the fields that will be read.
 C
       status = vsfsfld(vdata_id, FIELDNAME_LIST)
 C
 C     Place the current point to the position specified in RECORD_INDEX.
 C
       rec_pos = vsfseek(vdata_id, RECORD_INDEX)
 C
 C     Read the next N_RECORDS from the vdata and store the data in the buffer
 C     databuf with fully interlace mode.
 C
       num_of_records = vsfrd(vdata_id, databuf, N_RECORDS,
      +                        FULL_INTERLACE)
 C
 C     Display the read data as many records as the number of records returned
 C     by vsfrd.
 C
       write(*,*) '  Particle Position     Temperature Range'
       write(*,*)
       do 10 rec_num = 1, num_of_records
          write(*,1000) (databuf(i, rec_num), i = 1, N_VALS_PER_REC)
```

```
10    continue
1000  format(1x,3(f6.2), 8x,2(f6.2))
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

EXAMPLE 6.

Reading a Multi-field and Mixed-type Vdata with Packing

This example illustrates the use of **VSread/vsfread** to read part of a mixed data vdata and **VSfpack/vsfnpak/vsfcpak** to unpack the data read.

The program reads the vdata "Mixed Data Vdata" that was written to the file "Packed_Vdata.hdf" by the program in Example 4. In Example 6, all values of the fields "Temp" and "Ident" are read. The program unpacks and displays all the values after reading is complete. Again, note that in C only one call to **VSread** and one call to **VSfpack** are made to read and unpack all `N_RECORDS` records. In Fortran, data is read with one call to **vsfread**, but each field is unpacked using separate calls to **vsfnpak** and **vsfcpak**

C:

```
#include "hdf.h"

#define  N_RECORDS       20      /* number of records to be read */
#define  N_FIELDS        2       /* number of fields to be read */
#define  FILE_NAME       "Packed_Vdata.hdf"
#define  VDATA_NAME      "Mixed Data Vdata"
#define  FIELDNAME_LIST  "Temp,Ident"

/* number of bytes of the data to be read */
#define  BUFFER_SIZE     ( sizeof(float32) + sizeof(char)) * N_RECORDS

main ()
{
   /************************* Variable declaration **************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id, vdata_id,
         num_of_records,        /* number of records actually read */
         vdata_ref,             /* reference number of the vdata to be read */
         buffer_size;           /* number of bytes the vdata can hold       */
   float32 itemp[N_RECORDS];    /* buffer to hold values of first field     */
   char  idents[N_RECORDS];     /* buffer to hold values of fourth field    */
   uint8 databuf[BUFFER_SIZE];  /* buffer to hold read data, still packed   */
   VOIDP fldbufptrs[N_FIELDS];/*pointers to be pointing to the field buffers*/
   int   i;

   /********************** End of variable declaration *********************/

   /*
   * Open the HDF file for reading.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
```

```
    * Initialize the VS interface.
    */
    status_n = Vstart (file_id);

    /*
    * Get the reference number of the vdata, whose name is specified in
    * VDATA_NAME, using VSfind, which will be discussed in Section 4.7.3.
    */
    vdata_ref = VSfind (file_id, VDATA_NAME);

    /*
    * Attach to the vdata for reading.
    */
    vdata_id = VSattach (file_id, vdata_ref, "r");

    /*
    * Specify the fields that will be read.
    */
    status_n = VSsetfields(vdata_id, FIELDNAME_LIST);

    /*
    * Read N_RECORDS records of the vdata and store the values into the
    * buffer databuf.
    */
    num_of_records = VSread (vdata_id, (uint8 *)databuf, N_RECORDS,
                             FULL_INTERLACE);

    /*
    * Build an array of pointers each of which points to an array that
    * will hold all values of a field after being unpacked.
    */
    fldbufptrs[0] = &itemp[0];
    fldbufptrs[1] = &idents[0];

    /*
    * Unpack the data from the buffer databuf and store the values into the
    * appropriate field buffers pointed to by the set of pointers fldbufptrs.
    * Note that the second parameter is _HDF_VSUNPACK for unpacking and the
    * number of records is the one returned by VSread.
    */
    status_n = VSfpack (vdata_id, _HDF_VSUNPACK, FIELDNAME_LIST, (VOIDP)databuf,
                BUFFER_SIZE, num_of_records, NULL, (VOIDP)fldbufptrs);

    /*
    * Display the read data being stored in the field buffers.
    */
    printf ("\n     Temp      Ident\n");
    for (i=0; i < num_of_records; i++)
        printf ("   %6.2f        %c\n", itemp[i], idents[i]);

    /*
    * Terminate access to the vdata and the VS interface, then close
    * the HDF file.
    */
    status_32 = VSdetach (vdata_id);
    status_n = Vend (file_id);
    status_32 = Hclose (file_id);
}
```

FORTRAN:

```
 program read_mixed_vdata
      implicit none
C
C     Parameter declaration
C
      character*16 FILE_NAME
      character*16 VDATA_NAME
      character*4  FIELD1_NAME
      character*5  FIELD2_NAME
      character*10 FIELDNAME_LIST
      integer      N_RECORDS, N_FIELDS
      integer      BUFFER_SIZE
C
      parameter (FILE_NAME       = 'Packed_Vdata.hdf',
     +           VDATA_NAME      = 'Mixed Data Vdata',
     +           FIELD1_NAME     = 'Temp',
     +           FIELD2_NAME     = 'Ident',
     +           FIELDNAME_LIST = 'Temp,Ident')
      parameter (N_RECORDS   = 20,
     +           N_FIELDS    = 2,
     +           BUFFER_SIZE = (4 + 1)*N_RECORDS)

      integer DFACC_READ, DFNT_FLOAT32, DFNT_CHAR8,
     +        FULL_INTERLACE, HDF_VSUNPACK
      parameter (DFACC_READ       = 1,
     +           DFNT_FLOAT32     = 5,
     +           DFNT_CHAR8       = 4,
     +           FULL_INTERLACE   = 0,
     +           HDF_VSUNPACK     = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vsfatch, vsffnd, vsfsfld,
     +        vsfnpak, vsfcpak, vsfread, vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer   status
      integer   file_id, vdata_id
      integer   vdata_ref, num_of_records
      real      temp(N_RECORDS)
      character ident(N_RECORDS)
      integer   i
C
C     Buffer for read packed data should be big enough to hold N_RECORDS.
C
      integer   databuf(BUFFER_SIZE/4 + 1)
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
```

```
C     Get the reference number of the vdata, whose name is specified in
C     VDATA_NAME, using vsffnd, which will be discussed in Section 4.7.3.
C
      vdata_ref = vsffnd(file_id, VDATA_NAME)
C
C     Attach to the vdata for reading if it is found,
C     otherwise exit the program.
C
      if (vdata_ref .eq. 0) stop
      vdata_id = vsfatch(file_id, vdata_ref, 'r')
C
C     Specify the fields that will be read.
C
      status = vsfsfld(vdata_id, FIELDNAME_LIST)

C
C    Read N_RECORDS records of the vdata and store the values into the databuf.
C
      num_of_records = vsfread(vdata_id, databuf, N_RECORDS,
     +                         FULL_INTERLACE)
C
C     Unpack N_RECORDS from databuf into temp and ident arrays.
C     In Fortran, each field is unpacked using separate calls to
C     vsfnpak or vsfcpak.
C
      status = vsfnpak(vdata_id, HDF_VSUNPACK, FIELDNAME_LIST, databuf,
     +                 BUFFER_SIZE, num_of_records, FIELD1_NAME, temp)
      status = vsfcpak(vdata_id, HDF_VSUNPACK, FIELDNAME_LIST, databuf,
     +                 BUFFER_SIZE, num_of_records, FIELD2_NAME, ident)
C
C     Display the read data being stored in the field databufs.
C
      write (*,*) '    Temp  Ident'
      do 10 i = 1, num_of_records
         write(*,1000) temp(i), ident(i)
10    continue
1000  format (3x,F6.2, 4x, a)
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.7. Searching for Vdatas in a File

There are several HDF library routines that perform searches for a specific vdata in a file. In this section, we introduce these routines; methods for obtaining information about the members of a given vdata are described in the following section.

4.7.1. Finding All Vdatas that are Not Members of a Vgroup: VSlone

A ***lone vdata*** is one that is not a member of a vgroup. ***Vgroups*** are HDF objects that contain sets of HDF objects, including vgroups. Vgroups are described in Chapter 5, *Vgroups (V API)*.

VSlone searches an HDF file and retrieves the reference numbers of lone vdatas in the file. The syntax of **VSlone** is as follows:

```
C:          num_of_lone_vdatas = VSlone(file_id, ref_array, maxsize);

FORTRAN:    num_of_lone_vdatas = vsflone(file_id, ref_array, maxsize)
```

The parameter *ref_array* is an array allocated to hold the retrieved reference numbers of lone vdatas and the argument *maxsize* specifies the maximum size of *ref_array*. At most, *maxsize* reference numbers will be returned in *ref_array*.

The space that should be allocated for *ref_array* is dependent upon on how many lone vdatas are expected in the file. A size of `MAX_FIELD_SIZE` (or `65535`) integers is adequate to handle any case. To use dynamic memory instead of allocating such a large array, first call **VSlone** with *maxsize* set to a small value like 0 or 1, then use the returned value to allocate memory for *ref_array* to be passed to a subsequent call to **VSlone**.

VSlone returns the number of lone vdatas or `FAIL` (or `-1`). The parameters for **VSlone** are listed in Table 4J.

4.7.2. Sequentially Searching for a Vdata: VSgetid

VSgetid sequentially searches through an HDF file to obtain the vdata immediately following the vdata specified by the reference number in the parameter *vdata_ref*. The syntax of **VSgetid** is as follows:

```
C:          ref_num = VSgetid(file_id, vdata_ref);

FORTRAN:    ref_num = vsfgid(file_id, vdata_ref)
```

To obtain the reference number of the first vdata in the file, the user must set the parameter *vdata_ref* to -1. Thus, **VSgetid** can be repeatedly called, with the initial value of *vdata_ref* set to -1 so that the routine will sequentially return the reference number of each vdata in the file, starting from the first vdata. After the last vdata is reached, subsequent calls to **VSgetid** will return `FAIL` (or `-1`).

VSgetid returns a vdata reference number or `FAIL` (or `-1`). The parameters for **VSgetid** are listed in Table 4J.

4.7.3. Determining a Reference Number from a Vdata Name: VSfind

VSfind searches an HDF file for a vdata with the specified name and returns the vdata reference number. The syntax of **VSfind** is as follows:

```
C:          ref_num = VSfind(file_id, vdata_name);

FORTRAN:    ref_num = vsffnd(file_id, vdata_name)
```

The parameter *vdata_name* is the search key. Although there may be several identically named vdatas in the file, **VSfind** will only return the reference number of the first vdata in the file with the specified name.

VSfind returns either the vdata reference number if the named vdata is found or `0` otherwise. The parameters for **VSfind** are listed in Table 4J.

4.7.4. Searching for a Vdata by Field Name: VSfexist

VSfexist queries a vdata for a set of specified field names and is often useful for locating vdatas containing particular field names. The syntax of the **VSfexist** function is as follows:

```
C:          status = VSfexist(vdata_id, fieldname_list);

FORTRAN:    status = vsfex(vdata_id, fieldname_list)
```

The parameter *fieldname_list* is a string of comma-separated field names containing no white space, for example, “`PX,PY,PZ`”.

VSfexist returns `SUCCEED` (or `0`) if all of the fields specified in the parameter *fieldname_list* are found and `FAIL` (or `-1`) otherwise. The parameters for **VSfexist** are listed in Table 4J.

TABLE 4J **VSlone, VSgetid, VSfind, and VSfexist Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSlone [int32] **(vsflone)**	file_id	int32	integer	File identifier
	ref_array	int32 []	integer (*)	Buffer for a list of lone vdata reference numbers
	maxsize	int32	integer	Maximum number of reference numbers to be buffered
VSgetid [int32] **(vsfgid)**	file_id	int32	integer	File identifier
	vdata_ref	int32	integer	Reference number of the vdata preceding the vdata
VSfind [int32] **(vsffnd)**	file_id	int32	integer	File identifier
	vdata_name	char *	character*(*)	Name of the vdata to find
VSfexist [intn] **(vsfex)**	vdata_id	int32	integer	Vdata identifier
	fieldname_list	char *	character*(*)	Names of the fields to be queried

EXAMPLE 7. **Locating a Vdata Containing Specified Field Names**

This example illustrates the use of **VSgetid/vsfgid** to obtain the reference number of each vdata in an HDF file and the use of **VSfexist/vsfex** to determine whether a vdata contains specific fields.

In this example, the program searches the HDF file "General_Vdatas.hdf" to locate the first vdata containing the fields "Position" and "Temperature". The HDF file is an output of the program in Example 3.

C:

```
#include "hdf.h"

#define  FILE_NAME         "General_Vdatas.hdf"
#define  SEARCHED_FIELDS   "Position,Temperature"

main( )
{
   /************************* Variable declaration **************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id, vdata_id, vdata_ref,
         index = 0;     /* index of the vdata in the file - manually kept   */
```

```
int8  found_fields;  /* TRUE if the specified fields exist in the vdata  */

/********************** End of variable declaration **********************/

/*
* Open the HDF file for reading.
*/
file_id = Hopen (FILE_NAME, DFACC_READ, 0);

/*
* Initialize the VS interface.
*/
status_n = Vstart (file_id);

/*
* Set the reference number to -1 to start the search from
* the beginning of file.
*/
vdata_ref = -1;

/*
* Assume that the specified fields are not found in the current vdata.
*/
found_fields = FALSE;

/*
* Use VSgetid to obtain each vdata by its reference number then
* attach to the vdata and search for the fields.  The loop
* terminates when the last vdata is reached or when a vdata which
* contains the fields listed in SEARCHED_FIELDS is found.
*/
while ((vdata_ref = VSgetid (file_id, vdata_ref)) != FAIL)
{
   vdata_id = VSattach (file_id, vdata_ref, "r");
   if ((status_n = VSfexist (vdata_id, SEARCHED_FIELDS)) != FAIL)
   {
      found_fields = TRUE;
      break;
   }

   /*
   * Detach from the current vdata before continuing searching.
   */
   status_32 = VSdetach (vdata_id);

   index++;/* advance the index by 1 for the next vdata */
}

/*
* Print the index of the vdata containing the fields or a "not found"
* message if no such vdata is found.  Also detach from the vdata found.
*/
if (!found_fields)
   printf ("Fields Position and Temperature were not found.\n");
else
{
   printf
  ("Fields Position and Temperature found in the vdata at position %d\n",
    index);
   status_32 = VSdetach (vdata_id);
}

/*
```

```
   * Terminate access to the VS interface and close the HDF file.
   */
   status_n = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
 program locate_vdata
      implicit none
C
C     Parameter declaration
C
      character*18 FILE_NAME
      character*20 SEARCHED_FIELDS
C
      parameter (FILE_NAME       = 'General_Vdatas.hdf',
     +           SEARCHED_FIELDS = 'Position,Temperature')
      integer DFACC_READ
      parameter (DFACC_READ = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vsfatch, vsfgid, vsfex, vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, vdata_id, vdata_ref
      integer index
      logical found_fields
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
      index = 0
C
C     Set the reference number to -1 to start the search from the beginning
C     of the file.
C
      vdata_ref = -1
C
C     Assume that the specified fields are not found in the current vdata.
C
      found_fields = .FALSE.
10    continue
C
C     Use vsfgid to obtain each vdata by its reference number then
C     attach to the vdata and search for the fields. The loop terminates
C     when the last vdata is reached or when a vdata which contains the
C     fields listed in SEARCHED_FIELDS is found.
C
      vdata_ref = vsfgid(file_id, vdata_ref)
      if (vdata_ref .eq. -1) goto 100
      vdata_id = vsfatch(file_id, vdata_ref, 'r')
      status = vsfex(vdata_id, SEARCHED_FIELDS)
      if (status .ne. -1) then
          found_fields = .TRUE.
          goto 100
      endif
      status = vsfdtch(vdata_id)
      index = index + 1
      goto 10
100   continue
C
C     Print the index of the vdata containing the fields or a 'not found'
C     message if no such vdata is found. Also detach from the vdata found.
C
      if(.NOT.found_fields) then
         write(*,*) 'Fields Positions and Temperature were not found'
      else
         write(*,*)
     +   'Fields Positions and Temperature were found in the vdata',
     +   ' at position ', index
C
C        Terminate access to the vdata
C
         status = vsfdtch(vdata_id)
      endif
C
C     Terminate access to the VS interface and close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.7.5. Retrieving Vdatas in a File or in a Vgroup: VSgetvdatas

VSgetvdatas retrieves a list containing reference numbers of vdatas in a file or in a vgroup, which is identified by the parameter *id*. The syntax of **VSgetvdatas** is as follows:

```
C:          status = VSgetvdatas(id, start_vd, vd_count, refarray);

FORTRAN:    status = vsfgvdatas(id, start_vd, vd_count, refarray)
```

The library commonly use vgroups or vdatas to store HDF objects. For example, a vgroup is used to represent an SDS and a vdata for an attribute. **VSgetvdatas** retrieves only the vdatas that were previously created by user applications, not those that were created by the library internally. They are referred to as user-created vdatas, for brevity.

When *id* is a vgroup identifier, only the immediate sub-vdatas will be retrieved; that is, the sub-vgroups will not be traversed.

The parameter *vd_count* specifies the number of values that the *refarray* list can hold and can be any positive number smaller than `MAX_REF` (`65535`). If *vd_count* is larger than the actual number of user-created vdatas, then only the actual number of user-created vdatas will be retrieved.

The retrieval starts at the vdatas number *start_vd* going forward in the order which the vdatas were created. For example, if there are 100 vdatas that can be retrieved, specifying *start_vd* as 90 and *vd_count* as 10 will retrieve the last ten vdatas. The value for *start_vd* must be non-negative and smaller than the number of user-created vdatas, which can be obtained by invoking **VSgetvdatas** passing in `NULL` for the array *refarray*. This number of user-created vdatas will also allow applications to sufficiently allocate space for *refarray*.

- When *start_vd* is 0, the retrieval will start at the beginning of the file or the first sub-vdata of the specified vgroup.
- When *start_vd* is smaller than the number of user-created vdatas in the file or the specified vgroup, **VSgetvdatas** will start retrieving vdatas from the vdata number *start_vd*.
- When *start_vd* equals or is greater than the number of user-created vdatas in the file or the vgroup, **VSgetvdatas** will return `FAIL` (or `-1`).

Following are some examples of using **VSgetvdatas** to get the reference numbers of vdatas in a file, assuming that the file has been opened for reading successfully:

```
C:        /* Call VSgetvdatas the first time to get the number of vdatas in
             the file to allocate ref_array */
          n_vds = VSgetvdatas(file_id, 0, 0, NULL);

          /* Allocate space to retrieve reference numbers of n_vds vdatas */
          ref_array = (uint16 *)HDmalloc(sizeof(uint16)*n_vds);

          /* To get all the vdatas in the file: */
          n_vds = VSgetvdatas(file_id, 0, n_vds, ref_array);

          /* Assuming n_vds=100, to get the first 10 vdatas in the file: */
          n_vds = VSgetvdatas(file_id, 0, 10, ref_array);

          /* Assuming n_vds=100, to get the last 10 vdatas in the file: */
          n_vds = VSgetvdatas(file_id, 90, 10, ref_array);
```

Following are some examples of using **VSgetvdatas** to get the reference numbers of vdatas in a parent vgroup:

```
C:        vdata_id = Vattach(file_id, vdata_ref, "r");
          /* Call VSgetvdatas the first time to get the number of vdatas in the
             parent vgroup to allocate ref_array */
          n_vds = VSgetvdatas(vgroup_id, 0, 0, NULL);

          /* Allocate space to retrieve reference numbers of n_vds vdatas */
          ref_array = (uint16 *)HDmalloc(sizeof(uint16)*n_vds);

          /* Get all the vdatas in the parent vgroup */
          n_vds = VSgetvdatas(vgroup_id, 0, n_vds, ref_array);

          /* Close the vgroup */
          status = Vdetach(vgroup_id);
```

Note that, in the FORTRAN-77 version, if *vd_count* is `-1` then the function will return the number of user-created vdatas and disregard *refarray*; equivalent to passing `NULL` for *refarray* in the C version.

VSgetvdatas returns the number of user-created vdatas retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of this routine are further defined in Table 4K.

4.7.6. Determining Internal Vdata: VSisinternal

The HDF library commonly uses vgroups and vdatas to store metadata or data for the library's own use. For examples, vgroups are used to represent SDS or raster images, and vdatas are used to store attributes or dimensions. Typically, a user is only interested in vgroups/vdatas that were created by user applications, not by the library internally. **VSisinternal** allows an application to find out if a vdata is internally created.

The syntax of **VSisinternal** is as follows:

```
C:          is_internal = VSisinternal(vdata_id);

FORTRAN:    Currently unavailable
```

VSisinternal checks the class name of the given vdata against the list `HDF_INTERNAL_VDS` to determine whether the vdata was previously created by the library instead of by a user application. The names in `HDF_INTERNAL_VDS` are included:

```
DIM_VALS ("DimVal0.0")
DIM_VALS01 ("DimVal0.1")
_HDF_ATTRIBUTE ("Attr0.0")
HDF_SDSVAR ("SDSVar")
HDF_CRDVAR ("CoordVar")
_HDF_CHK_TBL_CLASS ("_HDF_CHK_TBL_")
RIGATTRCLASS("RIATTR0.0C")
```

VSisinternal returns `TRUE` (`1`) if the inquired vdata is one that was internally created by the library, `FALSE` (`0`) otherwise, and `FAIL` (`-1`) if failure occurs. The parameters of this routine are further defined in Table 4K.

4.7.7. Retrieving Vdatas in a File or in a Vgroup: VSofclass

VSofclass retrieves reference numbers of vdatas of the specified class in a file or in a vgroup. The syntax of **VSofclass** is as follows:

```
C:          status = VSofclass(id, vd_class, start_vd, vd_count, refarray);

FORTRAN:    Unavailable
```

When *id* is a vgroup identifier, only the immediate sub-vdatas will be checked; that is, the sub-vgroups will not be traversed. The parameter *vd_count* specifies the number of values that the *refarray* list can hold and can be any positive number smaller than `MAX_REF` (`65535`). If *vd_count* is larger than the actual number of vdatas that has the specified class, then only the actual number of such vdatas will be retrieved.

The parameter *start_vd* is the index of the vdatas having the specified class, *vd_class*. The retrieval starts at the vdata number *start_vd* going forward in the order which the vdatas were created. The combination of *start_vd* and *vd_count* provide flexibility in the retrieval. For example, if there are 100 vdatas that can be retrieved, specifying *start_vd* as 90 and *vd_count* as 10 will retrieve the last ten such vdatas. The value for *start_vd* must be non-negative and smaller than the number of vdatas having the specified class. This number can be obtained by invoking **VSofclass** passing in `NULL` for the array *refarray* and will also allow applications to sufficiently allocate space for *refarray.*

When *start_vd* is 0, the retrieval will start at the beginning of the file or the first sub-vdata of the specified vgroup.

When *start_vd* is smaller than the number of vdatas having the specified class name, **VSofclass** will start retrieving from the vdata number *start_vd*.

When *start_vd* equals or is greater than the number of vdatas having the specified class name, **VSofclass** will return `FAIL` (or `-1`).

VSofclass returns the number of vdatas retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of this routine are further defined in Table 4K.

TABLE 4K **VSgetvdatas, VSisinternal, and VSofclass Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSgetvdatas [intn] **(vsfgvdatas)**	id	int32	integer	File or vgroup identifier
	start_vd	uintn	integer	Vdata index to start retrieving at
	vd_count	uintn	integer	Number of vdatas to be retrieved
	refarray	uint16 *	integer (*)	Array to hold reference numbers of retrieved vdatas
VSisinternal [intn] **(unavailable)**	vdata_id	int32	N/A	Vdata identifier
VSofclass [intn] **(unavailable)**	id	int32	N/A	File or vgroup identifier
	vd_class	const char *	N/A	Class name of vdatas to be retrieved
	start_vd	uintn	N/A	Vdata index to start retrieving at
	vd_count	uintn	N/A	Number of vdatas to be retrieved
	*refarray	uint16	N/A	Array to hold reference numbers of retrieved vdatas

4.8. Vdata Attributes

HDF version 4.1r1 and later include the ability to assign attributes to a vdata and/or a vdata field. The concept of attributes is fully explained in Chapter 3, *Scientific Data Sets (SD API)*. To review briefly: an attribute has a name, a data type, a number of attribute values, and the attribute values themselves. All attribute values must be of the same data type. For example, an integer cannot be added to an attribute value consisting of ten characters, or a character value cannot be included in an attribute value consisting of 2 32-bit integers.

Any number of attributes can be assigned to either a vdata or any single field in a vdata. However, each attribute name should be unique within its scope. In other words, the name of a field's attribute must be unique among all attributes that belong to that same field, and the name of a vdata's attribute must be unique among all attributes assigned to the same vdata.

The following subsections describe routines that retrieve various information about vdata and vdata field attributes. Those routines that access field attributes require the field index as a parameter (*field_index*.)

4.8.1. Querying the Index of a Vdata Field Given the Field Name: VSfindex

VSfindex retrieves the index of a field given its name, *field_name*, and stores the value in the parameter *field_index*. The syntax of **VSfindex** is as follows:

```
C:          status = VSfindex(vdata_id, field_name, &field_index);

FORTRAN:    status = vsffidx(vdata_id, field_name, field_index)
```

The parameter *field_index* is the index number that uniquely identifies the location of the field within the vdata. Field index numbers are assigned in increasing order and are zero-based: for example, a *field_index* value of 4 would refer to the fifth field in the vdata.

VSfindex returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The parameters for **VSfindex** are further defined in Table 4L.

TABLE 4L **VSfindex Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSfindex [intn] **(vsffidx)**	vdata_id	int32	integer	Vdata identifier
	field_name	char *	character*(*)	Name of the vdata field
	field_index	int32 *	integer	Index of the vdata field

4.8.2. Setting the Attribute of a Vdata or Vdata Field: VSsetattr

VSsetattr attaches an attribute to a vdata or a vdata field. The syntax of **VSsetattr** is as follows:

```
C:          status = VSsetattr(vdata_id, field_index, attr_name, data_type,
                               n_values, values);

FORTRAN:    status = vsfsnat(vdata_id, field_index, attr_name, data_type, n_val-
                               ues, values)

    OR      status = vsfscat(vdata_id, field_index, attr_name, data_type, n_val-
                               ues, values)
```

If the attribute has already been attached, the new attribute values will replace the current values, provided the data type and the number of attribute values (*n_values*) have not been changed. If either of these have been changed, **VSsetattr** will return `FAIL` (or `-1`).

Set the parameter *field_index* to `_HDF_VDATA` (or `-1`) to set an attribute for a vdata or to a valid field index to set attribute for a vdata field. A valid field index is a zero-based integer value representing the ordinal location of a field within the vdata.

The parameter `attr_name` specifies the name of the attribute to be set and can contain `VSNAMELENMAX` (or `64`) characters. The parameter *data_type* specifies the data type of the attribute values. Data types supported by HDF are listed in Table 2G. The parameter *values* contains attribute values to be written.

The FORTRAN-77 version of **VSsetattr** has two routines: **vsfsnat** sets numeric attribute data and **vsfscat** sets character attribute data.

VSsetattr returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The parameters for **VSsetattr** are described in Table 4M.

4.8.3. Querying the Values of a Vdata or Vdata Field Attribute: VSgetattr

VSgetattr returns all of the values of the specified attribute of the specified vdata field or vdata. The syntax of **VSgetattr** is as follows:

```
C:          status = VSgetattr(vdata_id, field_index, attr_index, values);

FORTRAN:    status = vsfgnat(vdata_id, field_index, attr_index, values)

    OR      status = vsfgcat(vdata_id, field_index, attr_index, values)
```

Set the parameter *field_index* to `_HDF_VDATA` (or `-1`) to retrieve the values of the attribute attached to the vdata identified by the parameter *vdata_id*. Set *field_index* to a zero-based integer value to retrieve the values of an attribute attached to a vdata field; the value of *field_index* will be used as the index of the vdata field. In both cases, the values returned will be those of the attribute located at the position specified by the parameter *attr_index*, the zero-based index of the target attribute.

The parameter *values* must be sufficiently allocated to hold the retrieved attribute values. Use **VSattrinfo** to obtain information about the attribute values for appropriate memory allocation.

The FORTRAN-77 versions of **VSgetattr** has two routines: **vsfgnat** gets numeric attribute data and **vsfgcat** gets character attribute data.

VSgetattr returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The parameters for **VSgetattr** are described in Table 4M.

TABLE 4M **VSsetattr and VSgetattr Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSsetattr [intn] **(vsfsnat/vsfscat)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	`_HDF_VDATA` or index of the field
	attr_name	char *	character*(*)	Name of the attribute
	data_type	int32	integer	Data type of the attribute
	n_values	int32	integer	Number of values the attribute contains
	values	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer containing the attribute values
VSgetattr [intn] **(vsfgnat/vsfgcat)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	`_HDF_VDATA` or index of the field
	attr_index	intn	integer	Index of the attribute
	values	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer containing attribute values

4.8.4. Querying the Total Number of Vdata and Vdata Field Attributes: VSnattrs

VSnattrs returns the total number of attributes of the specified vdata *and* the fields contained in the vdata. This is different from the **VSfnattrs** routine, which returns the number of attributes of the specified vdata *or* a specified field contained in the specified vdata. The syntax of **VSnattrs** is as follows:

```
C:          num_of_attrs = VSnattrs(vdata_id);

FORTRAN:    num_of_attrs = vsfnats(vdata_id)
```

VSnattrs returns the total number of attributes assigned to the vdata and its fields when successful, and `FAIL` (or `-1`) otherwise. The parameters for **VSnattrs** are described in Table 4N.

4.8.5. Querying the Number of Attributes of a Vdata or a Vdata Field: VSfnattrs

VSfnattrs returns the number of attributes attached to the vdata field specified by the parameter *field_index or* the number of attributes attached to the vdata identified by *vdata_id*. This is different from the routine **VSnattrs**, which returns the total number of attributes of the specified vdata *and* the fields contained in it. The syntax of **VSfnattrs** is as follows:

```
C:          num_of_attrs = VSfnattrs(vdata_id, field_index);

FORTRAN:    num_of_attrs = vsffnas(vdata_id, field_index)
```

If *field_index* is set to a zero-based integer value, it will be used as the index of the vdata field, and the number of attributes attached to that field will be returned. If *field_index* is set to `_HDF_VDATA` (or `-1`), the number of attributes attached to the vdata specified by *vdata_id* will be returned.

VSfnattrs returns the number of attributes assigned to the specified vdata or to the specified vdata field when successful, and `FAIL` (or `-1`) otherwise. The parameters for **VSfnattrs** are described in Table 4N.

TABLE 4N **VSnattrs and VSfnattrs Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSnattrs [intn] **(vsfnats)**	vdata_id	int32	integer	Vdata identifier
VSfnattrs [int32] **(vsffnas)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	`_HDF_VDATA` or index of the field

4.8.6. Retrieving the Index of a Vdata or Vdata Field Attribute Given the Attribute Name: VSfindattr

VSfindattr returns the index of an attribute with the specified name. The attribute must be attached to either a vdata or one of its fields. The syntax of **VSfindattrs** is as follows:

```
C:          attr_index = VSfindattr(vdata_id, field_index, attr_name);

FORTRAN:    attr_index = vsffdat(vdata_id, field_index, attr_name)
```

If *field_index* is set to `_HDF_VDATA` (or `-1`), the index of the attribute identified by the parameter *attr_name* and attached to the vdata specified by *vdata_id* will be returned.

If the parameter *field_index* is set to a zero-based integer value, the value will be used as the index of the vdata field. Then, the index of the attribute named by the parameter *attr_name* and attached to the field specified by the parameter *field_index* will be returned.

VSfindattr returns an attribute index if successful, and `FAIL` (or `-1`) otherwise. The parameters for **VSfindattr** are described in Table 4O.

4.8.7. Querying Information on a Vdata or Vdata Field Attribute: VSattrinfo

VSattrinfo returns the name, data type, number of values, and the size of the values of the specified attribute of the specified vdata field or vdata. The syntax of **VSattrinfo** is as follows:

```
C:          status = VSattrinfo(vdata_id, field_index, attr_index, attr_name,
                                &data_type, &n_values, &size);
```

```
FORTRAN:  status = vsfainf(vdata_id, field_index, attr_index, attr_name,
                           data_type, n_values, size)
```

In C, the parameters *attr_name*, *data_type*, *n_values*, and *size* can be set to `NULL`, if the information returned by these parameters are not needed.

The parameter *field_index* is the same as the parameter *field_index* in **VSsetattr**; it can be set either to a nonnegative integer to specify the field or to `_HDF_VDATA` (or `-1`) to specify the vdata referred to by *vdata_id*.

VSattrinfo returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The parameters for **VSattrinfo** are described in Table 4O.

4.8.8. Determining whether a Vdata Is an Attribute: VSisattr

The HDF library stores vdata attributes and vdata field attributes as vdatas. HDF therefore provides the routine **VSisattr** to determine whether a particular vdata contains attribute data. The syntax of **VSisattr** is as follows:

```
C:        status = VSisattr(vdata_id);

FORTRAN:  status = vsfisat(vdata_id)
```

VSisattr returns `TRUE` (or `1`) if the vdata contains an attribute data and `FALSE` (or `0`) otherwise. The parameters for **VSisattr** are described in Table 4O.

TABLE 4O **VSfindattr, VSattrinfo, and VSisattr Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSfindattr [intn] **(vsffdat)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	`_HDF_VDATA` or index of the field
	attr_name	char *	character*(*)	Name of the attribute
VSattrinfo [intn] **(vsfainf)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Index of the field
	attr_index	intn	integer	Index of the attribute
	attr_name	char *	character*(*)	Returned name of the attribute
	data_type	int32 *	integer	Returned data type of the attribute
	n_values	int32 *	integer	Number of values of the attribute
	size	int32 *	integer	Size, in bytes, of the values of the attribute
VSisattr [intn] **(vsfisat)**	vdata_id	int32	integer	Vdata identifier

EXAMPLE 8. **Operations on Field and Vdata Attributes**

This example illustrates the use of **VSsetattr/vsfscat/vsfsnat** to attach an attribute to a vdata and to a field in a vdata, the use of **VSattrinfo/vsfainf** to get information about a field attribute and a vdata attribute, and the use of **VSgetattr/vsfgcat/vsfgnat** to get the values of an attribute of a vdata and the values of an attribute of a field in a vdata. The example also shows the use of **VSf-**

nattrs/vsffnas to obtain the number of attributes attached to a field of a vdata and the use of **VSnattrs/vsfnats** to obtain the total number of attributes attached to both a vdata and its fields.

In this example, the program finds the vdata, named "Solid Particle", in the HDF file "General_Vdatas.hdf" produced by Example 3. It then obtains the index of the field, named "Mass", in the vdata. An attribute named "Site Ident" is attached to the vdata to contain the identification of the experiment sites. Another attribute named "Scales" is attached to the field for its scale values. The vdata attribute has 3 character values and the field attribute has 4 integer values.

C:

```
#include "hdf.h"

#define  FILE_NAME        "General_Vdatas.hdf"
#define  VDATA_NAME       "Solid Particle"
#define  FIELD_NAME       "Mass"
#define  VATTR_NAME       "Site Ident"      /* name of the vdata attribute  */
#define  FATTR_NAME       "Scales"          /* name of the field attribute  */
#define  VATTR_N_VALUES   3      /* number of values in the vdata attribute */
#define  FATTR_N_VALUES   4      /* number of values in the field attribute */

main( )
{
   /************************* Variable declaration *************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id, vdata_ref, vdata_id,
         field_index,   /* index of a field within the vdata */
         n_vdattrs,     /* number of vdata attributes */
         n_fldattrs,    /* number of field attributes */
         vdata_type,    /* to hold the type of vdata's attribute */
         vdata_n_values,/* to hold the number of vdata's attribute values   */
         vdata_size,    /* to hold the size of vdata's attribute values     */
         field_type,    /* to hold the type of field's attribute            */
         field_n_values,/* to hold the number of field's attribute values   */
         field_size;    /* to hold the size of field's attribute values     */
   char  vd_attr[VATTR_N_VALUES] = {'A', 'B', 'C'};/* vdata attribute values*/
   int32 fld_attr[FATTR_N_VALUES] = {2, 4, 6, 8};  /* field attribute values*/
   char  vattr_buf[VATTR_N_VALUES];     /* to hold vdata attribute's values */
   int32 fattr_buf[FATTR_N_VALUES];     /* to hold field attribute's values */
   char  vattr_name[30],                /* name of vdata attribute */
         fattr_name[30];                /* name of field attribute */

   /********************** End of variable declaration *********************/

   /*
   * Open the HDF file for writing.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);

   /*
   * Get the reference number of the vdata named VDATA_NAME.
   */
   vdata_ref = VSfind (file_id, VDATA_NAME);

   /*
   * Attach to the vdata for writing.
```

```
*/
vdata_id = VSattach (file_id, vdata_ref, "w");

/*
* Attach an attribute to the vdata, i.e., indicated by the second parameter.
*/
status_n = VSsetattr (vdata_id, _HDF_VDATA, VATTR_NAME, DFNT_CHAR,
                                                VATTR_N_VALUES, vd_attr);

/*
* Get the index of the field FIELD_NAME within the vdata.
*/
status_n = VSfindex (vdata_id, FIELD_NAME, &field_index);

/*
* Attach an attribute to the field field_index.
*/
status_n = VSsetattr (vdata_id, field_index, FATTR_NAME, DFNT_INT32,
                                                FATTR_N_VALUES, fld_attr);

/*
* Get the number of attributes attached to the vdata’s first
* field - should be 0.
*/
n_fldattrs = VSfnattrs (vdata_id, 0);
printf ( "Number of attributes of the first field of the vdata: %d\n",
          n_fldattrs);

/*
* Get the number of attributes attached to the field specified by
* field_index - should be 1.
*/
n_fldattrs = VSfnattrs (vdata_id, field_index);
printf ( "Number of attributes of field %s: %d\n", FIELD_NAME, n_fldattrs);

/*
* Get the total number of the field’s and vdata’s attributes - should be 2.
*/
n_vdattrs = VSnattrs (vdata_id);
printf ( "Number of attributes of the vdata and its fields: %d\n",
          n_vdattrs);

/*
* Get information about the vdata’s first attribute, indicated
* by the third parameter which is the index of the attribute.
*/
status_n = VSattrinfo (vdata_id, _HDF_VDATA, 0, vattr_name,
                       &vdata_type, &vdata_n_values, &vdata_size);

/*
* Get information about the first attribute of the field specified by
* field_index.
*/
status_n = VSattrinfo (vdata_id, field_index, 0, fattr_name, &field_type,
                       &field_n_values, &field_size);

/*
* Get the vdata’s first attribute.
*/
status_n = VSgetattr (vdata_id, _HDF_VDATA, 0, vattr_buf);
printf("Values of the vdata attribute = %c %c %c\n", vattr_buf[0],
                       vattr_buf[1], vattr_buf[2]);
```

```
   /*
   * Get the first attribute of the field specified by field_index.
   */
   status_n = VSgetattr (vdata_id, field_index, 0, fattr_buf);
   printf("Values of the field attribute = %d %d %d %d\n", fattr_buf[0],
                       fattr_buf[1], fattr_buf[2], fattr_buf[3]);

   /*
   * Terminate access to the vdata and to the VS interface, then close
   * the HDF file.
   */
   status_32 = VSdetach (vdata_id);
   status_n  = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
 program vdata_attributes
      implicit none
C
C     Parameter declaration
C
      character*18 FILE_NAME
      character*14 VDATA_NAME
      character*4  FIELD_NAME
      character*10 VATTR_NAME
      character*6  FATTR_NAME
      integer      VATTR_N_VALUES, FATTR_N_VALUES
C
      parameter (FILE_NAME    = ’General_Vdatas.hdf’,
     +           VDATA_NAME   = ’Solid Particle’,
     +           FIELD_NAME   = ’Mass’,
     +           VATTR_NAME   = ’Site Ident’,
     +           FATTR_NAME   = ’Scales’)
      parameter (VATTR_N_VALUES = 3,
     +           FATTR_N_VALUES = 4)

      integer DFACC_WRITE, FULL_INTERLACE, HDF_VDATA
      integer DFNT_INT32, DFNT_CHAR8
      parameter (DFACC_WRITE    =  2,
     +           FULL_INTERLACE =  0,
     +           HDF_VDATA      = -1,
     +           DFNT_INT32     = 24,
     +           DFNT_CHAR8     =  4)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vsffnd, vsfatch, vsfscat, vsfsnat,
     +        vsffnas, vsffidx, vsfnats, vsfainf, vsfgcat, vsfgnat,
     +        vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer   status
      integer   file_id, vdata_id, vdata_ref
      integer   field_index, n_vdattrs, n_fldattrs
      integer   vdata_type, vdata_n_values, vdata_size
      integer   field_type, field_n_values, field_size
      character vd_attr(VATTR_N_VALUES)
      integer   fld_attr(FATTR_N_VALUES)
      character vattr_buf(VATTR_N_VALUES)
      integer   fattr_buf(FATTR_N_VALUES)
      character vattr_name_out(30), fattr_name_out(30)
      data vd_attr /'A', 'B', 'C'/
      data fld_attr /2, 4, 6, 8/
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
C     Get the reference number of the vdata named VDATA_NAME.
C
      vdata_ref = vsffnd(file_id, VDATA_NAME)
C
C     Attach to the vdata for writing.
C
      vdata_id = vsfatch(file_id, vdata_ref, 'w')
C
C     Attach an attribute to the vdata, as it is indicated by second parameter.
C
      status = vsfscat(vdata_id, HDF_VDATA, VATTR_NAME, DFNT_CHAR8,
     +                  VATTR_N_VALUES, vd_attr)
C
C     Get the index of the field FIELD_NAME within the vdata.
C
      status = vsffidx(vdata_id, FIELD_NAME, field_index)
C
C     Attach an attribute to the field with the index field_index.
C
      status = vsfsnat(vdata_id, field_index, FATTR_NAME, DFNT_INT32,
     +                  FATTR_N_VALUES, fld_attr)

C
C     Get the number of attributes attached to the vdata's first
C     field - should be 0.
C
      n_fldattrs = vsffnas(vdata_id, 0)
      write(*,*) 'Number of attributes of the first field'
      write(*,*) ' of the vdata: ', n_fldattrs
C
C     Get the number of the attributes attached to the field specified by
C     index field_index - should be 1.
C
      n_fldattrs = vsffnas(vdata_id, field_index)
      write(*,*) 'Number of attributes of field ', FIELD_NAME,
     +           n_fldattrs
C
C     Get the total number of the field's and vdata's attributes - should be 2.
C
      n_vdattrs = vsfnats(vdata_id)
      write(*,*) 'Number of attributes of the vdata and its fields: ',
     +           n_vdattrs
C
C     Get information about the vdata's first attribute, indicated by
C     the third parameter, which is the index of the attribute.
C
```

```
      status = vsfainf(vdata_id, HDF_VDATA, 0, vattr_name_out,
     +                 vdata_type, vdata_n_values, vdata_size)
C
C     Get information about the first attribute of the field specified by
C     field_index.
C
      status = vsfainf(vdata_id, field_index, 0, fattr_name_out,
     +                 field_type, field_n_values, field_size)
C
C     Get the vdata's first attribute.
C
      status = vsfgcat(vdata_id, HDF_VDATA, 0, vattr_buf)
      write(*,*) 'Values of vdata attribute  ', vattr_buf
C
C     Get the first attribute of the field specified by field_index.
C
      status = vsfgnat(vdata_id, field_index, 0, fattr_buf)
      write(*,*)  'Values of the field attribute = ', fattr_buf
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.9. Obtaining Information about a Specific Vdata

Once a vdata has been located, its contents must be obtained. In this section four categories of routines that obtain vdata information are described:

- A general inquiry routine named **VSinquire**.
- A set of ***vdata query*** routines with names prefaced by "VSQuery".
- A set of ***vdata inquiry*** routines prefaced by "VS". Some of these routines retrieve specific vdata information which can also be retrieved by the general inquiry routine **VSinquire**.
- A set of ***field query*** routines with names prefaced by "VF".

4.9.1. Obtaining Vdata Information: VSinquire

VSinquire retrieves information about the vdata identified by the parameter *vdata_id*. The routine has the following syntax:

```
C:        status = VSinquire(vdata_id, &n_records, &interlace_mode, field-
                           name_list, &vdata_size, vdata_name);

FORTRAN:  status = vsfinq(vdata_id, n_records, interlace_mode, fieldname_list,
                           vdata_size, vdata_name)
```

The parameter *n_records* contains the returned number of records in the vdata, the parameter *interlace_mode* contains the returned interlace mode of the vdata contents, the parameter *field-name_list* is a comma-separated list of the returned names of all the fields in the vdata, the parameter *vdata_size* is the returned size, in bytes, of the vdata record, and the parameter *vdata_name* contains the returned name of the vdata.

If any of the parameters are set to `NULL` in C, the corresponding data will not be returned. **VSinquire** will return `FAIL` if it is called before **VSdefine** and **VSsetfield** on the same vdata.

VSinquire returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **VSinquire** are further defined in Table 4P.

TABLE 4P

VSinquire Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSinquire [intn] **(vsfinq)**	vdata_id	int32	integer	Vdata identifier
	n_records	int32 *	integer	Number of records in the vdata
	interlace_mode	int32 *	integer	Interlace mode
	fieldname_list	char *	character*(*)	Buffer for the list of field names
	vdata_size	int32 *	integer	Size in bytes of the vdata record
	vdata_name	char *	character*(*)	Name of the vdata

EXAMPLE 9.

Obtaining Vdata Information

This example illustrates the use of **VSgetid/vsfgid** and **VSinquire/vsfinq** to obtain information about all vdatas in an HDF file.

In this example, the program uses **VSgetid** to locate all vdatas in the HDF file "General_Vdatas.hdf", which is the output of Example 3. For each vdata found, if it is not the storage of an attribute, the program uses **VSinquire/vsfinq** to obtain information about the vdata and displays its information. Recall that an attribute is also stored as a vdata; the function **VSisattr/vsfisat** checks whether a vdata is a storage of an attribute.

C:

```
#include "hdf.h"

#define  FILE_NAME      "General_Vdatas.hdf"
#define  FIELD_SIZE     80         /* maximum length of all the field names */

main( )
{
   /************************* Variable declaration *************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         n_records,     /* to retrieve the number of records in the vdata   */
         interlace_mode,/* to retrieve the interlace mode of the vdata      */
         vdata_size,    /* to retrieve the size of all specified fields     */
         file_id, vdata_ref, vdata_id;
   char  fieldname_list[FIELD_SIZE], /* buffer to retrieve the vdata data   */
         vdata_name[VSNAMELENMAX];   /* buffer to retrieve the vdata name   */

   /********************** End of variable declaration *********************/

   /*
   * Open the HDF file for reading.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the VS interface.
   */
   status_n = Vstart (file_id);
   /*
   * Set vdata_ref to -1 to start the search from the beginning of file.
   */
   vdata_ref = -1;

   /*
   * Use VSgetid to obtain each vdata by its reference number then attach
   * to the vdata and get its information.  The loop terminates when
   * the last vdata is reached.
   */
   while ((vdata_ref = VSgetid (file_id, vdata_ref)) != FAIL)
   {
      /*
      * Attach to the current vdata for reading.
      */
      vdata_id = VSattach (file_id, vdata_ref, "r");

      /*
      * Test whether the current vdata is not a storage of an attribute, then
      * obtain and display its information.
      */
      if( VSisattr (vdata_id) != TRUE )
      {
         status_n = VSinquire (vdata_id, &n_records, &interlace_mode,
                               fieldname_list, &vdata_size, vdata_name);
         printf ("Vdata %s: - contains %d records\n\tInterlace mode: %s \
                 \n\tFields: %s - %d bytes\n\t\n", vdata_name, n_records,
                 interlace_mode == FULL_INTERLACE ? "FULL" : "NONE",
                 fieldname_list, vdata_size );
      }

      /*
      * Detach from the current vdata.
      */
      status_32 = VSdetach (vdata_id);
   } /* while */

   /*
   * Terminate access to the VS interface and close the HDF file.
   */
   status_n = Vend (file_id);
   status_32 = Hclose (file_id);
}
```

FORTRAN:

```
       program vdata_info
      implicit none
C
C     Parameter declaration
C
      character*18 FILE_NAME
      integer      DFACC_READ, FULL_INTERLACE
      integer      FIELD_SIZE
C
      parameter (FILE_NAME      = 'General_Vdatas.hdf',
     +           DFACC_READ     = 1,
     +           FULL_INTERLACE = 0,
     +           FIELD_SIZE     = 80)

C
C     Function declaration
```

```
C
      integer hopen, hclose
      integer vfstart, vsfatch, vsfgid, vsfinq,
     +        vsfisat, vsfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer      status
      integer      file_id, vdata_id, vdata_ref
      integer      n_records, interlace_mode, vdata_size
      character*64 vdata_name
      character*80 fieldname_list
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the VS interface.
C
      status = vfstart(file_id)
C
C     Set the reference number to -1 to start the search from the beginning
C     of the file.
C
      vdata_ref = -1
10    continue
C
C     Use vsfgid to obtain each vdata by its reference number then
C     attach to the vdata and get information. The loop terminates
C     when the last vdata is reached.
C
      vdata_ref = vsfgid(file_id, vdata_ref)
      if (vdata_ref .eq. -1) goto 100
C
C     Attach to the current vdata for reading.
C
      vdata_id = vsfatch(file_id, vdata_ref, 'r')
C
C     Test whether the current vdata is not a storage for an attribute,
C     then obtain and display its information.
      if (vsfisat(vdata_id) .ne. 1) then
          status = vsfinq(vdata_id, n_records, interlace_mode,
     +                    fieldname_list, vdata_size, vdata_name)
          write(*,*) 'Vdata: ', vdata_name
          write(*,*) 'contains ', n_records, ' records'
          if (interlace_mode .eq. 0) then
              write(*,*) 'Interlace mode: FULL'
          else
              write(*,*) 'Interlace mode: NONE'
          endif
          write(*,*) 'Fields: ', fieldname_list(1:30)
          write(*,*) 'Vdata record size in bytes :', vdata_size
          write(*,*)
      endif
C
C     Detach from the current vdata.
C
      status = vsfdtch(vdata_id)
      goto 10
```

```
100   continue
C
C     Terminate access to the vdata and to the VS interface, and
C     close the HDF file.
C
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

4.9.2. Obtaining Linked Block Information: VSgetblockinfo

VSgetblockinfo retrieves the block size and number of blocks employed in a linked block vdata data element. The parameter *vdata_id* identifies the vdata. The size of blocks, in bytes, is returned in *block_size* and the number of blocks in *num_blocks*.

If either the block size or the number of blocks used in a particular vdata is likely to differ from the default setting, **VSgetblockinfo** must be called before any data is read from a vdata.

VSgetblockinfo returns `SUCCESS` (or `0`) upon successful completion or `FAIL` (or `-1`). Its parameters are further defined in Table 4Q.

4.9.3. Obtaining Linked Block Information: VSgetblockinfo

VSgetexternalinfo retrieves external file and data information of a vdata, when the vdata has external element. The information includes the external file's name, the position, where the data had been written in the external file, and the length of that external data. **VSgetexternalinfo** will return `0` if the vdata does not have external element.

The syntax of **VSgetexternalinfo** is as follows:

```
C:         status = VSgetexternalinfo(vdata_id, buf_size, filename, &offset,
                          &length);

FORTRAN:   Currently unavailable
```

The application must provide sufficient buffer for the external file name. When the external file name is available and *buf_size* is 0, **VSgetexternalinfo** simply returns the length of the external file name. Thus, application can call **VSgetexternalinfo** passing in 0 for *buf_size* first, then allocate the buffer sufficiently before calling **VSgetexternalinfo** again passing in the proper length for *buf_size* and appropriately allocated buffer *filename*. **VSgetexternalinfo** stores the external file name in *filename* up to the name's length or the value in *buf_size*, whichever smaller.

VSgetexternalinfo stores in the parameter *offset* the number of bytes from the beginning of the external file to the location where the first byte of data had been written and in the parameter *length* the length of the data.

VSgetexternalinfo returns one of the following values:

- the actual length of the external file name or the length of the retrieved file name, if there is external element
- `0`, if there is no external element
- `FAIL` (or `-1`), if failure occurs

The parameters of **VSgetexternalinfo** are described in Table 4Q.

TABLE 4Q **VSgetblockinfo Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSgetblockinfo [intn] **(vsfgetblinfo)**	vdata_id	int32	integer	Vdata identifier
	block_size	int32	integer	Size of each block, in bytes
	num_blocks	int32	integer	Number of linked blocks
VSgetexternalinfo [intn] **(unavailable)**	vdata_id	int32	N/A	Vdata identifier
	buf_size	int32	N/A	Size of external file name's buffer
	filename	char *	N/A	External file name
	offset	int32 *	N/A	Offset of external data
	length	int32 *	N/A	Length of external data

4.9.4. VSQuery Vdata Information Retrieval Routines

The syntax of the VSQuery routines are as follows:

```
C:          status = VSQueryname(vdata_id, vdata_name);
            status = VSQueryfields(vdata_id, fields);
            status = VSQueryinterlace(vdata_id, &interlace_mode);
            status = VSQuerycount(vdata_id, &n_records);
            vdata_tag = VSQuerytag(vdata_id);
            vdata_ref = VSQueryref(vdata_id);
            status = VSQueryvsize(vdata_id, &vdata_vsize);

FORTRAN:    status = vsqfname(vdata_id, vdata_name)
            status = vsqfflds(vdata_id, fields)
            status = vsqfintr(vdata_id, interlace_mode)
            status = vsqfnelt(vdata_id, n_records)
            vdata_tag = vsqtag(vdata_id)
            vdata_ref = vsqref(vdata_id)
            status = vsqfvsiz(vdata_id, vdata_vsize)
```

All VSQuery routines except **VSQuerytag** and **VSQueryref** have two arguments. The first argument identifies the vdata to be queried. The second argument is the type of vdata information being requested.

- **VSQueryname** retrieves the name of the specified vdata.
- **VSQueryfields** retrieves the names of the fields in the specified vdata.
- **VSQueryinterlace** retrieves the interlace mode of the specified vdata.
- **VSQuerycount** retrieves the number of records in the specified vdata.
- **VSQuerytag** returns the tag of the specified vdata.
- **VSQueryref** returns the reference number of the specified vdata.
- **VSQueryvsize** retrieves the size, in bytes, of a record in the specified vdata.

VSQuerytag and **VSQueryref** return the tag and reference number, respectively, or `FAIL` (or `-1`). All other routines return `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for these routines are listed in Table 4R.

TABLE 4R **VSQuery Routines Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSQueryname [intn] **(vsqfname)**	vdata_id	int32	integer	Vdata identifier
	vdata_name	char *	character*(*)	Name of the vdata
VSQueryfields [intn] **(vsqfflds)**	vdata_id	int32	integer	Vdata identifier
	fields	char *	character*(*)	Comma-separated list of the field names in the vdata
VSQueryinterlace [intn] **(vsqfintr)**	vdata_id	int32	integer	Vdata identifier
	interlace_mode	int32 *	integer	Interlace mode
VSQuerycount [intn] **(vsqfnelt)**	vdata_id	int32	integer	Vdata identifier
	n_records	int32 *	integer	Number of records in the vdata
VSQueryvsize [intn] **(vsqfvsiz)**	vdata_id	int32	integer	Vdata identifier
	vdata_size	int32 *	integer	Size in bytes of the vdata record
VSQuerytag [int32] **(vsqtag)**	vdata_id	int32	integer	Vdata identifier
VSQueryref [int32] **(vsqref)**	vdata_id	int32	integer	Vdata identifier

4.9.5. Other Vdata Information Retrieval Routines

The routines described in this section, with names prefaced by "VS", are used to obtain specific types of vdata information. The syntax of these routines are as follows:

```
C:          num_of_records = VSelts(vdata_id);
            num_of_fields = VSgetfields(vdata_id, fieldname_list);
            interlace_mode = VSgetinterlace(vdata_id);
            size_of_fields = VSsizeof(vdata_id, fieldname_list);
            status = VSgetname(vdata_id, vdata_name);
            status = VSgetclass(vdata_id, vdata_class);

FORTRAN:    num_of_records = vsfelts(vdata_id)
            num_of_fields = vsfgfld(vdata_id, fieldname_list)
            interlace_mode = vsfgint(vdata_id)
            size_of_fields = vsfsiz(vdata_id, fieldname_list)
            status = vsfgnam(vdata_id, vdata_name)
            status = vsfcls(vdata_id, vdata_class)
```

With the exception of **VSgetclass**, the information obtained through these routines can also be obtained through **VSinquire**. **VSinquire** provides a way to query commonly used vdata information with one routine call. The VS routines in this section are useful in situations where the HDF programmer wishes to obtain only specific information.

- **VSelts** returns the number of records in the specified vdata or `FAIL` (or `-1`).
- **VSgetfields** retrieves the names of all the fields in the specified vdata and returns the number of retrieved fields or `FAIL` (or `-1`).
- **VSgetinterlace** returns the interlace mode of the specified vdata or `FAIL` (or `-1`).
- **VSsizeof** returns the size, in bytes, of the specified fields or `FAIL` (or `-1`).

- **VSgetname** retrieves the name of the specified vdata and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`).
- **VSgetclass** retrieves the class of the specified vdata and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`).

The parameters for these routines are described in Table 4S.

TABLE 4S **VSelts, VSgetfields, VSgetinterlace, VSsizeof, VSgetname, and VSgetclass Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN -77**	
VSelts [int32] **(vsfelts)**	vdata_id	int32	integer	Vdata identifier
VSgetfields [int32] **(vsfgfld)**	vdata_id	int32	integer	Vdata identifier
	fieldname_list	char *	character*(*)	List of field names to be queried
VSgetinterlace [int32] **(vsfgint)**	vdata_id	int32	integer	Vdata identifier
VSsizeof [int32] **(vsfsiz)**	vdata_id	int32	integer	Vdata identifier
	fieldname_list	char *	character*(*)	List of field names to be queried
VSgetname [int32] **(vsfgnam)**	vdata_id	int32	integer	Vdata identifier
	vdata_name	char *	character*(*)	Vdata name
VSgetclass [int32] **(vsfcls)**	vdata_id	int32	integer	Vdata identifier
	vdata_class	char *	character*(*)	Class name of the vdata to be queried

4.9.6. VF Field Information Retrieval Routines

Routines whose names are prefaced by "VF" are used for obtaining information about specific fields in a vdata. The syntax of these routines are as follows:

```
C:          field_name = VFfieldname(vdata_id, field_index);
            field_file_size = VFfieldesize(vdata_id, field_index);
            field_mem_size = VFfieldisize(vdata_id, field_index);
            num_of_fields = VFnfields(vdata_id);
            field_order = VFfieldorder(vdata_id, field_index);
            field_type = VFfieldtype(vdata_id, field_index);

FORTRAN:    field_name = vffname(vdata_id, field_index, field_name)
            field_file_size = vffesiz(vdata_id, field_index)
            field_mem_size = vffisiz(vdata_id, field_index)
            num_of_fields = vfnflds(vdata_id)
            field_order = vffordr(vdata_id, field_index)
            field_type = vfftype(vdata_id, field_index)
```

The functionality of each of the VF routines is as follows:

- **VFfieldname** returns the name of the specified field.
- **VFfieldesize** returns the size of the specified field as stored in the HDF file. This is the size of the field as tracked by the HDF library.

- **VFfieldisize** returns the size of the specified field as stored in memory. This is the native machine size of the field.
- **VFnfields** returns the number of fields in the specified vdata.
- **VFfieldorder** returns the order of the specified field.
- **VFfieldtype** returns the data type of the specified field.

If the operations are unsuccessful, these routines return `FAIL` (or `-1`). The parameters for all of these routines are described in Table 4T.

TABLE 4T **VF Routines Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VFfieldname [char *] (vffname)	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Field index
	field_name		character*(*)	Field name (FORTRAN-77 only)
VFfieldesize [int32] **(vffesiz)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Field index
VFfieldisize [int32] **(vffisiz)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Field index
VFnfields [int32] **(vfnflds)**	vdata_id	int32	integer	Vdata identifier
VFfieldorder [int32] **(vffordr)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Field index
VFfieldtype [int32] **(vfftype)**	vdata_id	int32	integer	Vdata identifier
	field_index	int32	integer	Field index

CHAPTER 5 --

Vgroups (V API)

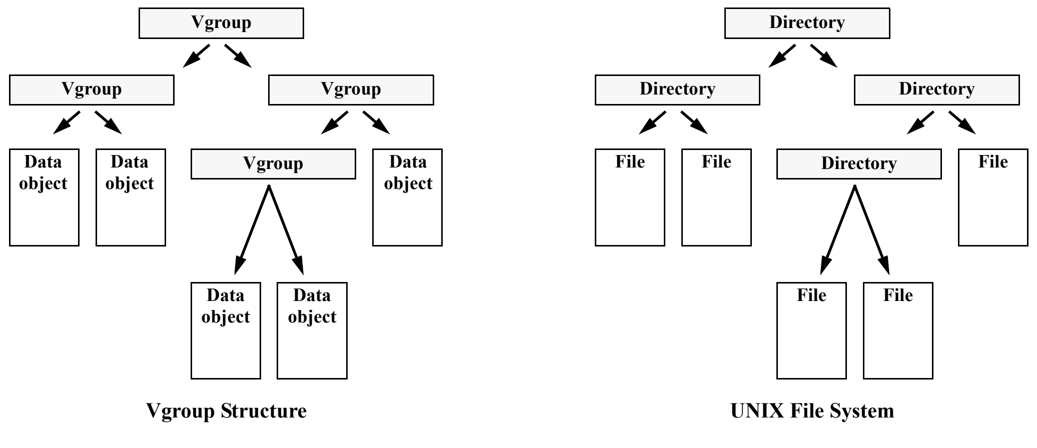
5.1. Chapter Overview

This chapter describes the vgroup data model and the Vgroup interface (also called the V interface or the V API). The first section describes the vgroup data model. The second section introduces the Vgroup interface, followed by a presentation of a programming model for vgroups. The next three sections describe the use of the Vgroup interface in accessing and creating vgroups. The final two sections cover vgroup attributes and obsolete Vgroup interface routines.

5.2. The Vgroup Data Model

A ***vgroup*** is a structure designed to associate related data objects. The general structure of a vgroup is similar to that of the UNIX file system in that the vgroup may contain references to other vgroups or HDF data objects just as the UNIX directory may contain subdirectories or files (see Figure 5a). In previous versions of HDF, the data objects in a vgroup were limited to vdatas and vgroups. The data objects that belong to a vgroup are often referred to as the vgroup's ***members***.

FIGURE 5a **Similarity of the HDF Vgroup Structure and the UNIX File System**



5.2.1. Vgroup Names and Classes

A vgroup can have a ***name*** and/or a ***class*** associated with it. The vgroup name and class are useful in describing and classifying the data objects belonging to the vgroup.

A vgroup name is a character string and is used to semantically distinguish between vgroups in an HDF file. Multiple vgroups in a file can have the same name; however, unique names make it easier to distinguish among vgroups and are recommended.

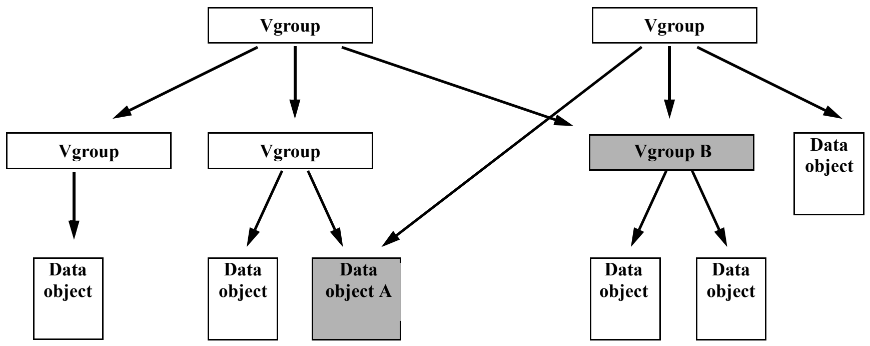
A ***vgroup class*** is a character string and can be used to classify data objects by their intended use. For example, a vdata object named "Storm Tracking Data - 5/11/94" and another vdata object named "Storm Tracking Data - 6/23/94" can be grouped together under a vgroup named "Storm Tracking Data - 1994". If the data was collected in Anchorage, Alaska the class name might be "Anchorage Data", particularly if other vgroups contain storm track data collected in different locations.

The specific use of the vgroup name and class name is solely determined by HDF users.

5.2.2. Vgroup Organization

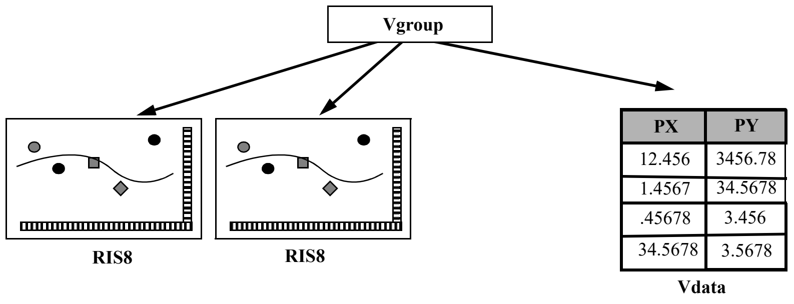
There are many ways to organize vgroups through the use of the Vgroup interface. Vgroups may contain any number of vgroups and data objects, including data objects and vgroups that are members of other vgroups. Therefore, a data object may have more than one parent vgroup. For example, Data object A and Vgroup B, shown in Figure 5b, are members of multiple vgroups with different organizational structures.

FIGURE 5b **Sharing Data Objects among Vgroups**



A vgroup can contain any combination of data objects. Figure 5c illustrates a vgroup that contains two raster images and a vdata.

FIGURE 5c **A Vgroup Containing Two 8-Bit Raster Images, or RIS8 Objects, and a Vdata**



5.2.3. An Example Using Vgroups

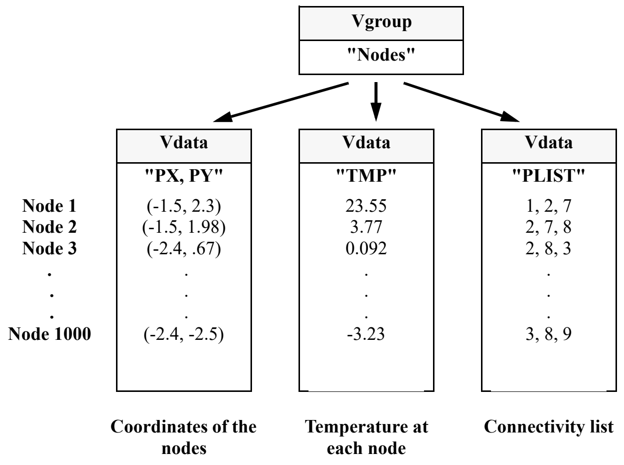
Although vgroups can contain any combination of HDF data objects, it is often useful to establish conventions on the content and structure of vgroups. This section, with the illustration in Figure 5d, describes an example of a vgroup convention that is used by scientific and graphics programmers to describe the surfaces of a mathematical or material object as well as its properties.

This vgroup consists of one list of coordinate data, one list of connectivity data, and one list of node property data. These three lists are stored in separate vdata objects within the vgroup.

Each 2-dimensional coordinate in the list of coordinate data defines the relative location of a vertex, or ***node***. Each entry in the list of connectivity data is an ordered list of node numbers which describes a polygon. This ordered list is referred to as the ***connectivity list***. For example, the number "2" as an item in a connectivity list would represent the second entry in the node table. ***Node properties*** are user-defined values attached to each node within the polygon and can be numbers or characters.

For example, consider a heated mesh of 400 triangles formed by connecting 1000 nodes. A vgroup describing this mesh might contain the coordinates of the vertices, the temperature value of the vertices, and a connectivity list describing the edges of the triangles.

FIGURE 5d **Vgroup Structure Describing a Heated Mesh**



5.3. The Vgroup Interface

The Vgroup interface consists of routines for creating and accessing vgroups, and getting information about vgroups and their members.

5.3.1. Vgroup Interface Routines

Vgroup interface routine names are prefaced by "V" in C and by "vf" in FORTRAN-77. These routines are categorized as follows:

- ***Access/Create routines*** control access to the Vgroup interface and to individual vgroups.
- ***Manipulation routines*** modify vgroups' characteristics, and add and delete vgroups' members.
- ***Vgroup inquiry routines*** obtain information about vgroups. Some of these routines are useful for locating vgroups in a file.

- ***Member inquiry routines*** obtain information about members of vgroups.
- ***Attributes routines*** provide information about vgroups' attributes.

The Vgroup interface routines are listed in Table 5A and described in the following sections.

TABLE 5A

Vgroup Interface Routines

Category	Routine Name		Description
	C	FORTRAN-77	
Access/Create	`Vstart`	`vfstart`	Initializes the Vdata and Vgroup interfaces (Section "*Accessing Files and Vgroups: Vstart and Vattach"*)
	`Vattach`	`vfatch`	Establishes access to a vgroup (Section "*Accessing Files and Vgroups: Vstart and Vattach"*)
	`Vdetach`	`vfdtch`	Terminates access to a vgroup (Section "*Terminating Access to Vgroups and Files: Vdetach and Vend"*)
	`Vend`	`vfend`	Terminates access to the Vdata and Vgroup interfaces (Section "*Terminating Access to Vgroups and Files: Vdetach and Vend"*)
Manipulation	`VHmakegroup`	`vhfmkgp`	Builds a vgroup containing elements specified by their tags/refs (Section "*Building a Vgroup with or without Elements: VHmakegroup"*)
	`Vaddtagref`	`vfadtr`	Adds an HDF data object to a vgroup (Section "*Inserting Any HDF Data Object into a Vgroup: Vaddtagref"*)
	`Vdelete`	`vdelete`	Removes a vgroup from a file (Section "*Deleting a Vgroup from a File: Vdelete"*)
	`Vdeleteta-gref`	`vfdtr`	Detaches a member from a vgroup (Section "*Deleting a Data Object from a Vgroup: Vdeletetagref"*)
	`Vinsert`	`vfinsrt`	Adds a vgroup or vdata to an existing vgroup (Section "*Inserting a Vdata or Vgroup Into a Vgroup: Vinsert"*)
	`Vsetclass`	`vfscls`	Assigns a class name to a vgroup (Section "*Assigning a Vgroup Name and Class: Vsetname and Vsetclass"*)
	`Vsetname`	`vfsnam`	Assigns a name to a vgroup (Section "*Assigning a Vgroup Name and Class: Vsetname and Vsetclass"*)
Vgroup Inquiry	`Vfind`	`vfind`	Returns the reference number of a vgroup given its name (Section "*Locating a Vgroup Given Its Name: Vfind"*)
	`Vfindclass`	`vfndcls`	Returns the reference number of a vgroup specified by class name (Section "*Locating a Vgroup Given Its Class Name: Vfindclass"*)
	`Vgetclass`	`vfgcls`	Retrieves the class of a vgroup (Section "*Obtaining the Class Name of a Vgroup: Vgetclass"*)
	`Vget-classname-len`	`[unavail-able]`	Retrieves the length of a vgroup's class name (Section "*Obtaining the Length of a Vgroup's Class Name: Vgetclassnamelen"*)
	`Vgetid`	`vfgid`	Returns the reference number for the next vgroup in the HDF file (Section "*Sequentially Searching for a Vgroup: Vgetid"*)
	`Vgetname`	`vfgnam`	Retrieves the name of a vgroup (Section "*Obtaining the Name of a Vgroup: Vgetname"*)
	`Vgetnamelen`	`[unavail-able]`	Retrieves the length of a vgroup's name (Section "*Obtaining the Length of a Vgroup's Name: Vgetnamelen"*)
	`Vgetversion`	`vfgver`	Returns the vgroup version of a vgroup (Section "*Obtaining the Vgroup Version Number of a Given Vgroup: Vgetversion"*)
	`Vinquire`	`vfinq`	Retrieves general information about a vgroup (Section "*Determining the Number of Members and Vgroup Name: Vinquire"*)
	`Vlone`	`vflone`	Retrieves the reference numbers of vgroups that are not members of other vgroups (Section "*Locating Lone Vgroups: Vlone"*)
	`Vntagrefs`	`vfntr`	Returns the number of tag/reference number pairs contained in the specified vgroup (Section "*Obtaining the Number of Objects in a Vgroup: Vntagrefs"*)
	`VQueryref`	`vqref`	Returns the reference number of a vgroup (Section "*Retrieving the Reference Number of a Vgroup: VQueryref"*)
	`VQuerytag`	`vqtag`	Returns the tag of a vgroup (Section "*Retrieving the Tag of a Vgroup: VQuerytag"*)

Member Inquiry	`Vflocate`	`vffloc`	Locates a vdata in a vgroup given a list of field names (Section "*Locating a Vdata in a Vgroup Given Vdata Fields: Vflocate"*)
	`Vgetnext`	`vfgnxt`	Returns the identifier of the next vgroup or vdata in a vgroup (Obsolete) (Section "*Determining the Next Vgroup or Vdata Identifier: Vgetnext"*)
	`Vgettagref`	`vfgttr`	Retrieves a tag/reference number pair for a data object in the vgroup (Section "*Obtaining the Tag/Reference Number Pair of a Data Object within a Vgroup : Vgettagref"*)
	`Vgettagrefs`	`vfgttrs`	Retrieves the tag/reference number pairs of all of the data objects belonging to a vgroup (Section "*Obtaining the Tag/Reference Number Pairs of Data Objects in a Vgroup: Vgettagrefs"*)
	`Vinqtagref`	`vfinqtr`	Determines whether a data object belongs to a vgroup (Section "*Testing Whether a Data Object Belongs to a Vgroup: Vinqtagref"*)
	`Visvg`	`vfisvg`	Determines whether a data object is a vgroup within another vgroup (Section "*Testing Whether a Data Object within a Vgroup is a Vgroup: Visvg"*)
	`Visvs`	`vfisvs`	Determines whether a data object is a vdata within a vgroup (Section "*Testing Whether an HDF Object within a Vgroup is a Vdata: Visvs"*)
	`Vnrefs`	`vnrefs`	Retrieves the number of tags of a given tag type in a vgroup (Section "*Retrieving the Number of Tags of a Given Type in a Vgroup: Vnrefs"*)
Attributes	`Vattrinfo`	`vfainfo`	Retrieves information of a vgroup attribute (Section "*Obtaining Information on a Given Vgroup Attribute: Vattrinfo"*)
	`Vfindattr`	`vffdatt`	Returns the index of a vgroup attribute given the attribute name (Section "*Retrieving the Index of a Vgroup Attribute Given the Attribute Name: Vfindattr"*)
	`Vgetattr`	`vfgnatt/` `vfgcatt`	Retrieves the values of a vgroup attribute (Section "*Retrieving the Values of a Given Vgroup Attribute: Vgetattr"*)
	`Vnattrs`	`vfnatts`	Returns the total number of vgroup attributes (Section "*Obtaining the Total Number of Vgroup Attributes: Vnattrs and Vnattrs2"*)
	`Vsetattr`	`vfsnatt/` `vfscatt`	Sets the attribute of a vgroup (Section "*Setting the Attribute of a Vgroup: Vsetattr"*)

5.3.2. Identifying Vgroups in the Vgroup Interface

The Vgroup interface identifies vgroups in several ways. In some cases, a vgroup can be accessed directly through the use of its unique ***reference number***. In other cases, the reference number and the routine **Vattach** are used to obtain a vgroup identifier. The reference number of a vgroup can be obtained from the name or the class of the vgroup, or by sequentially traversing the file. The concept of reference number is discussed in Section "*Data Descriptor"*.

When a vgroup is attached or created, it is assigned an identifier, called ***vgroup id***. After a vgroup has been attached or created, its identifier is used by the Vgroup interface routines in accessing the vgroup.

5.4. Programming Model for the Vgroup Interface

The programming model for accessing vgroups is as follows:

1. Open an HDF file.
2. Initialize the Vgroup interface.
3. Create a new vgroup or open an existing one.
4. Perform the desired operations on the vgroup.
5. Terminate access to the vgroup.
6. Terminate access to the Vgroup interface.
7. Close the file.

These steps correspond to the following sequence of function calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);
            vgroup_id = Vattach(file_id, vgroup_ref, vg_access_mode);
            <Optional operations>
            status = Vdetach(vgroup_id);
            status = Vend(file_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            status = vfstart(file_id)
            vgroup_id = vfatch(file_id, vgroup_ref, vg_access_mode)
            <Optional operations>
            status = vfdtch(vgroup_id)
            status = vfend(file_id)
            status = hclose(file_id)
```

The calling program must obtain a separate vgroup identifier for each vgroup to be accessed.

5.4.1. Accessing Files and Vgroups: Vstart and Vattach

An HDF file must be opened by **Hopen** before it can be accessed using the Vgroup interface. **Hopen** is described in Chapter 2, *HDF Fundamentals*.

The Vgroup interface routines are used in a similar manner to the Vdata interface routines. Before performing operations on a vgroup, a calling program must call **Vstart** for every file to be accessed. **Vstart** initializes the internal vgroup structures in a file. **Vstart** takes one argument, the file identifier returned by **Hopen**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). Note that the **Vstart** routine is used by both the Vdata and Vgroup interfaces.

The calling program must also call one **Vattach** for every vgroup to be accessed. **Vattach** provides access to an individual vgroup for all read and write operations. **Vattach** takes three arguments: *file_id*, *vgroup_ref*, and *vg_access_mode*, and returns either a vgroup identifier or `FAIL` (or `-1`).

The argument *file_id* is the file identifier returned by **Hopen**. The parameter *vgroup_ref* is the reference number that identifies the vgroup to be accessed. Specifying *vgroup_ref* with a value of `-1` will create a new vgroup; specifying *vgroup_ref* with a nonexistent reference number will return an error code of `FAIL` (or `-1`); and specifying *vgroup_ref* with a valid reference number will initiate access to the corresponding vgroup.

When a new vgroup is created, it does not have any members. Additional operations must be performed to add other HDF data objects to the vgroup. Refer to Section 5.5., "Creating and Writing to a Vgroup" for information.

To access an existing vdata, its reference number must be obtained. The Vgroup interface includes two routines for this purpose, **Vfind** and **Vgetid**. **Vfind** can be used to obtain the reference number of a vgroup when the vgroup's name is known. **Vgetid** can be used to obtain the reference number by sequentially traversing the file. These routines are discussed in Section 5.6.1.9., "Locating a Vgroup Given Its Name: Vfind" and Section 5.6.1.2., "Sequentially Searching for a Vgroup: Vgetid".

The parameter *vg_access_mode* in **Vattach** specifies the type of access ("*r*" or "*w*") required for operations on the selected vgroup.

Multiple attaches may be made to a vgroup, which will result in several vgroup identifiers being assigned to the same vgroup. Termination must be properly handled as described in the next section.

The parameters of **Vstart** and **Vattach** are defined in Table 5B.

5.4.2. Terminating Access to Vgroups and Files: Vdetach and Vend

Successfully terminating access to a vgroup requires one **Vdetach** call for every **Vattach** call made. Similarly, successfully terminating access to the Vgroup interface requires one **Vend** call for every **Vstart** call made.

Vdetach terminates access to a vgroup by updating internal library structures and freeing all memory associated with the vgroup and allocated by **Vattach**. Once a vgroup is detached, its identifier is invalid and any attempts to access this vgroup identifier will result in an error condition. **Vdetach** takes one argument, *vgroup_id*, the vgroup identifier returned by **Vattach**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`).

Vend releases all internal data structures allocated by **Vstart**. Attempts to use the Vgroup interface identifier after calling **Vend** will produce errors. **Vend** takes one argument, *file_id*, the file identifier returned by **Hopen**, and returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). Note that the first **Vend** call to a file must occur after all **Vdetach** calls for the vgroups in the same file have been made. Note also that the **Vend** routine is used by both the Vdata and Vgroup interfaces.

Hclose must be used to terminate access to the HDF file and only after all proper **Vend** calls are made. **Hclose** is described in Chapter 2, *HDF Fundamentals*.

The parameters of **Vdetach** and **Vend** are also defined in Table 5B.

TABLE 5B **Vstart, Vattach, Vdetach, and Vend Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vstart [intn] **(vfstart)**	file_id	int32	integer	File identifier
Vattach [int32] **(vfatch)**	file_id	int32	integer	File identifier
	vgroup_ref	int32	integer	Reference number for an existing vgroup or -1 to create a new one
	vg_access_mode	char *	character*(*)	Access mode of the vgroup operation
Vdetach [int32] **(vfdtch)**	vgroup_id	int32	integer	Vgroup identifier
Vend [intn] **(vfend)**	file_id	int32	integer	File identifier

EXAMPLE 1. **Creating HDF Files and Vgroups**

This example illustrates the use of **Hopen/hopen**, **Vstart/vfstart**, **Vattach/vfatch**, **Vdetach/vfdtch**, **Vend/vfend**, and **Hclose/hclose** to create and to access two vgroups in an HDF file.

The program creates the HDF file, named "Two_Vgroups.hdf", and two vgroups stored in the file. Note that, in this example, the program only create two empty vgroups.

C:

```
#include "hdf.h"

#define  FILE_NAME    "Two_Vgroups.hdf"

main()
{
   /************************* Variable declaration **************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         vgroup_ref = -1,
         vgroup1_id, vgroup2_id, file_id;

   /********************** End of variable declaration **********************/

   /*
   * Create the HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Create the first vgroup.  Note that the vgroup reference number is set
   * to -1 for creating and the access mode is "w" for writing.
   */
   vgroup1_id = Vattach (file_id, vgroup_ref, "w");

   /*
   * Create the second vgroup.
   */
   vgroup2_id = Vattach (file_id, vgroup_ref, "w");

   /*
   * Any operations on the vgroups.
   */

   /*
   * Terminate access to the first vgroup.
   */
   status_32 = Vdetach (vgroup1_id);

   /*
   * Terminate access to the second vgroup.
   */
   status_32 = Vdetach (vgroup2_id);

   /*
   * Terminate access to the V interface and close the HDF file.
   */
   status_n = Vend (file_id);
   status_n = Hclose (file_id);
}
```

FORTRAN:

```
      program  create_vgroup
      implicit none
```

```
C
C     Parameter declaration
C
      character*15 FILE_NAME
C
      parameter (FILE_NAME = 'Two_Vgroups.hdf')
      integer DFACC_CREATE
      parameter (DFACC_CREATE = 4)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer vgroup1_id, vgroup2_id, vgroup_ref
C
C**** End of variable declaration ************************************
C
C
C     Create the HDF file.
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Create the first vgroup. Note that the vgroup reference number is set
C     to -1 for creating and the access mode is 'w' for writing.
C
      vgroup_ref = -1
      vgroup1_id = vfatch(file_id, vgroup_ref, 'w')
C
C     Create the second vgroup.
C
      vgroup2_id = vfatch(file_id, vgroup_ref, 'w')
C
C     Any operations on the vgroups.
C
C     ..............................
C
C     Terminate access to the first vgroup.
C
      status = vfdtch(vgroup1_id)
C
C     Terminate access to the second vgroup.
C
      status = vfdtch(vgroup2_id)
C
C     Terminate access to the V interface and close the HDF file.
C
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

5.5. Creating and Writing to a Vgroup

There are two steps involved in the creation of a vgroup: creating the vgroup and inserting data objects into it. Any HDF data object can be inserted into a vgroup. Creation and insertion operations are usually performed at the same time, but that is not required.

HDF provides two routines that insert an HDF data object into a vgroup, **Vaddtagref** and **Vinsert**. **Vaddtagref** can insert any HDF data object into a vgroup, but requires that the tag and reference number of the object be available. Refer to Section "Data Descriptor" for the description of tags and reference numbers for HDF data objects. **Vinsert** only inserts a vdata or a vgroup to a vgroup, but only requires the identifier of the vdata or the vgroup.

Creating a vgroup with a member involves the following steps:

1. Open the HDF file.
2. Initialize the Vgroup interface.
3. Create the new vgroup.
4. Optionally assign a vgroup name.
5. Optionally assign a vgroup class.
6. Insert a data object.
7. Terminate access to the vgroup.
8. Terminate access to the Vgroup interface.
9. Close the HDF file.

These steps correspond to the following sequence of function calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            status = Vstart(file_id);
            vgroup_id = Vattach(file_id, vgroup_ref, vg_access_mode);
            status = Vsetname(vgroup_id, vgroup_name);
            status = Vsetclass(vgroup_id, vgroup_class);

            /* Use either Vinsert to add a vdata or a vgroup, or
               Vaddtagref to add any data object */
            num_of_tag_refs = Vaddtagref(vgroup_id, obj_tag, obj_ref);
    OR      obj_pos = Vinsert(vgroup_id, v_id);

            status = Vdetach(vgroup_id);
            status = Vend(file_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            status = vfstart(file_id)
            vgroup_id = vfatch(file_id, vgroup_ref, vg_access_mode)
            status = vfsnam(vgroup_id, vdata_name)
            status = vfscls(vgroup_id, vdata_class)

C           Use either Vinsert to add a vdata or a vgroup, or Vaddtagref to
C           add any data object
            num_of_tag_refs = vfadtr(vgroup_id, obj_tag, obj_ref)
    OR      obj_pos = vfinsrt(vgroup_id, v_id)

            status = vfdtch(vgroup_id)
            status = vfend(file_id)
            status = hclose(file_id)
```

The parameter *v_id* in the calling sequence is either a vdata or vgroup identifier. The parameter *vgroup_id* is the vgroup identifier returned by **Vattach**.

When a new vgroup is created, the value of *vgroup_ref* must be set to `-1` and the value of *vg_access_mode* must be "*w*".

5.5.1. Assigning a Vgroup Name and Class: Vsetname and Vsetclass

Vsetname assigns a name to a vgroup. The parameter *vgroup_name* is a character string with the name to be assigned to the vgroup. If **Vsetname** is not called, the vgroup name is set to a zero-length character string. A name may be assigned and reset any time after the vgroup is created.

Vsetclass assigns a class to a vgroup. The parameter *vgroup_class* is a character string with the class name to be assigned to the vgroup. If **Vsetclass** is not called, the vgroup class is set to a zero-length string. As with the vgroup names, the class may be set and reset at any time after the vgroup is created.

Starting from release 4.2.4, the maximum length of vgroup's name is no longer limited to `VGNAMELENMAX` (or `64`) and release 4.2.5 for vgroup's class name.

Vsetname and **Vsetclass** return either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of these routines are further described in Table 5C.

5.5.2. Inserting Any HDF Data Object into a Vgroup: Vaddtagref

Vaddtagref inserts HDF data objects into the vgroup identified by *vgroup_id*. HDF data objects may be added to a vgroup when the vgroup is created or at any point thereafter.

The parameters *obj_tag* and *obj_ref* in **Vaddtagref** are the tag and reference number, respectively, of the data object to be inserted into the vgroup. Note that duplicated tag and reference number pairs are allowed.

Vaddtagref returns the total number of tag and reference number pairs, i.e., the total number of data objects, in the vgroup if the operation is successful, and `FAIL` (or `-1`) otherwise. The parameters of **Vaddtagref** are further described in Table 5C.

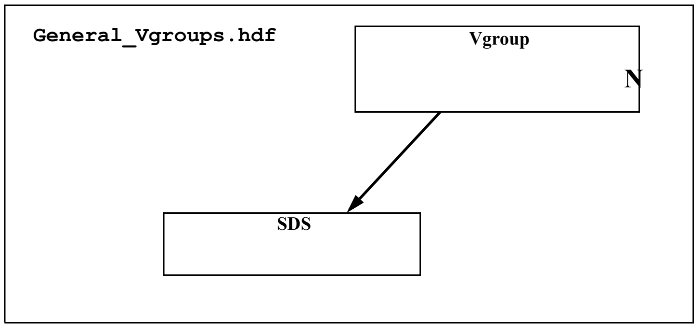
Note that **Vaddtagref** does *not* verify that the tag and reference number exist.

EXAMPLE 2. **Adding an SDS to a New Vgroup**

This example illustrates the use of **Vaddtagref/vfadtr** to add an HDF data object, an SDS specifically, to a vgroup.

In this example, the program first creates the HDF file "General_Vgroups.hdf", then an SDS in the SD interface, and a vgroup in the Vgroup interface. The SDS is named "Test SD" and is a one-dimensional array of type int32 of 10 elements. The vgroup is named "SD Vgroup" and is of class "Common Vgroups". The program then adds the SDS to the vgroup using **Vaddtagref/vfadtr**. Notice that, when the operations are complete, the program explicitly terminates access to the SDS, the vgroup, the SD interface, and the Vgroup interface before closing the HDF file. Refer to

Chapter 3, *Scientific Data Sets (SD API)* for the discussion of the SD routines used in this example.



C:

```
#include   "hdf.h"      /* Note: in this example, hdf.h can be omitted...*/
#include   "mfhdf.h"    /* ...since mfhdf.h already includes hdf.h */

#define  FILE_NAME    "General_Vgroups.hdf"
#define  SDS_NAME     "Test SD"
#define  VG_NAME      "SD Vgroup"
#define  VG_CLASS     "Common Vgroups"

main()
{
   /************************* Variable declaration *************************/

   intn   status_n;     /* returned status for functions returning an intn  */
   int32  status_32,    /* returned status for functions returning an int32 */
          sd_id,        /* SD interface identifier */
          sds_id,       /* data set identifier */
          sds_ref,      /* reference number of the data set */
          dim_sizes[1], /* dimension of the data set - only one */
          rank = 1,     /* rank of the data set array */
          vgroup_id,    /* vgroup identifier */
          file_id;      /* HDF file identifier, same for V interface */

   /********************** End of variable declaration *********************/

   /*
   * Create the HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Initialize the SD interface.
   */
   sd_id = SDstart (FILE_NAME, DFACC_WRITE);

   /*
   * Set the size of the SDS's dimension.
   */
   dim_sizes[0] = 10;

   /*
   * Create the SDS.
   */
   sds_id = SDcreate (sd_id, SDS_NAME, DFNT_INT32, rank, dim_sizes);

   /*
   * Create a vgroup and set its name and class.
   */
   vgroup_id = Vattach (file_id, -1, "w");
   status_32 = Vsetname (vgroup_id, VG_NAME);
   status_32 = Vsetclass (vgroup_id, VG_CLASS);

   /*
   * Obtain the reference number of the SDS using its identifier.
   */
   sds_ref = SDidtoref (sds_id);

   /*
   * Add the SDS to the vgroup.  Note: the tag DFTAG_NDG is used
   * when adding an SDS.  Refer to Appendix A for the entire list of tags.
   */
   status_32 = Vaddtagref (vgroup_id, DFTAG_NDG, sds_ref);

   /*
   * Terminate access to the SDS and to the SD interface.
   */
   status_n = SDendaccess (sds_id);
   status_n = SDend (sd_id);

   /*
   * Terminate access to the vgroup and to the V interface, and
   * close the HDF file.
   */
   status_32 = Vdetach (vgroup_id);
   status_n = Vend (file_id);
   status_n = Hclose (file_id);
}
```

FORTRAN:

```
      program  add_SDS_to_a_vgroup
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*7  SDS_NAME
      character*9  VG_NAME
      character*13 VG_CLASS
C
      parameter (FILE_NAME = 'General_Vgroups.hdf',
     +           SDS_NAME  = 'Test SD',
     +           VG_NAME   = 'SD Vgroup',
     +           VG_CLASS  = 'Common Vgroups')
      integer DFACC_CREATE, DFACC_WRITE
      parameter (DFACC_CREATE = 4, DFACC_WRITE = 2)
      integer DFNT_INT32
      parameter (DFNT_INT32 = 24)
      integer DFTAG_NDG
      parameter (DFTAG_NDG = 720)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfsnam, vfscls, vfadtr, vfdtch, vfend
      integer sfstart, sfcreate, sfid2ref, sfendacc, sfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer vgroup_id
      integer sd_id, sds_id, sds_ref
      integer dim_sizes(1), rank
C
C**** End of variable declaration ************************************
C
C
C     Create the HDF file.
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)

C
C     Initialize SD interface.
C
      sd_id = sfstart(FILE_NAME, DFACC_WRITE)
C
C     Set the rank and the size of SDS's dimension.
C
      rank = 1
      dim_sizes(1) = 10
C
C     Create the SDS.
C
      sds_id = sfcreate(sd_id, SDS_NAME, DFNT_INT32, rank, dim_sizes)
C
C     Create a vgroup and set its name and class.
C
      vgroup_id = vfatch(file_id, -1 , 'w')
      status    = vfsnam(vgroup_id, VG_NAME)
      status    = vfscls(vgroup_id, VG_CLASS)
C
C     Obtain the reference number of the SDS using its identifier.
C
      sds_ref = sfid2ref(sds_id)
C
C     Add the SDS to the vgroup. Note: the tag DFTAG_NDG is used
C    when adding an SDS.  Refer to HDF Reference Manual, Section III, Table 3K,
C     for the entire list of tags.
C
      status = vfadtr(vgroup_id, DFTAG_NDG, sds_ref)
C
C     Terminate access to the SDS and to the SD interface.
C
      status = sfendacc(sds_id)
      status = sfend(sd_id)
C
C     Terminate access to the vgroup.
C
      status = vfdtch(vgroup_id)
C
C     Terminate access to the V interface and close the HDF file.
C
```

```
status = vfend(file_id)
status = hclose(file_id)
end
```

5.5.3. Inserting a Vdata or Vgroup Into a Vgroup: Vinsert

Vinsert is a routine designed specifically for inserting vdatas or vgroups into a parent vgroup. To use **Vinsert**, you must provide the identifier of the parent vgroup, *vgroup_id*, as well as the identifier of the vdata or vgroup to be inserted, *v_id*.

The parameter *v_id* of **Vinsert** is either a vdata identifier or a vgroup identifier, depending on whether a vdata or vgroup is to be inserted.

Vinsert returns the index of the inserted vdata or vgroup if the operation is successful, and `FAIL` (or `-1`) otherwise. The parameters of **Vinsert** are further defined in Table 5C.

5.5.4. Building a Vgroup with or without Elements: VHmakegroup

VHmakegroup is a high-level routine, designed to facilite the process of creating and inserting elements into a vgroup. The vgroup will have a name and/or class name if these information are provided to **VHmakegroup**. By using **VHmakegroup**, an application can by pass a number of function calls such as **Vattach**, **Vsetname**, **Vsetclass**, **Vinsert**/**Vaddtagref**, and **Vdetach**.

VHmakegroup creates a vgroup with the name specified by the parameter *vgroup_name* and the class name specified by the parameter *vgroup_class* in the file identified by the parameter *file_id*. The routine inserts *n_objects* objects into the vgroup. The tag and reference numbers of the objects to be inserted are specified in the arrays *tag_array* and *ref_array.*

Creating empty vgroups with **VHmakegroup** is allowed. **VHmakegroup** does not check if the tag/reference number pair is valid, or if the corresponding data object exists. However, all of the tag/reference number pairs must be unique.

Vstart must precede any calls to **VHmakegroup**.

The elements in the arrays *tag_array* and *ref_array* are the matching tag/reference number pairs of the objects to be inserted, that means *tag_array[0]* and *ref_array[0]* refer to one data object, and *tag_array[1]* and *ref_array[1]* to another, etc. If name and/or class name are not desired, the parameters *vgroup_name* and/or *vgroup_class* can be `NULL`.

The syntax of **VHmakegroup** is as follows:

```
C:          vgroup_ref = VHmakegroup(file_id, tag_array, ref_array, n_objects,
                                     vgroup_name, vgroup_class);

FORTRAN:    vgroup_ref = vhfmkgp(file_id, tag_array, ref_array, n_objects,
                                 vgroup_name, vgroup_class)
```

VHmakegroup returns the reference number of the newly-created vgroup if successful, `FAIL` (or `-1`) otherwise.

The parameters of **VHmakegroup** are further defined in Table 5F.

TABLE 5C

Vsetname, Vsetclass, Vaddtagref, Vinsert, and VHmakegroup Parameter Lists

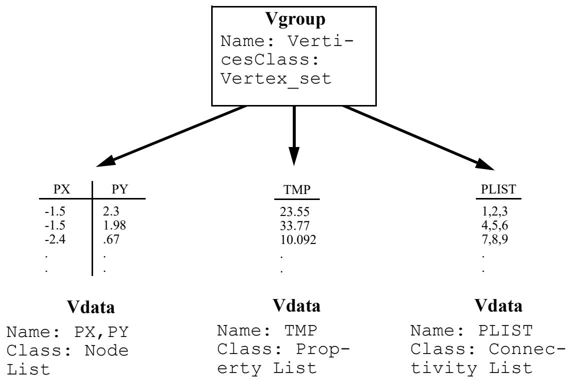
Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vsetname [int32] **(vfsnam)**	vgroup_id	int32	integer	Vgroup identifier
	vgroup_name	char *	character*(*)	Vgroup name
Vsetclass [int32] **(vfscls)**	vgroup_id	int32	integer	Vgroup identifier
	vgroup_class	char *	character*(*)	Vgroup class
Vaddtagref [int32] **(vfadtr)**	vgroup_id	int32	integer	Vgroup identifier
	obj_tag	int32	integer	Tag of the data object to be inserted
	obj_ref	int32	integer	Reference number of the data object to be inserted
Vinsert [int32] **(vfinsrt)**	vgroup_id	int32	integer	Vgroup identifier
	v_id	int32	integer	Identifier of the vgroup or vdata to be inserted
VHmakegroup [int32] **(vhfmkgp)**	file_id	int32	integer	File identifier
	tag_array	int32 *	integer(*)	Array of tags
	ref_array	int32 *	integer(*)	Array of reference numbers
	n_objects	int32	integer	Number of items in tag_array or ref_array (must be the same)
	vgroup_name	char *	character*(*)	Name of the vgroup
	vgroup_class	char *	character*(*)	Class of the vgroup

EXAMPLE 3.

Adding Three Vdatas into a Vgroup

This example illustrates the use of **Vinsert/vfinsrt** to add a vdata to a vgroup. Note that **Vaddtagref/vfadtrf**, used in the previous example, performs the same task and only differs in the argument list.

In this example, the program creates three vdatas and a vgroup in the existing HDF file "General_Vgroups.hdf" then adds the three vdatas to the vgroup. Notice that the vdatas and the vgroup are created in the same interface that is initialized by the call **Vstart/vfstart**. The first vdata is named "X,Y Coordinates" and has two order-1 fields of type float32. The second vdata is named "Temperature" and has one order-1 field of type float32. The third vdata is named "Node List" and has one order-3 field of type int16. The vgroup is named "Vertices" and is of class "Vertex Set". The program uses **Vinsert/vfinsrt** to add the vdatas to the vgroup using the vdata identifiers. Refer to Chapter 4, *Vdatas (VS API)*, for the discussion of the VS routines used in this example.



C:

```
#include "hdf.h"

#define  FILE_NAME         "General_Vgroups.hdf"
#define  N_RECORDS         30       /* number of records in the vdatas */
#define  ORDER             3        /* order of field FIELD_VD2 */
#define  VG_NAME           "Vertices"
#define  VG_CLASS          "Vertex Set"
#define  VD1_NAME          "X,Y Coordinates"   /* first vdata to hold X,Y...*/
#define  VD1_CLASS         "Position"          /*...values of the vertices */
#define  VD2_NAME          "Temperature"       /* second vdata to hold the...*/
#define  VD2_CLASS         "Property List"     /*...temperature field */
#define  VD3_NAME          "Node List"         /* third vdata to hold...*/
#define  VD3_CLASS         "Mesh"              /*...the list of nodes */
#define  FIELD1_VD1        "PX"    /* first field of first vdata - X values */
#define  FIELD2_VD1        "PY"/* second field of first vdata - Y values */
#define  FIELD_VD2         "TMP"/* field of third vdata */
#define  FIELD_VD3         "PLIST"/* field of second vdata */
#define  FIELDNAME_LIST    "PX,PY" /* field name list for first vdata */
/* Note that the second and third vdatas can use the field names as
   the field name lists unless more fields are added to a vdata.
   Then a field name list is needed for that vdata */

main( )
{
   /************************* Variable declaration *************************/

   intn     status_n;   /* returned status for functions returning an intn  */
   int32    status_32,  /* returned status for functions returning an int32 */
            file_id, vgroup_id,
            vdata1_id, vdata2_id, vdata3_id;
   int32    num_of_records,          /* number of records actually written */
            vd_index;                /* position of a vdata in the vgroup  */
   int8     i, j, k = 0;
   float32  pxy[N_RECORDS][2] =      /* buffer for data of the first vdata */
                {-1.5, 2.3, -1.5, 1.98, -2.4, .67,
                -3.4, 1.46, -.65, 3.1, -.62, 1.23,
                -.4, 3.8, -3.55, 2.3, -1.43, 2.44,
                .23, 1.13, -1.4, 5.43, -1.4, 5.8,
                -3.4, 3.85, -.55, .3, -.21, 1.22,
                -1.44, 1.9, -1.4, 2.8, .94, 1.78,
                -.4, 2.32, -.87, 1.99, -.54, 4.11,
                -1.5, 1.35, -1.4, 2.21, -.22, 1.8,
                -1.1, 4.55, -.44, .54, -1.11, 3.93,
                -.76, 1.9, -2.34, 1.7, -2.2, 1.21};
   float32  tmp[N_RECORDS];          /* buffer for data of the second vdata */
   int16    plist[N_RECORDS][3];     /* buffer for data of the third vdata */

   /********************** End of variable declaration ***********************/

   /*
   * Open the HDF file for writing.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Buffer the data for the second and third vdatas.
   */
   for (i = 0; i < N_RECORDS; i++)
      for (j = 0; j < ORDER; j++)
         plist[i][j] = ++k;

   for (i = 0; i < N_RECORDS; i++)
      tmp[i] = i * 10.0;

   /*
   * Create the vgroup then set its name and class.  Note that the vgroup's
   * reference number is set to -1 for creating and the access mode is "w" for
   * writing.
   */
   vgroup_id = Vattach (file_id, -1, "w");
   status_32 = Vsetname (vgroup_id, VG_NAME);
   status_32 = Vsetclass (vgroup_id, VG_CLASS);

   /*
   * Create the first vdata then set its name and class. Note that the vdata's
   * reference number is set to -1 for creating and the access mode is "w" for
   * writing.
   */
   vdata1_id = VSattach (file_id, -1, "w");
   status_32 = VSsetname (vdata1_id, VD1_NAME);
   status_32 = VSsetclass (vdata1_id, VD1_CLASS);

   /*
   * Introduce and define the fields of the first vdata.
   */
   status_n = VSfdefine (vdata1_id, FIELD1_VD1, DFNT_FLOAT32, 1);
   status_n = VSfdefine (vdata1_id, FIELD2_VD1, DFNT_FLOAT32, 1);
   status_n = VSsetfields (vdata1_id, FIELDNAME_LIST);

   /*
   * Write the buffered data into the first vdata with full interlace mode.
   */
   num_of_records = VSwrite (vdata1_id, (uint8 *)pxy, N_RECORDS,
                             FULL_INTERLACE);

   /*
   * Insert the vdata into the vgroup using its identifier.
   */
   vd_index = Vinsert (vgroup_id, vdata1_id);

   /*
```

```
   * Detach from the first vdata.
   */
   status_32 = VSdetach (vdata1_id);

   /*
   * Create, write, and insert the second vdata to the vgroup using
   * steps similar to those used for the first vdata.
   */
   vdata2_id = VSattach (file_id, -1, "w");
   status_32 = VSsetname (vdata2_id, VD2_NAME);
   status_32 = VSsetclass (vdata2_id, VD2_CLASS);
   status_n = VSfdefine (vdata2_id, FIELD_VD2, DFNT_FLOAT32, 1);
   status_n = VSsetfields (vdata2_id, FIELD_VD2);
   num_of_records = VSwrite (vdata2_id, (uint8 *)tmp, N_RECORDS,
                             FULL_INTERLACE);
   vd_index = Vinsert (vgroup_id, vdata2_id);
   status_32 = VSdetach (vdata2_id);

   /*
   * Create, write, and insert the third vdata to the vgroup using
   * steps similar to those used for the first and second vdatas.
   */
   vdata3_id = VSattach (file_id, -1, "w");
   status_32 = VSsetname (vdata3_id, VD3_NAME);
   status_32 = VSsetclass (vdata3_id, VD3_CLASS);
   status_n = VSfdefine (vdata3_id, FIELD_VD3, DFNT_INT16, 3);
   status_n = VSsetfields (vdata3_id, FIELD_VD3);
   num_of_records = VSwrite (vdata3_id, (uint8 *)plist, N_RECORDS,
                             FULL_INTERLACE);
   vd_index = Vinsert (vgroup_id, vdata3_id);
   status_32 = VSdetach (vdata3_id);

   /*
   * Terminate access to the vgroup "Vertices".
   */
   status_32 = Vdetach (vgroup_id);

   /*
   * Terminate access to the V interface and close the HDF file.
   */
   status_n = Vend (file_id);
   status_n = Hclose (file_id);
}
```

FORTRAN:

```
      program  add_vdatas_to_a_vgroup
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*8  VG_NAME
      character*10 VG_CLASS
      character*15 VD1_NAME
      character*8  VD1_CLASS
      character*11 VD2_NAME
      character*13 VD2_CLASS
      character*9  VD3_NAME
      character*4  VD3_CLASS
C
      parameter (FILE_NAME = 'General_Vgroups.hdf',
     +           VG_NAME   = 'Vertices',
     +           VG_CLASS  = 'Vertex Set')
      parameter (VD1_NAME  = 'X,Y Coordinates',
     +           VD2_NAME  = 'Temperature',
     +           VD3_NAME  = 'Node List')
      parameter (VD1_CLASS = 'Position',
     +           VD2_CLASS = 'Property List',
     +           VD3_CLASS = 'Mesh')
      character*2 FIELD1_VD1
      character*2 FIELD2_VD1
      character*3 FIELD_VD2
      character*4 FIELD_VD3
      character*5 FIELDNAME_LIST
      parameter (FIELD1_VD1 = 'PX',
     +           FIELD2_VD1 = 'PY',
     +           FIELD_VD2  = 'TMP',
     +           FIELD_VD3  = 'PLIST',
     +           FIELDNAME_LIST = 'PX,PY')
      integer N_RECORDS
      parameter (N_RECORDS = 30)

      integer  DFACC_WRITE
      parameter (DFACC_WRITE = 2)
      integer DFNT_FLOAT32, DFNT_INT16
      parameter (DFNT_FLOAT32 = 5, DFNT_INT16 = 22)
      integer FULL_INTERLACE
      parameter (FULL_INTERLACE = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfsnam, vfscls, vfinsrt, vfdtch, vfend
      integer vsfatch, vsfsnam, vsfscls, vsffdef, vsfsfld,
     +        vsfwrt, vsfwrtc, vsfdtch

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer vgroup_id
      integer vdata1_id, vdata2_id, vdata3_id, vd_index
      integer num_of_records
      integer i, j, k
      real    pxy(2,N_RECORDS), tmp(N_RECORDS)
      integer plist(3,N_RECORDS)
      data pxy /-1.5, 2.3, -1.5, 1.98, -2.4, .67,
     +          -3.4, 1.46, -.65, 3.1, -.62, 1.23,
     +          -.4, 3.8, -3.55, 2.3, -1.43, 2.44,
     +          .23, 1.13, -1.4, 5.43, -1.4, 5.8,
     +          -3.4, 3.85, -.55, .3, -.21, 1.22,
     +          -1.44, 1.9, -1.4, 2.8, .94, 1.78,
     +          -.4, 2.32, -.87, 1.99, -.54, 4.11,
     +          -1.5, 1.35, -1.4, 2.21, -.22, 1.8,
     +          -1.1, 4.55, -.44, .54, -1.11, 3.93,
     +          -.76, 1.9, -2.34, 1.7, -2.2, 1.21/
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
```

```
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Buffer the data for the third and second vdatas.
C
      do 20 i = 1, N_RECORDS
         do 10 j = 1, 3
            plist(j,i) = k
            k = k+1
10       continue
20    continue
      do 30 i = 1, N_RECORDS
         tmp(i) = (i-1) * 10.0
30    continue
C
C     Create a vgroup and set its name and class.
C     Note that the vgroup’s reference number is set to -1 for creating
C     and the access mode is ’w’ for writing.
C
      vgroup_id = vfatch(file_id, -1 , ’w’)
      status    = vfsnam(vgroup_id, VG_NAME)
      status    = vfscls(vgroup_id, VG_CLASS)
C
C     Create the first vdata then set its name and class. Note that the vdata’s
C     reference number is set to -1 for creating and the access mode is ’w’ for
C     writing.
C
      vdata1_id = vsfatch(file_id, -1, ’w’)
      status = vsfsnam(vdata1_id, VD1_NAME)
      status = vsfscls(vdata1_id, VD1_CLASS)
C
C     Introduce and define the fields of the first vdata.
C
      status = vsffdef(vdata1_id, FIELD1_VD1, DFNT_FLOAT32, 1)
      status = vsffdef(vdata1_id, FIELD2_VD1, DFNT_FLOAT32, 1)
      status = vsfsfld(vdata1_id, FIELDNAME_LIST)
C
C     Write the buffered data into the first vdata.
C
      num_of_records = vsfwrt(vdata1_id, pxy, N_RECORDS,
     +                        FULL_INTERLACE)
C
C     Insert the vdata into the vgroup using its identifier.
C
      vd_index = vfinsrt(vgroup_id, vdata1_id)
C
C     Detach from the first vdata.
C
      status = vsfdtch(vdata1_id)
C
C     Create, write, and insert the second vdata to the vgroup using
C     steps similar to those used for the first vdata.
C
      vdata2_id = vsfatch(file_id, -1, ’w’)
      status = vsfsnam(vdata2_id, VD2_NAME)
      status = vsfscls(vdata2_id, VD2_CLASS)
      status = vsffdef(vdata2_id, FIELD_VD2, DFNT_FLOAT32, 1)
      status = vsfsfld(vdata2_id, FIELD_VD2)
      num_of_records = vsfwrt(vdata2_id, tmp, N_RECORDS,
     +                        FULL_INTERLACE)
      vd_index = vfinsrt(vgroup_id, vdata2_id)
      status = vsfdtch(vdata2_id)
```

```
C
C     Create, write, and insert the third vdata to the vgroup using
C     steps similar to those used for the first and second vdatas.
C
      vdata3_id = vsfatch(file_id, -1, 'w')
      status = vsfsnam(vdata3_id, VD3_NAME)
      status = vsfscls(vdata3_id, VD3_CLASS)
      status = vsffdef(vdata3_id, FIELD_VD3, DFNT_INT16, 3)
      status = vsfsfld(vdata3_id, FIELD_VD3)
      num_of_records = vsfwrtc(vdata3_id, plist, N_RECORDS,
     +                         FULL_INTERLACE)
      vd_index = vfinsrt(vgroup_id, vdata3_id)
      status = vsfdtch(vdata3_id)

C
C     Terminate access to the vgroup 'Vertices'.
C
      status = vfdtch(vgroup_id)
C
C     Terminate access to the V interface and close the HDF file.
C
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

5.6. Reading from Vgroups

Reading from vgroups is more complicated than writing to vgroups. The process of reading from vgroups involves two steps: locating the appropriate vgroup and obtaining information about the member or members of a vgroup. This section describes routines that provide these functionalities.

5.6.1. Locating Vgroups and Obtaining Vgroup Information

There are several routines provided for the purpose of locating a particular vgroup, each corresponding to an identifying aspect of a vgroup. These aspects include whether the vgroup has vgroups included in it, the identification of the vgroup in the file based on its reference number, and the name and class name of the vgroup. The routines are described in the following subsections.

5.6.1.1. Locating Lone Vgroups: Vlone

A ***lone vgroup*** is one that is not a member of any other vgroups, i.e., not linked with any other vgroups. **Vlone** searches the file specified by the parameter *file_id* and retrieves the reference numbers of lone vgroups in the file. This routine is useful for locating unattached vgroups in a file or the vgroups at the top of a grouping hierarchy. The syntax of **Vlone** is as follows:

```
C:         num_of_lones = Vlone(file_id, ref_array, maxsize);

FORTRAN:   num_of_lones = vflone(file_id, ref_array, maxsize)
```

The parameter *ref_array* is an array allocated to hold the reference numbers of the found vgroups. The argument *maxsize* specifies the maximum size of *ref_array*. At most *maxsize* reference numbers will be retrieved in *ref_array*. The value of *max_size*, the space allocated for *ref_array*, depends on how many lone vgroups are expected to be found.

To use dynamic memory instead of allocating an unnecessarily large array (i.e., one that will hold the maximum possible number of reference numbers), call **Vlone** twice. In the first call to **Vlone**, set *maxsize* to a small value, for example, 0 or 1, then use the returned value (the total number of lone vgroups in the file) to allocate memory for *ref_array*. This array is then passed into the second call to **Vlone**.

Vlone returns the total number of lone vgroups or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5D.

5.6.1.2. Sequentially Searching for a Vgroup: Vgetid

Vgetid sequentially searches through an HDF file to obtain the reference number of the vgroup immediately following the vgroup specified by the reference number, *vgroup_ref*. The syntax of **Vgetid** is as follows:

```
C:          ref_num = Vgetid(file_id, vgroup_ref);

FORTRAN:    ref_num = vfgid(file_id, vgroup_ref)
```

To initiate a search, **Vgetid** may be called with *vgroup_ref* set to `-1`. Doing so returns the reference number of the first vgroup in the file. Any attempt to search past the last vgroup in a file will cause **Vgetid** to return a value of `FAIL` (or `-1`).

Vgetid returns a vgroup reference number or `FAIL` (or `-1`). The parameters of **Vgetid** are further defined in Table 5D.

5.6.1.3. Retrieving vgroups in a file or in a vgroup: Vgetvgroups

Vgetvgroups retrieves a list containing reference numbers of vgroups in a file or in a vgroup, which is identified by the parameter *id*. The syntax of **Vgetvgroups** is as follows:

```
C:          status = Vgetvgroups(id, start_vgroup, vgroup_count, refarray);

FORTRAN:    status = vfgvgroups(id, start_vg, vg_count, refarray)
```

The library commonly use vgroups or vdatas to store HDF objects. For example, a vgroup is used to represent an SDS and a vdata for an attribute. **Vgetvgroups** retrieves only the vgroups that were previously created by user applications, not those that were created by the library internally. They are referred to as user-created vgroups, for brevity.

When *id* is a vgroup identifier, only the immediate sub-vgroups will be retrieved; that is, the sub-vgroups will not be traversed.

The parameter *vgroup_count* specifies the number of values that the refarray list can hold and can be any positive number smaller than `MAX_REF` (`65535`). If *vgroup_count* is larger than the actual number of user-created vgroups, then only the actual number of user-created vgroups will be retrieved.

The retrieval starts at the vgroup number *start_vgroup* going forward in the order which the vgroups were created. For example, if there are 100 vgroups that can be retrieved, specifying *start_vgroup* as 90 and *vgroup_count* as 10 will retrieve the last ten vgroups. The value for *start_vgroup* must be non-negative and smaller than or equal to the number of user-created vgroups, which can be obtained by invoking **Vgetvgroups** passing in `NULL` for the array *refarray*. This number of user-created vgroups will also allow applications to sufficiently allocate space for *refarray*.

- When *start_vgroup* is 0, the retrieval will start at the beginning of the file or the first sub-vgroup of the specified vgroup.
- When *start_vgroup* is smaller than the number of user-created vgroups in the file or the specified vgroup, **Vgetvgroups** will start retrieving vgroups from the vgroup number *start_vgroup*.

- When *start_vgroup* is greater than the number of user-created vgroups in the file or the vgroup, **Vgetvgroups** will return `FAIL` (or `-1`).

Following are some examples of using **Vgetvgroups** to get the reference numbers of vgroups in a file, assuming that the file has been opened for reading successfully:

```
C:          /* Call Vgetvgroups the first time to get the number of vgroups in
               the file to allocate refarray */
            n_vgs = Vgetvgroups(file_id, 0, 0, NULL);

            /* Allocate space to retrieve reference numbers of n_vgs vgroups */
            refarray = (uint16 *)HDmalloc(sizeof(uint16)*n_vgs);

            /* To get all the vgroups in the file: */
            n_vgs = Vgetvgroups(file_id, 0, n_vgs, refarray);

            /* Assuming n_vgs=100, to get the first 10 vgroups in the file: */
            n_vgs = Vgetvgroups(file_id, 0, 10, refarray);

            /* Assuming n_vgs=100, to get the last 10 vgroups in the file: */
            n_vgs = Vgetvgroups(file_id, 90, 10, refarray);
```

Following are some examples of using **Vgetvgroups** to get the reference numbers of vgroups in a parent vgroup:

```
C:          vgroup_id = Vattach(file_id, vgroup_ref, "r");
            /* Call Vgetvgroups the first time to get the number of vgroups in
               the parent vgroup to allocate refarray */
            n_vgs = Vgetvgroups(vgroup_id, 0, 0, NULL);

            /* Allocate space to retrieve reference numbers of n_vgs vgroups */
            refarray = (uint16 *)HDmalloc(sizeof(uint16)*n_vgs);

            /* Get all the vgroups in the parent vgroup */
            n_vgs = Vgetvgroups(vgroup_id, 0, n_vgs, refarray);

            /* Close the vgroup */
            status = Vdetach(vgroup_id);
```

Note that, in the FORTRAN-77 version, if *vg_count* is `-1` then the function will return the number of user-created vgroups and disregard *refarray*; equivalent to passing `NULL` for *refarray* in the C version.

Vgetvgroups returns the number of user-created vgroups retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of this routine are further defined in Table 5D.

5.6.1.4. Determining Internal Vgroup: Vgisinternal

The HDF library commonly uses vgroups and vdatas to store metadata or data for the library's own use. For examples, vgroups are used to represent SDS or raster images, and vdatas are used to store attributes or dimensions. Typically, a user is only interested in vgroups/vdatas that were created by user applications, not by the library internally. **Vgisinternal** allows an application to find out if a vgroup is internally created.

The syntax of **Vgisinternal** is as follows:

```
C:          is_internal = Vgisinternal(vgroup_id);

FORTRAN:    Currently unavailable
```

Vgisinternal checks the class name of the given vgroup against the list `HDF_INTERNAL_VGS` to determine whether the vgroup was previously created by the library instead of by a user application. The names in `HDF_INTERNAL_VGS` are included:

```
_HDF_VARIABLE ("Var0.0")
_HDF_DIMENSION ("Dim0.0")
_HDF_UDIMENSION ("UDim0.0")
_HDF_CDF ("CDF0.0")
GR_NAME ("RIG0.0")
RI_NAME ("RI0.0")
```

There is one special case where an internal vgroup having a null class name and a name as `GR_NAME`. This should be extremely rare, yet it is a possibility.

Vgisinternal returns `TRUE` (`1`) if the inquired vgroup is one that was internally created by the library, `FALSE` (`0`) otherwise, and `FAIL` (`-1`) if failure occurs. The parameters of this routine are further defined in Table 5D.

TABLE 5D **Vlone, Vgetid, Vgetvgroups, and Vgisinternal Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Vlone [int32] **(vflone)**	file_id	int32	integer	File identifier
	ref_array	int32 *	integer (*)	Buffer for the reference numbers of lone vgroups
	maxsize	int32	integer	Maximum number of vgroups to store in `ref_array`
Vgetid [int32] **(vfgid)**	file_id	int32	integer	File identifier
	vgroup_ref	int32	integer	Reference number of the current vgroup
Vgetvgroups [intn] **(vfgvgroups)**	id	int32	integer	File or vgroup identifer
	start_vgroup	uintn	integer	Vgroup index to start retrieving at
	vgroup_count	uintn	integer	Number of vgroups to be retrieved
	refarray	int32 *	integer (*)	Array to hold reference numbers of retrieved vgroups
Vgisinternal [intn] **(unavailable)**	vgroup_id	int32	N/A	Vgroup identifier

5.6.1.5. Obtaining the Name of a Vgroup: Vgetname

Vgetname retrieves the name of the vgroup identified by the parameter *vgroup_id* into the parameter *vgroup_name*. The syntax of **Vgetname** is as follows:

```
C:          status = Vgetname(vgroup_id, vgroup_name);

FORTRAN:    status = vfgnam(vgroup_id, vgroup_name)
```

Starting from release 4.2.4, the maximum length of vgroup's name is no longer limited to `VGNAMELENMAX` (or `64`). When an application attempts to read a vgroup's name that is longer than 64 characters with an insufficient buffer, the result will be unpredictable. Applications can use **Vgetnamelen** to get the length of the vgroup's name prior to calling **Vgetname**.

Vgetname returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5E.

5.6.1.6. Obtaining the Length of a Vgroup's Name: Vgetnamelen

Vgetnamelen retrieves the length of a vgroup's name and stores it in the parameter *name_len*. The vgroup is identified by the parameter *vgroup_id* . The syntax of **Vgetnamelen** is as follows:

```
C:          status = Vgetnamelen(vgroup_id, name_len);

FORTRAN:    Currently unavailable
```

Vgetnamelen returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5E.

5.6.1.7. Obtaining the Class Name of a Vgroup: Vgetclass

Vgetclass retrieves the class name of the vgroup specified by the parameter *vgroup_id* into the parameter *vgroup_class*. The syntax of **Vgetclass** is as follows:

```
C:          status = Vgetclass(vgroup_id, vgroup_class);

FORTRAN:    status = vfgcls(vgroup_id, vgroup_class)
```

Starting from release 4.2.5, the maximum length of vgroup's class name is no longer limited to `VGNAMELENMAX` (or `64`). When an application attempts to read a vgroup's name that is longer than 64 characters with an insufficient buffer, the result will be unpredictable. Applications can use **Vgetclassnamelen** to get the length of the vgroup's class name prior to calling **Vgetclass**.

Vgetclass returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5E.

5.6.1.8. Obtaining the Length of a Vgroup's Class Name: Vgetclassnamelen

Vgetclassnamelen retrieves the length of a vgroup's class name and stores it in the parameter *classname_len*. The vgroup is identified by the parameter *vgroup_id* . The syntax of **Vgetclassnamelen** is as follows:

```
C:          status = Vgetclassnamelen(vgroup_id, classname_len);

FORTRAN:    Currently unavailable
```

Vgetclassnamelen returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5E.

TABLE 5E **Vgetname, Vgetnamelen, Vgetclass, and Vgetclassnamelen Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vgetname [int32] **(vfgnam)**	vgroup_id	int32	integer	Vgroup identifier
	vgroup_name	char *	character*(*)	Buffer for the name of the vgroup
Vgetnamelen [int32] **(unavailable)**	vgroup_id	int32	N/A	Vgroup identifier
	name_len	uint16*	N/A	Buffer for the length of the vgroup's name
Vgetclass [int32] **(vfgcls)**	vgroup_id	int32	integer	Vgroup identifier
	vgroup_class	char *	character*(*)	Buffer for the vgroup class
Vgetclassnamelen [int32] **(unavailable)**	vgroup_id	int32	N/A	Vgroup identifier
	classname_len	uint16*	N/A	Buffer for the length of the vgroup's class name

5.6.1.9. Locating a Vgroup Given Its Name: Vfind

Vfind searches the file identified by *file_id* for a vgroup with the name specified by the parameter *vgroup_name*. The syntax for **Vfind** is as follows:

```
C:          vgroup_ref = Vfind(file_id, vgroup_name);

FORTRAN:   vgroup_ref = vfind(file_id, vgroup_name)
```

Vfind returns the reference number of the vgroup if one is found, or `0` otherwise. If more than one vgroup has the same name, **Vfind** will return the reference number of the first one.

The parameters of **Vfind** are further defined in Table 5F.

5.6.1.10. Locating a Vgroup Given Its Class Name: Vfindclass

Vfindclass searches the file identified by *file_id* for a vgroup with the class name specified by the parameter *vgroup_class*. The syntax of **Vfindclass** is as follows:

```
C:          vgroup_ref = Vfindclass(file_id, vgroup_class);

FORTRAN:   vgroup_ref = vfndcls(file_id, vgroup_class)
```

Vfindclass returns the reference number of the vgroup if one is found, or `0` otherwise. If more than one vgroup has the same class name, **Vfindclass** will return the reference number of the first one.

The parameters of **Vfindclass** are further defined in Table 5F.

TABLE 5F

Vfind and Vfindclass Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vfind [int32] **(vfind)**	file_id	int32	integer	File identifier
	vgroup_name	char *	character*(*)	Buffer for the name of the vgroup
Vfindclass [int32] **(vfndcls)**	file_id	int32	integer	File identifier
	vgroup_class	char *	character*(*)	Buffer for the vgroup class

EXAMPLE 4.

Obtaining Information about Lone Vgroups

This example illustrates the use of **Vlone/vflone** to obtain the list of reference numbers of all lone vgroups in the file and the use of **Vgetname/vfgnam** and **Vgetclass/vfgcls** to obtain the name and the class of a vgroup.

In this example, the program calls **Vlone/vflone** twice. The first call is to obtain the number of lone vgroups in the file so that sufficient space can be allocated; the later call is to obtain the actual reference numbers of the lone vgroups. The program then goes through the list of lone vgroup reference numbers to get and display the name and class of each lone vgroup. The file used in this example is "General_Vgroups.hdf".

C:

```
#include "hdf.h"

#define  FILE_NAME   "General_Vgroups.hdf"

main( )
{
   /************************* Variable declaration **************************/

   intn   status_n;     /* returned status for functions returning an intn  */
   int32  status_32,    /* returned status for functions returning an int32 */
          file_id, vgroup_id;
   int32  lone_vg_number,      /* current lone vgroup number */
          num_of_lones = 0;    /* number of lone vgroups */
   int32 *ref_array;    /* buffer to hold the ref numbers of lone vgroups   */
   char   vgroup_name[VGNAMELENMAX], vgroup_class[VGNAMELENMAX];

   /********************** End of variable declaration **********************/

   /*
   * Open the HDF file for reading.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Get and print the names and class names of all the lone vgroups.
   * First, call Vlone with num_of_lones set to 0 to get the number of
   * lone vgroups in the file, but not to get their reference numbers.
   */
   num_of_lones = Vlone (file_id, NULL, num_of_lones );

   /*
   * Then, if there are any lone vgroups,
   */
   if (num_of_lones > 0)
   {
      /*
      * use the num_of_lones returned to allocate sufficient space for the
      * buffer ref_array to hold the reference numbers of all lone vgroups,
      */
      ref_array = (int32 *) malloc(sizeof(int32) * num_of_lones);

      /*
      * and call Vlone again to retrieve the reference numbers into
      * the buffer ref_array.
      */
      num_of_lones = Vlone (file_id, ref_array, num_of_lones);

      /*
      * Display the name and class of each lone vgroup.
      */
      printf ("Lone vgroups in this file are:\n");
      for (lone_vg_number = 0; lone_vg_number < num_of_lones;
                                                         lone_vg_number++)
      {
         /*
         * Attach to the current vgroup then get and display its
         * name and class. Note: the current vgroup must be detached before
         * moving to the next.
         */
         vgroup_id = Vattach (file_id, ref_array[lone_vg_number], "r");
         status_32 = Vgetname (vgroup_id, vgroup_name);
         status_32 = Vgetclass (vgroup_id, vgroup_class);
         printf ("   Vgroup name %s and class %s\n", vgroup_name,
                     vgroup_class);
         status_32 = Vdetach (vgroup_id);
      } /* for */
   } /* if */

   /*
   * Terminate access to the V interface and close the file.
   */
   status_n = Vend (file_id);
   status_n = Hclose (file_id);

   /*
   * Free the space allocated by this program.
   */
   free (ref_array);
}
```

FORTRAN:

```
      program  getinfo_about_vgroup
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
C
      parameter (FILE_NAME = 'General_Vgroups.hdf')
      integer DFACC_READ
      parameter (DFACC_READ = 1)
      integer SIZE
      parameter(SIZE = 10)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfgnam, vfgcls, vflone, vfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer vgroup_id
      integer lone_vg_number, num_of_lones
      character*64 vgroup_name, vgroup_class
      integer ref_array(SIZE)
      integer i
C
C**** End of variable declaration ************************************
C
C
C     Initialize ref_array.
C
      do 10 i = 1, SIZE
         ref_array(i) = 0
10    continue
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Get and print the name and class name of all lone vgroups.
C     First, call vflone with num_of_lones set to 0 to get the number of
C     lone vgroups in the file and check whether size of ref_array is
C     big enough to hold reference numbers of ALL lone groups.
C     If ref_array is not big enough, exit the program after displaying an
C     informative message.
C
      num_of_lones = 0
      num_of_lones = vflone(file_id, ref_array, num_of_lones)
      if (num_of_lones .gt. SIZE) then
      write(*,*) num_of_lones, 'lone vgroups is found'
      write(*,*) 'increase the size of ref_array to hold reference '
      write(*,*) 'numbers of all lone vgroups in the file'
      stop
      endif
C
C     If there are any lone groups in the file,
C
      if (num_of_lones .gt. 0) then
C
C     call vflone again to retrieve the reference numbers into ref_array.
C
      num_of_lones = vflone(file_id, ref_array, num_of_lones)
C
C     Display the name and class of each vgroup.
```

```
C
      write(*,*) 'Lone vgroups in the file are:'

      do 20 lone_vg_number = 1, num_of_lones
C
C     Attach to the current vgroup, then get and display its name and class.
C     Note: the current vgroup must be detached before moving to the next.
C
      vgroup_name = ' '
      vgroup_class = ' '
      vgroup_id = vfatch(file_id, ref_array(lone_vg_number), 'r')
      status    = vfgnam(vgroup_id, vgroup_name)
      status    = vfgcls(vgroup_id, vgroup_class)
      write(*,*) 'Vgroup name ' ,  vgroup_name
      write(*,*) 'Vgroup class ' , vgroup_class
      write(*,*)
      status = vfdtch(vgroup_id)
20    continue

      endif
C
C     Terminate access to the V interface and close the HDF file.
C
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

5.6.2. Obtaining Information about the Contents of a Vgroup

This section describes the Vgroup interface routines that allow the user to obtain various information about the contents of vgroups.

5.6.2.1. Obtaining the Number of Objects in a Vgroup: Vntagrefs

Vntagrefs returns the number of tag/reference number pairs (i.e., the number of vgroup members) stored in the specified vgroup. The syntax of **Vntagrefs** is as follows:

```
C:          num_of_tagrefs = Vntagrefs(vgroup_id);

FORTRAN:    num_of_tagrefs = vfntr(vgroup_id)
```

Vntagrefs can be used together with **Vgettagrefs** or **Vgettagref** to identify the data objects linked to a given vgroup.

Vntagrefs returns 0 or a positive number representing the number of HDF data objects linked to the vgroup if successful, or `FAIL` (or `-1`) otherwise. The parameter of **Vntagrefs** is further defined in Table 5G.

5.6.2.2. Obtaining the Tag/Reference Number Pair of a Data Object within a Vgroup : Vgettagref

Vgettagref retrieves the tag/reference number pair of a specified data object stored within the vgroup identified by the parameter *vgroup_id*. The syntax of **Vgettagref** is as follows:

```
C:          status = Vgettagref(vgroup_id, index, &obj_tag, &obj_ref);

FORTRAN:    status = vfgttr(vgroup_id, index, obj_tag, obj_ref)
```

Vgettagref stores the tag and reference number in the parameters *obj_tag* and *obj_ref*, respectively. The parameter *index* specifies the location of the data object within the vgroup and is zero-based.

Often, this routine is called in a loop to identify the tag/reference number pair of each data object belong to a vgroup. In this case, **Vntagrefs** is used to obtain the loop boundary.

Vgettagref returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5G.

5.6.2.3. Obtaining the Tag/Reference Number Pairs of Data Objects in a Vgroup: Vgettagrefs

Vgettagrefs retrieves the tag/reference number pairs of the members of a vgroup and returns the number of pairs retrieved. The syntax of **Vgettagrefs** is as follows:

```
C:         num_of_pairs = Vgettagrefs(vgroup_id, tag_array, ref_array, max-
                              size);

FORTRAN:   num_of_pairs = vfgttrs(vgroup_id, tag_array, ref_array, maxsize)
```

Vgettagrefs stores the tags into the array *tag_array* and the reference numbers into the array *ref_array*. The parameter *maxsize* specifies the maximum number of tag/reference number pairs to return, therefore each array must be at least *maxsize* in size.

Vgettagrefs can be used to obtain the value of *maxsize* if the tag/reference number pairs for all members of the vgroup are desired. To do this, set *maxsize* to 1 in the first call to **Vgettagrefs**.

Vgettagrefs returns the number of tag/reference number pairs or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5G.

TABLE 5G **Vntagrefs, Vgettagref, and Vgettagrefs Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Vntagrefs [int32] **(vfntr)**	vgroup_id	int32	integer	Vgroup identifier
Vgettagref [intn] **(vfgttr)**	vgroup_id	int32	integer	Vgroup identifier
	index	int32	integer	Index of the tag/reference number pair to be retrieved
	obj_tag	int32 *	integer	Tag of the data object
	obj_ref	int32 *	integer	Reference number of the data object
Vgettagrefs [int32] **(vfgttrs)**	vgroup_id	int32	integer	Vgroup identifier
	tag_array	int32 []	integer (*)	Buffer for the returned tags
	ref_array	int32 []	integer (*)	Buffer for the returned reference numbers
	maxsize	int32	integer	Maximum number of tag/reference number pairs to be returned

EXAMPLE 5. **Obtaining Information about the Contents of a Vgroup**

This example illustrates the use of **Vgetid/vfgid** to get the reference number of a vgroup, **Vntagrefs/vfntr** to get the number of HDF data objects in the vgroup, **Vgettagref/vfgttr** to get the tag/

reference number pair of a data object within the vgroup, and **Visvg/vfisvg** and **Visvs/vfisvs** to determine whether a data object is a vgroup and a vdata, respectively.

In the example, the program traverses the HDF file "General_Vgroups.hdf" from the beginning and obtains the reference number of each vgroup so it can be attached. Once a vgroup is attached, the program gets the total number of tag/reference number pairs in the vgroup and displays some information about the vgroup. The information displayed includes the position of the vgroup in the file, the tag/reference number pair of each of its data objects, and the message stating whether the object is a vdata, vgroup, or neither.

C:

```
#include "hdf.h"

#define   FILE_NAME        "General_Vgroups.hdf"

main( )
{
   /************************* Variable declaration **************************/

   intn   status_n;     /* returned status for functions returning an intn  */
   int32  status_32,    /* returned status for functions returning an int32 */
          file_id, vgroup_id, vgroup_ref,
          obj_index,    /* index of an object within a vgroup */
          num_of_pairs, /* number of tag/ref number pairs, i.e., objects */
          obj_tag, obj_ref,     /* tag/ref number of an HDF object */
          vgroup_pos = 0;       /* position of a vgroup in the file */

   /********************** End of variable declaration **********************/

   /*
   * Open the HDF file for reading.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Obtain each vgroup in the file by its reference number, get the
   * number of objects in the vgroup, and display the information about
   * that vgroup.
   */
   vgroup_ref = -1;        /* set to -1 to search from the beginning of file */
   while (TRUE)
   {
      /*
      * Get the reference number of the next vgroup in the file.
      */
      vgroup_ref = Vgetid (file_id, vgroup_ref);

      /*
      * Attach to the vgroup for reading or exit the loop if no more vgroups
      * are found.
      */
      if (vgroup_ref == -1) break;
      vgroup_id = Vattach (file_id, vgroup_ref, "r");

      /*
      * Get the total number of objects in the vgroup.
      */
```

```
         num_of_pairs = Vntagrefs (vgroup_id);

         /*
         * If the vgroup contains any object, print the tag/ref number
         * pair of each object in the vgroup, in the order they appear in the
         * file, and indicate whether the object is a vdata, vgroup, or neither.
         */
         if (num_of_pairs > 0)
         {
            printf ("\nVgroup #%d contains:\n", vgroup_pos);
            for (obj_index = 0; obj_index < num_of_pairs; obj_index++)
            {
               /*
               * Get the tag/ref number pair of the object specified
               * by its index, obj_index, and display them.
               */
               status_n = Vgettagref (vgroup_id, obj_index, &obj_tag, &obj_ref);
               printf ("tag = %d, ref = %d", obj_tag, obj_ref);

               /*
               * State whether the HDF object referred to by obj_ref is a vdata,
               * a vgroup, or neither.
               */
               if (Visvg (vgroup_id, obj_ref))
                  printf ("  <-- is a vgroup\n");
               else if (Visvs (vgroup_id, obj_ref))
                  printf ("  <-- is a vdata\n");
               else
                  printf ("  <-- neither vdata nor vgroup\n");
            } /* for */
         } /* if */

         else
            printf ("Vgroup #%d contains no HDF objects\n", vgroup_pos);

         /*
         * Terminate access to the current vgroup.
         */
         status_32 = Vdetach (vgroup_id);

         /*
         * Move to the next vgroup position.
         */
         vgroup_pos++;
      } /* while */

      /*
      * Terminate access to the V interface and close the file.
      */
      status_n = Vend (file_id);
      status_n = Hclose (file_id);
   }
```

FORTRAN:

```
         program  vgroup_contents
         implicit none
   C
   C     Parameter declaration
   C
         character*19 FILE_NAME
   C
         parameter (FILE_NAME = 'General_Vgroups.hdf')
```

```
      integer DFACC_ READ
      parameter (DFACC_READ = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfgid, vntrc, vfgttr, vfisvg,
     +        vfisvs, vfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer vgroup_id, vgroup_ref,  vgroup_pos
      integer obj_index, num_of_pairs
      integer obj_tag, obj_ref
C
C**** End of variable declaration ***********************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Obtain each vgroup in the file by its reference number, get the
C     number of objects in the vgroup, and display the information
C     about that vgroup.
C
      vgroup_ref = -1
      vgroup_pos = 0
10    continue
C
C     Get the reference number of the next vgroup in the file.
C
      vgroup_ref = vfgid(file_id, vgroup_ref)
C
C     Attach to the vgroup or go to the end if no additional vgroup is found.
C
      if(vgroup_ref. eq. -1) goto 100
      vgroup_id = vfatch(file_id, vgroup_ref , 'r')
C
C     Get the total number of objects in the vgroup.
C
      num_of_pairs = vntrc(vgroup_id)
C
C     If the vgroup contains any object, print the tag/ref number
C     pair of each object in vgroup, in the order they appear in the
C     file, and indicate whether the object is a vdata, vgroup, or neither.
C
      if (num_of_pairs .gt. 0) then
         write(*,*) 'Vgroup # ', vgroup_pos, ' contains:'
         do 20 obj_index = 1, num_of_pairs
C
C     Get the tag/ref number pair of the object specified by its index
C     and display them.
C
         status = vfgttr(vgroup_id, obj_index-1, obj_tag, obj_ref)
C
```

```
C     State whether the HDF object referred to by obj_ref is a vdata,
C     a vgroup, or neither.
C
         if( vfisvg(vgroup_id, obj_ref) .eq. 1) then
             write(*,*) 'tag = ', obj_tag, ' ref = ', obj_ref,
     +       '  <--- is a vgroup '
         else if ( vfisvs(vgroup_id, obj_ref) .eq. 1) then
             write(*,*) 'tag = ', obj_tag, ' ref = ', obj_ref,
     +       '  <--- is a vdata '
         else
             write(*,*) 'tag = ', obj_tag, ' ref = ', obj_ref,
     +       '  <--- neither vdata nor vgroup '
         endif
20       continue
      else
         write (*,*) 'Vgroup #', vgroup_pos, ' contains no HDF objects'
      endif
      write(*,*)
      vgroup_pos = vgroup_pos + 1
      goto 10
100   continue
C
C     Terminate access to the vgroup.
C
      status = vfdtch(vgroup_id)
C
C     Terminate access to the V interface and close the HDF file.
C
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

5.6.2.4. Testing Whether a Data Object Belongs to a Vgroup: Vinqtagref

Vinqtagref determines whether a data object is a member of the vgroup specified by the parameter *vgroup_id*. The syntax of **Vinqtagref** is as follows:

```
C:         true_false = Vinqtagref(vgroup_id, obj_tag, obj_ref);

FORTRAN:   true_false = vfinqtr(vgroup_id, obj_tag, obj_ref)
```

The data object is specified by its tag/reference number pair in the parameters *obj_tag* and *obj_ref*. **Vinqtagref** returns `TRUE` (or `1`) if the object belongs to the vgroup, and `FALSE` (or `0`) otherwise. The parameters of this routine are further defined in Table 5H.

5.6.2.5. Testing Whether a Data Object within a Vgroup is a Vgroup: Visvg

Visvg determines whether the data object specified by its reference number, *obj_ref*, is a vgroup and is a member of the vgroup identified by the parameter *vgroup_id*. The syntax of **Visvg** is as follows:

```
C:         true_false = Visvg(vgroup_id, obj_ref);

FORTRAN:   true_false = vfisvg(vgroup_id, obj_ref)
```

Visvg returns either `TRUE` (or `1`) or `FALSE` (or `0`). The parameters of this routine are further defined in Table 5H.

5.6.2.6. Testing Whether an HDF Object within a Vgroup is a Vdata: Visvs

Visvs determines whether the data object specified by its reference number, *obj_ref*, is a vdata and is a member of the vgroup identified by the parameter *vgroup_id*. The syntax of **Visvs** is as follows:

```
C:          true_false = Visvs(vgroup_id, obj_ref);

FORTRAN:    true_false = vfisvs(vgroup_id, obj_ref)
```

Visvs returns either `TRUE` (or `1`) or `FALSE` (or `0`). The parameters of this routine are further defined in Table 5H.

TABLE 5H **Vinqtagref, Visvg, and Visvs Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Vinqtagref [intn] **(vfinqtr)**	vgroup_id	int32	integer	Vgroup identifier
	obj_tag	int32	integer	Tag of the data object to be queried
	obj_ref	int32	integer	Reference number of the data object to be queried
Visvg [intn] **(vfisvg)**	vgroup_id	int32	integer	Vgroup identifier
	obj_ref	int32	integer	Data object reference number to be queried
Visvs [intn] **(vfisvs)**	vgroup_id	int32	integer	Vgroup identifier
	obj_ref	int32	integer	Data object reference number to be queried

5.6.2.7. Locating a Vdata in a Vgroup Given Vdata Fields: Vflocate

Vflocate locates a vdata that belongs to the vgroup identified by the parameter *vgroup_id* and contains the fields specified in the parameter *fieldname_list*. The syntax of **Vflocate** is as follows:

```
C:          vdata_ref = Vflocate(vgroup_id, fieldname_list);

FORTRAN:    vdata_ref = vffloc(vgroup_id, fieldname_list)
```

The parameter *fieldname_list* is a string of comma-separated field names containing no white space, for example, "`PX,PY,PZ`". Note that a vdata must contain all of the fields specified in *fieldname_list* to be qualified.

Vflocate returns the reference number of the vdata, if one is found, and `FAIL` (or `-1`) otherwise. The parameters of this routine are further defined in Table 5I.

5.6.2.8. Retrieving the Number of Tags of a Given Type in a Vgroup: Vnrefs

Vnrefs returns the number of tags of the type specified by the parameter *tag_type* in the vgroup identified by the parameter *vgroup_id*. The syntax of **Vnrefs** is as follows:

```
C:          num_of_tags = Vnrefs(vgroup_id, tag_type);

FORTRAN:    num_of_tags = vnrefs(vgroup_id, tag_type)
```

Possible values of the parameter *tag_type* are defined in Appendix A of this manual. **Vnrefs** returns 0 or the number of tags if successful, and `FAIL` (or `-1`) otherwise. The parameters of this routine are further defined in Table 5I.

TABLE 5I **Vflocate and Vnrefs Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vflocate [int32] **(vffloc)**	vgroup_id	int32	integer	Vgroup identifier
	fieldname_list	char *	character*(*)	Buffer containing the names of the fields
Vnrefs [int32] **(vnrefs)**	vgroup_id	int32	integer	Vgroup identifier
	tag_type	int32	integer	Tag type

5.6.2.9. Retrieving the Reference Number of a Vgroup: VQueryref

VQueryref returns the reference number of the vgroup identified by the parameter *vgroup_id*, or `FAIL` (or `-1`) if unsuccessful. The syntax of **VQueryref** is as follows:

```
C:          vgroup_ref = VQueryref(vgroup_id);

FORTRAN:    vgroup_ref = vqref(vgroup_id)
```

VQueryref is further defined in Table 5J.

5.6.2.10. Retrieving the Tag of a Vgroup: VQuerytag

VQuerytag returns `DFTAG_VG` (or `1965`), which would be the tag of the vgroup identified by the parameter *vgroup_id*, or `FAIL` (or `-1`) if unsuccessful. The syntax of **VQuerytag** is as follows:

```
C:          vgroup_tag = VQuerytag(vgroup_id);

FORTRAN:    vgroup_tag = vqtag(vgroup_id)
```

VQuerytag is further defined in Table 5J.

TABLE 5J **VQueryref and VQuerytag Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VQueryref [int32] **(vqref)**	vgroup_id	int32	integer	Vgroup identifier
VQuerytag [int32] **(vqtag)**	vgroup_id	int32	integer	Vgroup identifier

5.7. Deleting Vgroups and Data Objects within a Vgroup

The Vgroup interface includes two routines for deletion: one deletes a vgroup from a file and the other deletes a data object from a vgroup. These routines are discussed in the following subsections.

5.7.1. Deleting a Vgroup from a File: Vdelete

Vdelete removes the vgroup identified by the parameter *vgroup_id* from the file identified by the parameter *file_id*. The syntax of **Vdelete** is as follows:

```
C:          status = Vdelete(file_id, vgroup_id);

FORTRAN:    status = vdelete(file_id, vgroup_id)
```

This routine will remove the vgroup from the internal data structures and from the file.

Vdelete returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **Vdelete** are further described in Table 5K.

5.7.2. Deleting a Data Object from a Vgroup: Vdeletetagref

Vdeletetagref deletes the data object, specified by the parameters *obj_tag* and *obj_ref*, from the vgroup, identified by the parameter *vgroup_id*. The syntax of **Vdeletetagref** is as follows:

```
C:          status = Vdeletetagref(vgroup_id, obj_tag, obj_ref);

FORTRAN:    status = vfdtr(vgroup_id, obj_tag, obj_ref)
```

Vinqtagref should be used to determine whether the tag/reference number pair exists before calling **Vdeletetagref**. If duplicate tag/reference number pairs are found in the vgroup, **Vdeletetagref** deletes the first occurrence. **Vinqtagref** should also be used to determine whether duplicate tag/reference number pairs exist in the vgroup.

Vdeletetagref returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further described in Table 5K.

TABLE 5K **Vdelete and Vdeletetagref Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
Vdelete [int32] **(vdelete)**	file_id	int32	integer	File identifier
	vgroup_id	int32	integer	Vgroup identifier
Vdeletetagref [int32] **(vfdtr)**	vgroup_id	int32	integer	Vgroup identifier
	obj_tag	int32	integer	Tag of the data object to be deleted
	obj_ref	int32	integer	Reference number of the data object to be deleted

5.8. Vgroup Attributes

HDF version 4.1r1 and later include the ability to assign attributes to a vgroup. The concept of attributes is fully explained in Chapter 3, *Scientific Data Sets (SD API)*. To review briefly, an attribute has a name, a data type, a number of attribute values, and the attribute values themselves. All attribute values must be of the same data type. For example, an attribute value cannot consist of ten characters and one integer, or a character value cannot be included in an attribute value consisting of two 32-bit integers.

Any number of attributes can be assigned to a vgroup, however, each attribute name must be unique among all attributes in the vgroup.

5.8.1. Obtaining the Vgroup Version Number of a Given Vgroup: Vgetversion

The structure of the vgroup has gone through several changes since HDF was first written. Determining the version of any particular vgroup is necessary as some of the older versions of vgroups do not support some of the newer features, such as attributes. **Vgetversion** returns the version number of the vgroup identified by the parameter *vgroup_id*. The syntax of **Vgetversion** is as follows:

```
C:          version_num = Vgetversion(vgroup_id);

FORTRAN:    version_num = vfgver(vgroup_id)
```

There are three valid version numbers: `VSET_OLD_VERSION` (or `2`), `VSET_VERSION` (or `3`), and `VSET_NEW_VERSION` (or `4`).

`VSET_OLD_VERSION` is returned when the vgroup is of a version that corresponds to an HDF library version before version 3.2.

`VSET_VERSION` is returned when the vgroup is of a version that corresponds to an HDF library version between versions 3.2 and 4.0 release 2.

`VSET_NEW_VERSION` is returned when the vgroup is of a version that corresponds to an HDF library version of version 4.1 release 1 or higher.

Vgetversion returns the vgroup version number if successful, and `FAIL` (or `-1`) otherwise. This routine is further defined in Table 5L.

5.8.2. Setting the Attribute of a Vgroup: Vsetattr

Vsetattr attaches an attribute to the vgroup specified by the parameter *vgroup_id*. The syntax of **Vsetattr** is as follows:

```
C:          status = Vsetattr(vgroup_id, attr_name, data_type, n_values,
                              attr_values);

FORTRAN:    status = vfsnatt(vgroup_id, attr_name, data_type, n_values,
                              attr_values)

   OR       status = vfscatt(vgroup_id, attr_name, data_type, n_values,
                              attr_values)
```

If the attribute with the name specified in the parameter *attr_name* already exists, the new values will replace the current ones, provided the data type and count are not different. If either the data type or the count have been changed, **Vsetattr** will return `FAIL` (or `-1`).

The parameter *data_type* is an integer number specifying the data type of the attribute values. Refer to Table 2F for the definition of the data types to interpret this value. The parameter *n_values* specifies the number of values to be stored in the attribute. The buffer *attr_values* contains the values to be stored in the attribute.

Note that the FORTRAN-77 version of **Vsetattr** has two routines; **vfsnatt** sets a numeric value attribute and **vfscatt** sets a character value attribute.

Vsetattr returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5L.

5.8.3. Retrieving the Index of a Vgroup Attribute Given the Attribute Name:

Vfindattr

Vfindattr searches the vgroup, identified by the parameter *vgroup_id*, for the attribute with the name specified by the parameter *attr_name*, and returns the index of that attribute. The syntax of this routine is as follows:

```
C:          attr_index = Vfindattr(vgroup_id, attr_name);

FORTRAN:    attr_index = vffdatt(vgroup_id, attr_name)
```

Vfindattr returns either an attribute index or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5L.

TABLE 5L **Vgetversion, Vsetattr, and Vfindattr Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vgetversion [int32] **(vfgver)**	vgroup_id	int32	integer	Vgroup identifier
Vsetattr [intn] **(vfsnatt/vfscatt)**	vgroup_id	int32	integer	Vgroup identifier
	attr_name	char *	character*(*)	Name of the attribute
	data_type	int32	integer	Data type of the attribute
	n_values	int32	integer	Number of values the attribute contains
	attr_values	VOIDP	<valid numeric data type>(*)/ character* (*)	Buffer containing the attribute values
Vfindattr [intn] **(vffdatt)**	vgroup_id	int32	integer	Vgroup identifier
	attr_name	char *	character*(*)	Name of the target attribute

5.8.4. Obtaining the Total Number of Vgroup Attributes: Vnattrs and Vnattrs2

Both **Vnattrs** and **Vnattrs2** return the number of attributes assigned to the vgroup specified by the parameter *vgroup_id*, but Vnattrs2 is an updated version of Vnattrs. The syntax of both functions are as follows:

```
C:          num_of_attrs = Vnattrs(vgroup_id);
            num_of_attrs = Vnattrs2(vgroup_id);

FORTRAN:    num_of_attrs = vfnatts(vgroup_id)
            Unavailable
```

There are two types of attributes for vgroups. One is the old-style that was created using methods other than the standard attribute API function **Vsetattr**, which was introduced after HDF Version 4.0 Release 2, July 19, 1996. Without the use of **Vsetattr**, an application could simulate an attribute for a vgroup by creating and writing a vdata of class `_HDF_ATTRIBUTE` and adding that vdata to the vgroup via these calls:

```
vdata_ref = VHstoredatam(file_id, ATTR_FIELD_NAME, values, size, type,
                              attr_name, _HDF_ATTRIBUTE, order);
ret_value = Vaddtagref (vgroup_id, DFTAG_VH, vdata_ref);
```

While both types of attributes are stored as vdatas, the vdatas of the two types of attributes are saved differently in the file. Because of the different storages, the new-style attribute functions, such as **Vnattrs**, **Vgetattr** or **Vattrinfo**, would miss the old-style attributes. Starting in release 4.2.6, new functions were added to allow applications to get access to both types of attributes, i.e., **Vnattrs2**, **Vattrinfo2**, and **Vgetattr2**.

Note that, when a vgroup has both type of attributes, the old-style attributes will preceed the new ones, regardless of when they were created. Applications that anticipate to access files that were created by HDF Version 4.0 Release 2 and before (circa July 1996,) should use **Vnattrs2** instead of **Vnattrs** in order to include the old-style attributes if they exist and are desired.

Vnattrs and **Vnattrs2** both returns the number of attributes, if successful, or `FAIL` (or `-1`), otherwise. These routines are further defined in Table 5M.

TABLE 5M **Vnattrs and Vnattrs2 Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vnattrs [intn] **(vfnatts)**	vgroup_id	int32	integer	Vgroup identifier
Vnattrs2 [int32] **(Unavailable)**	vgroup_id	int32	integer	Vgroup identifier

5.8.5. Obtaining Information on a Given Vgroup Attribute: Vattrinfo

Vattrinfo retrieves the name, data type, number of values, and the size of the values of an attribute that belongs to the vgroup identified by the parameter *vgroup_id*. The syntax of **Vattrinfo** is as follows:

```
C:          status = Vattrinfo(vgroup_id, attr_index, attr_name, &data_type,
                               &n_values, &size);

FORTRAN:    status = vfainfo(vgroup_id, attr_index, attr_name, data_type, n_val-
                               ues, size)
```

Vattrinfo stores the name, data type, number of values, and the size of the value of the attribute into the parameters *attr_name*, *data_type*, *n_values*, and *size*, respectively.

The attribute is specified by its index, *attr_index*. The valid values of *attr_index* range from 0 to the total number of attributes attached to the vgroup - 1. The number of vgroup attributes can be obtained using the routine **Vnattrs**.

The parameter *data_type* is an integer number. Refer to Table 2F for the definitions of the data types to interpret this value. The parameter *size* contains the number of bytes taken by an attribute value.

In C, the parameters *attr_name*, *data_type*, *n_values*, and *size* can be set to `NULL`, if the information returned by these parameters is not needed.

Note that, when working with HDF files that were created by HDF Version 4.0 Release 2 and before (circa July 1996,) please refer to the section about **Vattrinfo2**.

Vattrinfo returns either SUCCEED (or 0) or FAIL (or -1). The parameters of this routine are further described in Table 5N.

5.8.6. Obtaining Information on a Given Vgroup Attribute: Vattrinfo2

Vattrinfo2 is an updated version of **Vattrinfo**. Beside retrieving the name, datatype, number of values, and value size of an attribute identified by its index, *attr_index,* in the vgroup, *vgroup_id* as **Vattrinfo**, **Vattrinfo2** also provides the reference number of and the number of fields in the vdata that represents the attribute.

The syntax of **Vattrinfo2** is as follows:

```
C:          status = Vattrinfo2(vgroup_id, attr_index, attr_name, &data_type,
                               &n_values, &size, &n_fields, &ref_num);

FORTRAN:    Unavailable
```

The attribute is specified by its index, *attr_index*. The valid values of *attr_index* range from 0 to the total number of attributes attached to the vgroup - 1. The number of vgroup attributes can be obtained using the routine **Vnattrs2**.

The parameter *data_type* is an integer number. Refer to Table 2F for the definitions of the data types to interpret this value. The parameter *size* contains the number of bytes taken by an attribute value.

In C, the parameters *attr_name*, *data_type*, *n_values*, and *size* can be set to NULL, if the information returned by these parameters is not needed.

Note that, this function should be used in place of **Vattrinfo** when working with HDF files that were created by HDF Version 4.0 Release 2 and before (circa July 1996.) Please refer to Section 5.8.4., "Obtaining the Total Number of Vgroup Attributes: Vnattrs and Vnattrs2" and the Appendix Attribute for more details about vgroup attributes.

Vattrinfo2 returns either SUCCEED (or 0) or FAIL (or -1). The parameters of this routine are further described in Table 5N.

TABLE 5N **Vattrinfo and Vattrinfo2 Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vattrinfo [intn] **(vfainfo)**	vgroup_id	int32	integer	Vgroup identifier
	attr_index	intn	integer	Index of the attribute
	attr_name	char *	character*(*)	Returned name of the attribute
	data_type	int32 *	integer	Returned data type of the attribute
	n_values	int32 *	integer	Returned number of values of the attribute
	size	int32 *	integer	Returned size, in bytes, of the value of the attribute
Vattrinfo2 [intn] **(Unavailable)**	vgroup_id	int32	N/A	Vgroup identifier
	attr_index	intn	N/A	Index of the attribute
	attr_name	char *	N/A	Returned name of the attribute
	data_type	int32 *	N/A	Returned data type of the attribute
	n_values	int32 *	N/A	Returned number of values of the attribute
	size	int32 *	N/A	Returned size, in bytes, of the value of the attribute
	n_fields	int32 *	N/A	Returned number of fields in the attribute vdata
	ref_num	uint16 *	N/A	Returned reference number of the attribute vdata

5.8.7. Retrieving the Values of a Given Vgroup Attribute: Vgetattr

Vgetattr retrieves the values of an attribute of the vgroup specified by the parameter *vgroup_id*. The syntax of **Vgetattr** is as follows:

```
C:          status = Vgetattr(vgroup_id, attr_index, attr_values);

FORTRAN:    status = vfgnatt(vgroup_id, attr_index, attr_values)

    OR      status = vfgcatt(vgroup_id, attr_index, attr_values)
```

The attribute is specified by its index, *attr_index*. The valid values of *attr_index* range from 0 to the total number of attributes attached to the vgroup - 1. The number of vgroup attributes can be obtained using the routine **Vnattrs**.

The buffer *attr_values* must be sufficiently allocated to hold the retrieved attribute values. Use **Vattrinfo** to obtain information about the attribute values for appropriate memory allocation.

Note that the FORTRAN-77 version of **Vgetattr** has two routines; **vfgnatt** gets a numeric value attribute and **vfgcatt** gets a character value attribute.

Vgetattr returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5O.

5.8.8. Retrieving the Values of a Given Vgroup Attribute: Vgetattr2

As **Vgetattr**, **Vgetattr2** retrieves the values of an attribute of the vgroup specified by the parameter *vgroup_id*. The syntax of **Vgetattr2** are as follows:

```
C:          status = Vgetattr2(vgroup_id, attr_index, attr_values);

FORTRAN:    Currently unavailable
```

Unlike **Vgetattr**, **Vgetattr2** can also read values from attributes that were created by methods other than **Vsetattr**. Please refer to Section 5.8.4., "Obtaining the Total Number of Vgroup Attributes: Vnattrs and Vnattrs2" and the Appendix Attribute for information about the different types of vgroup attributes.

The attribute is specified by its index, *attr_index*. The valid values of *attr_index* range from 0 to the total number of attributes attached to the vgroup - 1. The number of vgroup attributes can be obtained using the routine **Vnattrs2**.

The buffer *attr_values* must be sufficiently allocated to hold the retrieved attribute values. Use **Vattrinfo2** to obtain information about the attribute values for appropriate memory allocation.

Vgetattr2 returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 5O.

TABLE 5O **Vgetattr and Vgetattr2 Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vgetattr [intn] **(vfgnatt/vfgcatt)**	vgroup_id	int32	integer	Vgroup identifier
	attr_index	intn	integer	Index of the attribute
	attr_values	VOIDP	<valid numeric data type> (*)/ character*(*)	Buffer containing attribute values
Vgetattr2 [intn] **(unavailable)**	vgroup_id	int32	N/A	Vgroup identifier
	attr_index	intn	N/A	Index of the attribute
	attr_values	VOIDP	N/A	Buffer containing attribute values

EXAMPLE 6. **Operations on Vgroup Attributes**

This example illustrates the use of **Vfind/vfind** to locate a vgroup by its name, **Vsetattr/vfscatt** to attach an attribute to the vgroup, **Vattrinfo/vfainfo** to obtain information about the vgroup attribute, and **Vgetattr/vfgcatt** to obtain the attribute values.

The program obtains the version of the group then sets an attribute named "First Attribute" for the vgroup named "SD Vgroup". Next, the program gets the number of attributes that the vgroup has, and obtains and displays the name, the number of values, and the values of each attribute.

C:

```
#include "hdf.h"

#define  FILE_NAME      "General_Vgroups.hdf"
#define  VGROUP_NAME    "SD Vgroup"
#define  VGATTR_NAME    "First Attribute"
#define  N_ATT_VALUES   7              /* number of values in the attribute */

main( )
{
   /************************* Variable declaration *************************/

   intn   status_n,     /* returned status for functions returning an intn  */
          n_attrs;      /* number of attributes of the vgroup */
   int32  status_32,    /* returned status for functions returning an int32 */
          file_id, vgroup_ref, vgroup_id,
          attr_index, i, vg_version,
          data_type, n_values, size;
   char   vg_attr[N_ATT_VALUES] = {'v','g','r','o','u','p','\0'};
   char   vgattr_buf[N_ATT_VALUES], attr_name[30];

   /********************* End of variable declaration *********************/

   /*
   * Open the HDF file for writing.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Get the reference number of the vgroup named VGROUP_NAME.
   */
   vgroup_ref = Vfind (file_id, VGROUP_NAME);

   /*
   * Attach to the vgroup found.
   */
   vgroup_id = Vattach (file_id, vgroup_ref, "w");

   /*
   * Get and display the version of the attached vgroup.
   */
   vg_version = Vgetversion (vgroup_id);
   switch (vg_version) {
        case VSET_NEW_VERSION:
              printf ("\nVgroup %s is of the newest version, version 4\n",
                   VGROUP_NAME);
           break;
        case VSET_VERSION:
              printf ("Vgroup %s is of a version between 3.2 and 4.0r2\n",
                   VGROUP_NAME);
           break;
        case VSET_OLD_VERSION:
              printf ("Vgroup %s is of version before 3.2\n", VGROUP_NAME);
           break;
        default:
           printf ("Unknown version = %d\n", vg_version);
        } /* switch */

   /*
   * Add the attribute named VGATTR_NAME to the vgroup.
   */
   status_n = Vsetattr (vgroup_id, VGATTR_NAME, DFNT_CHAR, N_ATT_VALUES,
                        vg_attr);

   /*
   * Get and display the number of attributes attached to this vgroup.
   */
   n_attrs = Vnattrs (vgroup_id);
   printf ("\nThis vgroup has %d attribute(s)\n", n_attrs);

   /*
   * Get and display the name and the number of values of each attribute.
   * Note that the fourth and last parameters are set to NULL because the type
   * and the size of the attribute are not desired.
   */
```

```
   for (attr_index = 0; attr_index < n_attrs; attr_index++)
   {
      status_n = Vattrinfo (vgroup_id, attr_index, attr_name, NULL,
                            &n_values, NULL);
      printf ("\nAttribute #%d is named %s and has %d values: ",
                            attr_index+1, attr_name, n_values);

      /*
      * Get and display the attribute values.
      */
      status_n = Vgetattr (vgroup_id, attr_index, vgattr_buf);
      for (i = 0; i < n_values; i++)
         printf ("%c ", vgattr_buf[i]);
      printf ("\n");
   }


   /*
   * Terminate access to the vgroup and to the V interface, and close
   * the HDF file.
   */
   status_32 = Vdetach (vgroup_id);
   status_n = Vend (file_id);
   status_n = Hclose (file_id);
}
```

FORTRAN:

```
      program  vgroup_attribute
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*9  VGROUP_NAME
      character*15 VGATTR_NAME
C
      parameter (FILE_NAME    = 'General_Vgroups.hdf',
     +           VGROUP_NAME  = 'SD Vgroup',
     +           VGATTR_NAME  = 'First Attribute')
      integer VSET_NEW_VERSION, VSET_VERSION, VSET_OLD_VERSION
      parameter (VSET_NEW_VERSION = 4,
     +           VSET_VERSION     = 3,
     +           VSET_OLD_VERSION = 2)
      integer DFACC_WRITE
      parameter (DFACC_WRITE = 2)
      integer DFNT_CHAR
      parameter (DFNT_CHAR = 4)
      integer N_ATT_VALUES
      parameter (N_ATT_VALUES = 6)
C
C     Function declaration
C
      integer hopen, hclose
      integer vfstart, vfatch, vfgver, vfscatt, vfnatts, vfainfo,
     +        vfind, vfgcatt, vfdtch, vfend
C
C**** Variable declaration *******************************************
C
      integer status, n_attrs
      integer file_id
      integer vgroup_id, vgroup_ref, vg_version
      integer attr_index, i
      integer data_type, n_values, size
      character vg_attr(N_ATT_VALUES)
      character vgattr_buf(N_ATT_VALUES), attr_name(30)
      data vg_attr /’v’,’g’,’r’,’o’,’u’,’p’/
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading/writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Get the reference number of the vgroup named VGROUP_NAME.
C
      vgroup_ref = vfind(file_id, VGROUP_NAME)
C
C     Attach to the vgroup found.
C
      vgroup_id = vfatch(file_id, vgroup_ref , ’w’)
C
C     Get and display the version of the attached vgroup.
C
      vg_version = vfgver(vgroup_id)
      if (vg_version .eq. VSET_NEW_VERSION) write(*,*)
     +   VGROUP_NAME, ’ is of the newest version, version 4’
      if (vg_version .eq. VSET_VERSION) write(*,*)
     +   VGROUP_NAME, ’ is of a version between 3.2 and 4.0r2’
      if(vg_version .eq. VSET_OLD_VERSION) write(*,*)
     +   VGROUP_NAME, ’ is of version before 3.2’
      if ((vg_version .ne. VSET_NEW_VERSION) .and.
     +    (vg_version .ne. VSET_VERSION)     .and.
     +    (vg_version .ne. VSET_OLD_VERSION)) write(*,*)
     +    ’Unknown version’
C
C     Add the attribute named VGATTR_NAME to the vgroup.
C
      status = vfscatt(vgroup_id, VGATTR_NAME, DFNT_CHAR, N_ATT_VALUES,
     +                 vg_attr)
C
C     Get and display the number of attributes attached to this group.
C
      n_attrs = vfnatts(vgroup_id)
      write(*,*) ’This group has’, n_attrs, ’ attributes’
C
C     Get and display the name and the number of values of each attribute.
C
      do 10 attr_index=1, n_attrs
         status = vfainfo(vgroup_id, attr_index-1, attr_name, data_type,
     +                    n_values, size)
      write(*,*) ’Attribute #’, attr_index-1, ’ is named ’, attr_name
      write(*,*) ’and has’, n_values, ’ values: ’
C
C     Get and display the attribute values.
C
      status = vfgcatt(vgroup_id, attr_index-1, vgattr_buf)
      write(*,*) (vgattr_buf(i), i=1,n_values)
10    continue
C
```

```
C     Terminate access to the vgroup.
C
      status = vfdtch(vgroup_id)
C
C     Terminate accessto the V interface and close the HDF file.
C
      status = vfend(file_id)
      status = hclose(file_id)
      end
```

5.9. Obsolete Vgroup Interface Routines

The following routines have been replaced by newer routines with similar functionality. These routines are still supported by the Vgroup interface, but their use is not recommended. HDF may not support these routines in a future version.

5.9.1. Determining the Next Vgroup or Vdata Identifier: Vgetnext

Vgetnext gets the reference number of the next member of a vgroup. This member can be either a vgroup or vdata. The syntax for **Vgetnext** is as follows:

```
C:          ref_num = Vgetnext(vgroup_id, v_ref);

FORTRAN:    ref_num = vfgnxt(vgroup_id, v_ref)
```

Vgetnext searches the vgroup, identified by the parameter *vgroup_id*, for the vgroup or vdata whose reference number is specified by the parameter *v_ref*. If this vgroup or vdata is found, **Vgetnext** finds the next vgroup or vdata and returns its reference number. If *v_ref* is set to `-1`, the routine will return the reference number of the first vgroup or vdata in the vgroup.

Vgetnext is now obsolete as the routine **Vgettagref** provides the same functionality. In addition, **Vgettagref** is not restricted to searching for members that are vgroups or vdatas.

Vgetnext returns a reference number if the next vgroup or vdata is found, or `FAIL` (or `-1`) when an error occurs or when there are no more vdatas or vgroups in the vgroup. The parameters of **Vgetnext** are further defined in Table 5P.

5.9.2. Determining the Number of Members and Vgroup Name: Vinquire

Vinquire retrieves the number of data objects and the name of the vgroup identified by the parameter *vgroup_id*. The syntax for **Vinquire** is as follows:

```
C:          status = Vinquire(vgroup_id, &n_members, vgroup_name);

FORTRAN:    status = vfinq(vgroup_id, n_members, vgroup_name)
```

Vinquire stores the number of data objects and the vgroup name in the parameters *n_members* and *vgroup_name*, respectively. In C, if either *n_members* or *vgroup_name* is set to `NULL`, the corresponding data is not returned. The maximum length of the vgroup's name is defined by `VGNAMELENMAX` (or `64`).

Vinquire is now obsolete as the **Vntagrefs** routine can be used to get the number of data objects in a vgroup and **Vgetname** can be used to retrieve the name of a vgroup.

Vinquire returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routines are further defined in Table 5P.

TABLE 5P

Vgetnext and Vinquire Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vgetnext [int32] **(vfgnxt)**	vgroup_id	int32	integer	Vgroup identifier of the parent vgroup
	v_ref	int32	integer	Reference number for the target vgroup
Vinquire [intn] **(vfinq)**	vgroup_id	int32	integer	Vgroup identifier
	n_members	int32 *	integer	Pointer to the number of entries in the vgroup
	vgroup_name	char *	character*(*)	Buffer for the name of the vgroup

CHAPTER 6 --

8-Bit Raster Images (DFR8 API)

6.1. Chapter Overview

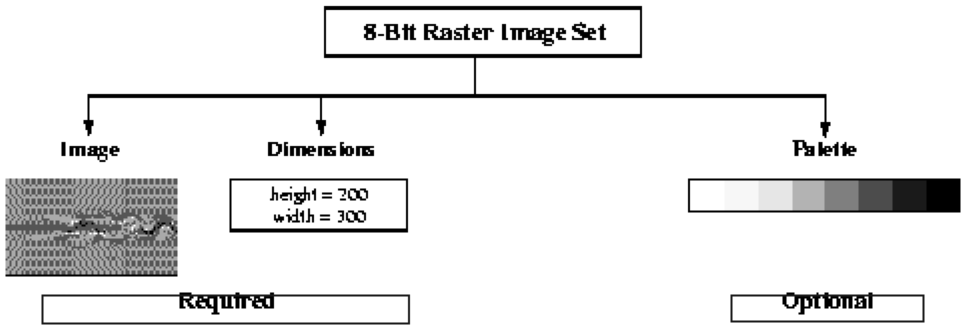
This chapter describes the 8-bit raster image data model and the single-file DFR8 interface routines. The DFR8 interface is a single-file interface that consists of routines for reading and writing raster image sets.

Note: This interface is now deprecated and superseded by the *General Raster Images (GR API)* interface (Chapter 8.)

The 8-Bit Raster Data Model

The data model for the ***8-bit raster image*** set, or ***RIS8***, an acronym for "Raster Image Set, 8-bit", supports three types of objects; two-dimensional 8-bit raster images, dimensions and palettes. The latter two items occur once per RIS8. The following figure shows the contents of an 8-bit raster image set.

FIGURE 6a **8-Bit Raster Image Set Contents**



6.1.1. Required 8-Bit Raster Image Data Set Objects

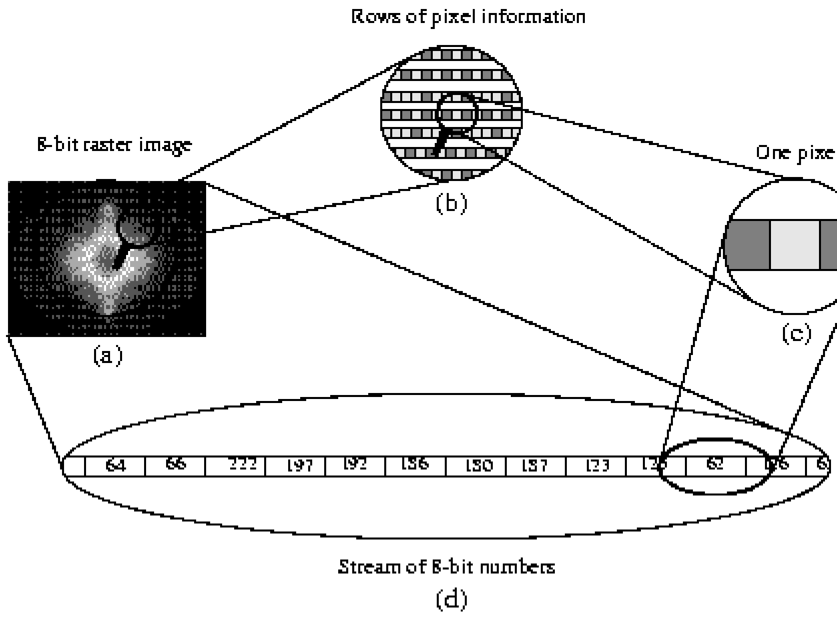
Every RIS8 object requires an image and dimension object. Required objects are created by the HDF library using information provided at the time the image is written.

6.1.1.1. 8-Bit Raster Image Data Representation

An ***8-bit raster image*** is a two-dimensional array of 8-bit numbers which represent ***pixels*** or "picture elements".The first row of pixels corresponds to the top row of the image, the second row of pixels to the second row of the image and so forth. Pixel values range from 0 to 255, and indicate to the hardware which colors to use when mapping the corresponding pixels to the screen display. A ***color lookup table,*** or ***palette***, provides the means of correlating pixel values to colors.

As an example, consider a stream of 8-bit numbers representing a raster image. (See Figure 6b) When the image is displayed, the color associated with the first number in the data stream is placed 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 is necessary to complete the line. The remainder of the rows are similarly painted from left-to-right and top-to-bottom until every value in the data stream appears is represented by one pixel in the image.

FIGURE 6b **The Data Representation of an 8-Bit Raster Image**



An 8-bit raster image (a) is a set of rows displayed from left-to-right consisting of rows of pixel information (b) where each pixel (c) is represented by values stored as a single stream of 8-bit numbers (d).

6.1.1.2. 8-Bit Raster Image Dimension

The dimensions of an image are its height and width in pixels.

6.1.2. Optional 8-Bit Raster Image Data Set Objects

6.1.2.1. Palettes

A ***palette*** is a lookup table consisting of 256 unique numerical values, each of which map to the 256 possible pixel color values and is stored in a RIS8 object. For more details on HDF palettes refer to Chapter 9, titled *Palettes (DFP API)*.

6.1.3. Compression Method

The compression method indicates if and how the image is compressed. It can be, at the programmer's option, explicitly set or left as its default setting of no compression. Compression schemes supported by HDF version 4.0 are run-length encoding or RLE, joint photographic expert group compression, or JPEG, and image compression, or IMCOMP . The list of compression methods is presented below. (See TABLE 6A) The HDF tags `COMP_RLE`, `COMP_IMCOMP` and `COMP_JPEG` are respectively defined as the values 11, 12 and 2 in the "hcomp.h" header file.

TABLE 6A **8-Bit Raster Image Compression Method List**

Compression Method	Type	Compression Code	Requirements
None	N/A	`COMP_NONE`	Image data only (default setting).
RLE	Lossless	`COMP_RLE`	Image data only.
JPEG	Lossy	`COMP_JPEG`	Image data, quality factor and compatibility factor.
IMCOMP	Lossy	`COMP_IMCOMP`	Image data and palette.

RLE Compression

The ***RLE*** method is a lossless compression method recommended for images where data retention is critical. The RLE algorithm compresses images by condensing strings of identical pixel values into two bytes. The first byte identifies the number of pixels in the string and the second byte records the pixel value for the string.

The amount of space saved by RLE depends upon how much repetition there is among adjacent pixels. If there is a great deal of repetition, more space is saved and if there is little repetition, the savings can be very small. In the worst case when every pixel is different from the one that precedes it an extra byte is added for every 127 bytes in the image.

JPEG Compression

The ***JPEG***, or Joint Photographic Expert Group, compression method is a lossy compression algorithm whose use is recommended for photographic or scanned images. Using JPEG compression to reduce the size of an image changes the values of the pixels and hence may alter the meaning of the corresponding data. Version 5.0 of the JPEG library is available in HDF version 4.0.

JPEG compression requires two parameters, the first the level of image quality and the second, compatibility. The ***quality factor*** determines how much of the data will be lost and thus directly impacts the size of the compressed image. A quality factor of 1 specifies the lowest quality or maximum image compression. A quality factor of 100 specifies the highest quality or minimum image compression. Note that all images compressed using the JPEG algorithm are stored in a lossy manner, even those stored with a quality factor of 100. Usually, it is best to experiment with the quality factor to find the most acceptable one.

The *baseline* parameter determines whether the contents of the quantization tables used during compression are forced into the range of 0 to 255. The *baseline* parameter is normally set to the value `1` which forces baseline results. You should set the value of the *baseline* parameter to values other than `1` *only* if you are familiar with the JPEG algorithm.

IMCOMP Compression

IMCOMP is a lossy compression method available in earlier versions of HDF. IMCOMP compression is generally of inferior quality to JPEG compression and is not recommended unless your images will be viewed on a 16-color monitor. For backward compatibility, IMCOMP compression is supported in the HDF library. For details on IMCOMP refer to Appendix F, titled *Backward Compatibility Issues*.

6.2. The 8-Bit Raster Image Interface

The HDF library contains routines for reading and writing 8-bit raster image sets. The functions **DFR8addimage**, **DFR8putimage** and **DFR8getimage** are sufficient for most reading and writing operations.

6.2.1. 8-Bit Raster Image Library Routines

The names of all C functions in the 8-bit raster image interface are prefaced by "DFR8" and the names of the equivalent FORTRAN-77 functions are prefaced by "d8". These routines are divided into the following categories:

- ***Write routines*** create raster image sets and store them in new files or append them to existing files.
- ***Read routines*** determine the dimensions and palette assignment for an image set, read the actual image data and provide sequential or random read access to any raster image set.

The DFR8 function calls are further defined in Table 6B and in the *HDF Reference Manual*.

TABLE 6B **DFR8 Library Routines**

Category	Routine Name		Description
	C	FORTRAN-77	
Write	`DFR8addimage`	`d8aimg`	Appends an 8-bit raster image to a file.
	`DFR8putimage`	`d8pimg`	Writes an 8-bit raster image to an existing file or creates the file.
	`DFR8setcompress`	`d8setcomp`	Sets the compression type.
	`DFR8setpalette`	`d8spal`	Sets palette for multiple 8-bit raster images.
	`DFR8writeref`	`d8wref`	Stores the raster image using the specified reference number.
	`None`	`d8sjpeg`	Passes the quality and compatibility factors needed for the JPEG compression algorithm.
Read	`DFR8getdims`	`d8gdims`	Retrieves dimensions for an 8-bit raster image.
	`DFR8getimage`	`d8gimg`	Retrieves an 8-bit raster image and its palette.
	`DFR8getpalref`	`None`	Returns the reference number of the palette associated with the last image accessed.
	`DFR8lastref`	`d8lref`	Returns reference number of the last element accessed.
	`DFR8nimages`	`d8nims`	Returns number of raster images in a file.
	`DFR8readref`	`d8rref`	Gets the next raster image with the specified reference number.
	`DFR8restart`	`d8first`	Ignores information about last file accessed and restarts from beginning.

6.3. Writing 8-Bit Raster Images

The DFR8 programming model for writing an 8-bit raster image sets is as follows:

1. Set the compression type if the image is to be compressed. (optional)
2. Identify the palette if one is to be stored with the image. (optional)
3. Write the raster data to the file.

The two optional steps can be invoked in any order, as long as they are executed before Step 3. By default, images are stored uncompressed with no associated palette.

6.3.1. Storing a Raster Image: DFR8putimage and DFR8addimage

To write a raster image to an HDF file, the calling program must contain the following:

```
C:          status = DFR8putimage(filename, image, width, height, compress);

FORTRAN:    status = d8pimg(filename, image, width, height, compress)
```

OR

```
C:          status = DFR8addimage(filename, image, width, height, compress);

FORTRAN:    status = d8aimg(filename, image, width, height, compress)
```

DFR8putimage and **DFR8addimage** write an 8-bit raster image to an HDF file named by the *filename* parameter. When given a new filename, **DFR8putimage** and **DFR8addimage** create a new file and write the raster image as the first raster image in the file. When given an existing file-name, **DFR8putimage** overwrites the file whereas **DFR8addimage** appends data to the end of the file.

In the **DFR8putimage** and **DFR8addimage** functions, the raster data is passed in the image parameter and the width and height of the image are passed in the width and height parameters. The compression algorithm used to store the image is passed in the compress parameter. Valid compress values include `COMP_NONE`, `COMP_RLE`, `COMP_JPEG` and `COMP_IMCOMP`. `COMP_NONE` represents no compression (storage only), `COMP_RLE` represents run-length encoding, `COMP_JPEG` represents JPEG compression and `COMP_IMCOMP` represents IMCOMP encoding.

Parameters for **DFR8putimage** and **DFR8addimage** are further described in Table 6C.

TABLE 6C **DFR8putimage and DFR8addimage Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFR8putimage [intn] **(d8pimg)** and **DFR8addimage** [intn] **(d8aimg)**	filename	char *	character*(*)	Name of file the raster image will be stored in.
	image	VOIDP	<valid numeric data type>	Image data array.
	width	int32	integer	Number of columns in the raster image.
	height	int32	integer	Number of rows in the raster image.
	compress	int16	integer	Compression type.

EXAMPLE 1. **Writing an 8-Bit Raster Image to an HDF File**

In the following code examples, **DFR8addimage** and **d8aimg** are used to write an 8-bit image to a file named "Example1.hdf". Note that the order in which the dimensions for the image array are declared differs between C and FORTRAN-77.

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 };

    intn status;


    /* Write the 8-bit raster image to file */
    status = DFR8addimage("Example1.hdf", raster_data,
                        WIDTH, HEIGHT, 0);

}
```

FORTRAN:

```
      PROGRAM RASTER8

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

       integer*4 WIDTH, HEIGHT
       parameter(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 /

C      Write the 8-bit raster image to the file
       retn = d8aimg('Example1.hdf', raster_data, WIDTH, HEIGHT, 0)

       end
```

6.3.2. Adding a Palette to an RIS8 Object: DFR8setpalette

DFR8setpalette identifies the palette to be used for the subsequent write operations. It may be used to assign a palette to a single image or several images. After a palette has been set, it acts as the current palette until it is replaced by another call to **DFR8setpalette**. To create a raster image set containing a palette, the calling program must contain the following:

```
C:          status = DFR8setpalette(palette);
            status = DFR8addimage(filename, image, width, height, compress);

FORTRAN:    status = d8spal(palette)
            status = d8aimg(filename, image, width, height, compress)
```

DFR8setpalette takes *palette* as its only parameter. To set the default palette to “no palette”, pass `NULL` as the *palette* parameter. **DFR8setpalette** is further defined in the following table.

TABLE 6D

DFR8setpalette Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFR8setpalette [intn] **(d8spal)**	palette	uint8 *	character*(*)	Palette to be assigned.

EXAMPLE 2.

Writing a Palette and an Image in RIS8 Format

These examples demonstrate how a palette stored in the array colors and the raw image stored in the 20 x 20 array picture is written to a RIS8 object. The image is not compressed and, in these examples, uninitialized. The raster image set is stored as the first image in "Example2.hdf". Note that because **DFR8putimage** recreates the file, anything previously contained in this file will be erased.

C:

```
#include "hdf.h"

#define WIDTH 20
#define HEIGHT 20

main( )
{
    uint8 colors[256*3], picture[HEIGHT][WIDTH];
    uint8 i, j;
    int16 status;

    /* Initialize image arrays. */
    for (j = 0; j < WIDTH; j++) {
            for (i = 0; i < HEIGHT; i++)
                picture[j][i] = 1;
      }

    /* Set the current palette. */
    status = DFR8setpalette(colors);

    /* Write the image data to the file. */
    status = DFR8putimage("Example2.hdf", picture, WIDTH,
                        HEIGHT, COMP_NONE);

}
```

FORTRAN:

```
     PROGRAM WRITE UNCOMPRESSED RIS8

      integer   d8spal, d8pimg, status, i, j
      integer   colors(768)
      integer*4   WIDTH, HEIGHT, COMP_NONE
      parameter (COMP_NONE = 0,
     +           WIDTH = 20,
     +           HEIGHT = 20)
      integer   picture(WIDTH, HEIGHT)

C     Initialize the image data.
```

```
      do 20 j = 1, WIDTH
       do 10 i = 1, HEIGHT
          picture(j, i) = 1
10     continue
20    continue

C     Set the current palette.
      status = d8spal(colors)

C     Write the image data to the file.
      status = d8pimg('Example2.hdf', picture, WIDTH, HEIGHT,
     +                COMP_NONE)

      end
```

6.3.3. Compressing 8-Bit Raster Image Data: DFR8setcompress

The compression type is determined by the tag passed as the fifth argument in calls to the **DFR8-putimage** and **DFR8addimage** routines. **DFR8setcompress** is currently required only to reset the default JPEG compression options. However, future versions of this routine will support additional compression schemes.

To set non-default compression parameters, the calling program should contain the following sequence of routines:

```
C:          status = DFR8setcompress(type, c_info);
            status = DFR8addimage(filename, image, width, height, compress);

FORTRAN:    status = d8scomp(type)
            <compression-specific code>
            status = d8aimg(filename, image, width, height, compress)
```

Notice that the calling sequence for C differs from the calling sequence for FORTRAN-77. Once the compression is set, the parameter *type* in the **DFR8setcompress** routine, or **d8scomp** in FORTRAN-77, specifies the compression method that will be used when storing the raster images. However, the *c_info* parameter, which is a pointer to a structure that contains information specific to the compression scheme indicated by the *type* parameter in **DFR8setcompress**, is missing from **d8scomp**. Because data structures of variable size are not supported in FORTRAN-77, another routine specific to the compression library is required in the FORTRAN-77 calling sequence.

The *c_info* union is described in Chapter 3, titled *Scientific Data Sets (SD API)*. The values contained in this union are passed into the **d8sjpeg** FORTRAN-77-specific routine.

Parameters for **DFR8setcompress** and **d8sjpeg** are further described in Table 6E below.

TABLE 6E

DFR8setcompress Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFR8setcompress [intn] **(d8scomp)**	type	int32	integer	Compression method.
	c_info	comp_info *	None	Pointer to JPEG information structure.
(d8sjpeg) **[integer]**	quality	none	integer	JPEG quality factor.
	baseline	none	integer	JPEG baseline.

EXAMPLE 3.

Writing a Set of Compressed 8-Bit Raster Images

These examples contain a series of calls in which four 20 x 20 images are written to the same file. The first two use palette *paletteA* and are compressed using the RLE method; the third and fourth use palette *paletteB* and are not compressed.

C:

```
 #include "hdf.h"

#define WIDTH 20
#define HEIGHT 20

main ( )
{
    uint8 paletteA[256*3], paletteB[256*3];
    uint8 picture1[HEIGHT][WIDTH], picture2[HEIGHT][WIDTH];
    uint8 picture3[HEIGHT][WIDTH], picture4[HEIGHT][WIDTH];
    uint8 i, j;
    int16 status;

    /* Initialize image arrays. */
    for (j = 0; j < WIDTH; j++) {
            for (i = 0; i < HEIGHT; i++) {
                picture1[j][i] = 1;
                picture2[j][i] = 1;
                picture3[j][i] = 1;
                picture4[j][i] = 1;
            }
    }

    /* Set the first palette. */
    status = DFR8setpalette(paletteA);

    /* Write the compressed image data to the HDF file. */
    status = DFR8putimage("Example3.hdf", (VOIDP)picture1, WIDTH, HEIGHT, \
                    COMP_RLE);
    status = DFR8addimage("Example3.hdf", (VOIDP)picture2, WIDTH, HEIGHT, \
                    COMP_RLE);

    /* Set the second palette. */
    status = DFR8setpalette(paletteB);

    /* Write the uncompressed image data to the HDF file. */
    status = DFR8addimage("Example3.hdf", (VOIDP)picture3, WIDTH, HEIGHT, \
                    COMP_NONE);
    status = DFR8addimage("Example3.hdf", (VOIDP)picture4, WIDTH, HEIGHT, \
                          COMP_NONE);

}
```

FORTRAN:

```
      PROGRAM WRITE IMAGE SETS

       integer d8spal, d8pimg, d8aimg, status
       integer*4 COMP_RLE, COMP_NONE, WIDTH, HEIGHT
       parameter (COMP_RLE = 11,
      +           COMP_NONE = 0,
      +           WIDTH = 20,
      +           HEIGHT = 20)

       integer paletteA(768), paletteB(768)
       integer picture1(WIDTH, HEIGHT), picture2(WIDTH, HEIGHT)
       integer picture3(WIDTH, HEIGHT), picture4(WIDTH, HEIGHT)

C      Initialize the image data.
       do 20 j = 1, WIDTH
        do 10 i = 1, HEIGHT
         picture1(j, i) = 1
         picture2(j, i) = 1
         picture3(j, i) = 1
         picture4(j, i) = 1
10      continue
20     continue

C      Set the first palette.
       status = d8spal(paletteA)

C      Write the compressed image data to the HDF file.
       status = d8pimg('Example3.hdf', picture1, WIDTH, HEIGHT,
      +               COMP_RLE)
       status = d8aimg('Example3.hdf', picture2, WIDTH, HEIGHT,
      +               COMP_RLE)

C      Set the second palette.
       status = d8spal(paletteB)

C      Write the uncompressed image data to the HDF file.
       status = d8aimg('Example3.hdf', picture3, WIDTH, HEIGHT,
      +               COMP_NONE)
       status = d8aimg('Example3.hdf', picture4, WIDTH, HEIGHT,
      +               COMP_NONE)

       end
```

EXAMPLE 4.

Compressing and Writing a 8-Bit Raster Image

In the following examples, **DFR8addimage** and **DFR8compress** are used to compress an 8-bit image and write it to an HDF file named "Example2.hdf". Notice that compressing an image in C requires only one function call, whereas compressing an image using FORTRAN-77 requires two. The second FORTRAN-77 call is required because it is not valid to pass a structure as a parameter in FORTRAN-77.

C:

```
#include "hdf.h"
#include "hcomp.h"
```

```
#define WIDTH 3
#define HEIGHT 5
#define PIXEL_DEPTH 3

main( )
{
    /* Initialize the image array. */
    static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
            {  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, 31,32,33, 34,35,36,
              37,38,39, 40,41,42, 43,44,45 };
    static comp_info compress_info;
    intn status;

    /* Initialize JPEG compression structure. */
    compress_info.jpeg.quality = 60;
    compress_info.jpeg.force_baseline = 1;

    /* Set JPEG compression for storing the image. */
    status = DFR8setcompress(COMP_JPEG, &compress_info);

    /* Write the 8-bit image data to file. */
    status = DFR8addimage("Example2.hdf", (VOIDP)raster_data, WIDTH,
                          HEIGHT, COMP_JPEG);

}
```

FORTRAN:

```
      PROGRAM COMPRESS RIS8

       integer d8aimg, d8scomp, d8sjpeg, status
       integer*4 WIDTH, HEIGHT, PIXEL_DEPTH, COMP_JPEG

C      COMP_JPEG is defined in hcomp.h.
       parameter(WIDTH = 3,
      +          HEIGHT = 5,
      +          COMP_JPEG = 1,
      +          PIXEL_DEPTH = 3)
       character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

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, 31,32,33, 34,35,36,
      +    37,38,39, 40,41,42, 43,44,45  /

C      Set compression.
       status = d8scomp(COMP_JPEG)

C      Set JPEG parameters to quality = 60, and turn compatibility on.
       status = d8sjpeg(60, 1)

C      Write the 8-bit image data to the HDF file.
       status = d8aimg('Example2.hdf', raster_data, WIDTH, HEIGHT,
      +              COMP_JPEG)

       end
```

6.3.4. Specifying the Reference Number of an RIS8: DFR8writeref

DFR8writeref specifies the reference number of the image to be written when **DFR8addimage** or **DFR8putimage** is called. Use the following calling sequence to invoke **DFR8writeref**:

```
C:          status = DFR8writeref(filename, ref);
            status = DFR8addimage(filename, image, width, height, compress);

FORTRAN:    status = d8wref(filename, ref)
            status = d8aimg(filename, image, width, height, compress)
```

DFR8writeref assigns the reference number passed in the *ref* parameter to the next image the file specified by the *filename* parameter. If the value of *ref* is the same as the reference number of an existing RIS8, the existing raster image data will be overwritten. The parameters for **DFR8writeref** are further described below. (See TABLE 6F)

It is unlikely that you will need this routine, but if you do, use it with caution. It is not safe to assume that a reference number indicates the file position of the corresponding image as there is no guarantee that reference numbers appear in sequence in an HDF file.

TABLE 6F **DFR8writeref Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFR8writeref [intn] **(d8wref)**	filename	char *	character*(*)	Name of the HDF file containing the raster image.
	ref	uint16	integer	Reference number for next call to **DFR8getimage**.

6.4. Reading 8-Bit Raster Images

The DFR8 programming model for reading an 8-bit raster image set is as follows:

1. Determine the dimensions of the image if they are not known prior to the read operation.
2. Read the image from the file.

6.4.1. Reading a Raster Image: DFR8getimage

If dimensions of the image are known, **DFR8getimage** is the only function call needed to read a raster image. If a file is being opened for the first time, **DFR8getimage** returns the first image in the file. Additional calls will return successive images in the file, therefore images are read in the order which they were written to the file. **DFR8getdims** is called before **DFR8getimage** so that space allocations for the image and palette can be checked and the dimensions verified. If this information is already known, **DFR8getdims** may be omitted.

To read a raster image from an HDF file, the calling program must contain the following:

```
C:          status = DFR8getimage(filename, image, width, height, palette);

FORTRAN:    status = d8gimg(filename, image, width, height, palette)
```

DFR8getimage retrieves the next 8-bit image from the HDF file name specified by the *filename* parameter. If the image in the file is compressed, **DFR8getimage** first decompresses it then places it in memory at the location pointed to by the *image* parameter. The dimensions of the array allocated to hold the image are specified by the *width* and *height* parameters and may be larger than the actual image.The palette, if present, is stored in memory at the location pointed to by the *palette* parameter. If it contains a `NULL` value the palette is not loaded, even if there is one stored with the image. The parameters for **DFR8getimage** are defined further in Table 6G below.

Notice that in Example 4, as in the case of **DFR8addimage**, the order in which the dimensions for the image array are declared differs between C and FORTRAN-77. FORTRAN-77 declarations require the width before the height while the C declaration requires the height before the width as FORTRAN-77 arrays are stored in column-major order, while C arrays are stored in row-major order. (row-major order implies that the second coordinate varies fastest). When **d8gimg** reads an image from a file, it assumes column-major order.

6.4.2. Querying the Dimensions of an 8-Bit Raster Image: DFR8getdims

DFR8getdims opens a named file, finds the next image or the first image if the file is being opened for the first time, retrieves the dimensions of the image and determines if there is a palette associated with the image. If the file is being opened for the first time, **DFR8getdims** returns information about the first image in the file. If an image has already been read, **DFR8getdims** finds the next image. In this way, images are read in the same order in which they were written to the file.

To determine the dimensions of an image before attempting to read it, the calling program must include the following routines:

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

FORTRAN:   status = d8gdim(filename, width, height, haspalette)
           status = d8gimg(filename, image, width, height, palette)
```

DFR8getdims retrieves dimension and palette information about the next 8-bit image in the file specified by *filename*. The returned information is pointed to by the *width* and *height* parameters. The *haspalette* parameter determines the presence of a palette and returns a value of 1 if it exists and 0 otherwise. The parameters for **DFR8getdims** are defined further in the following table.

TABLE 6G

DFR8getdims and DFR8getimage Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFR8getdims [intn] **(d8gdims)**	filename	char *	character*(*)	Name of the HDF file containing the set of raster images.
	width	int32 *	integer	Number of columns in the next raster image.
	height	int32 *	integer	Number of rows in the next raster image.
	ispalette	intn *	integer	“1” if a palette exists, otherwise “0”.
DFR8getimage [intn] **(d8gimg)**	filename	char *	character*(*)	Name of HDF file with the raster image.
	image	uint8 *	character*(*)	Buffer for the raster image.
	width	int32	integer	Width of the raster image buffer.
	height	int32	integer	Height of the raster image buffer.
	palette	uint8 *	character*(*)	Palette assigned to the raster image.

EXAMPLE 5.

Reading an 8-Bit Raster Image

The following examples search the "Example1.hdf" file created in Example 1 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. This example also assumes that the data set does not include a palette, therefore `NULL` is passed as the palette parameter. If the palette argument is `NULL` (or "0" in FORTRAN-77), 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, status;

    /* Get the dimensions of the image */
    status = DFR8getdims("Example1.hdf", &width, &height, &haspal);

    /* Read the raster data if the dimensions are correct */
    if (width <= WIDTH && height <= HEIGHT)
            status = DFR8getimage("Example1.hdf", (VOIDP)raster_data, width,
                        height, NULL);

}
```

FORTRAN:

```
      PROGRAM RASTER8

       character*1 image(5, 6)
       integer status, height, width, d8gimg, d8gdims, haspal
       integer*4 width, height

C      Get the dimensions of the image.
```

```
      status = d8gdims('Example1.hdf', width, height, haspal)

C     Read the raster data if the dimensions are correct.
      if (width .le. 5 .and. height .le. 6) then
         status = d8gimg('Example1.hdf', image, width, height, 0)
      endif

      end
```

6.4.3. Reading an Image with a Given Reference Number: DFR8readref

DFR8readref accesses specific images that are stored in files containing multiple raster image sets. It is an optionally used before **DFR8getimage** to set the access pointer to the specified raster image. **DFR8readref** can be used in connection with vgroups, which identify their members by tag/reference number pairs. See Chapter 5, titled Vgroups (V API), for a discussion of vgroups and tag/reference number pairs.

To access a specific raster image set, use the following calling sequence:

```
C:         status = DFR8readref(filename, ref);
           status = DFR8getimage(filename, image, width, height, palette);

FORTRAN:   status = d8rref(filename, ref)
           status = d8gimg(filename, image, width, height, palette)
```

DFR8readref specifies that the target for the next read operation performed on the HDF file specified by the *filename* parameter is the object with the reference number named in the *ref* parameter. The parameters required for **DFR8readref** are defined further in the following table.

TABLE 6H **DFR8readref Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFR8readref [intn] **(d8rref)**	filename	char *	character*(*)	Name of HDF file containing the raster image.
	ref	uint16	integer	Reference number for next call to **DFR8getimage**.

6.4.4. Specifying the Next 8-Bit Raster Image to be Read: DFR8restart

DFR8restart causes the next call to **DFR8getimage** or **DFR8getdims** to read the first raster image set in the file. Use the following call to invoke **DFR8restart**:

```
C:         status = DFR8restart( );

FORTRAN:   status = d8first( )
```

6.5. 8-Bit Raster Image Information Retrieval Routines

6.5.1. Querying the Total Number of 8-Bit Raster Images: DFR8nimages

DFR8nimages returns the total number of 8-bit raster image sets in a file and has the following syntax:

```
C:          num_of_images = DFR8nimages(filename);

FORTRAN:    num_of_images = d8nimg(filename)
```

TABLE 6I **DFR8nimages Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FOR-TRAN-77**	
DFR8nimages [intn] **(d8nims)**	filename	char *	character*(*)	Name of the HDF file.

6.5.2. Determining the Reference Number of the Most-Recently-Accessed 8-Bit Raster Image: DFR8lastref

DFR8lastref returns the reference number most recently used in writing or reading an 8-bit raster image. This routine is primarily used for attaching annotations to images and adding images to vgroups. (See Chapters 8, titled Annotations (DFAN API) and Chapter 5, titled Vgroups (V API) for more detailed information on how to use reference numbers in connection with these applications.)

The following calling sequence uses **DFR8lastref** to find the reference number of the 8-bit raster image most recently added to an HDF file:

```
C:          status = DFR8addimage(filename, image, width, height, compress);
            lastref = DFR8lastref( );

FORTRAN:    status = d8aimg(filename, image, width, height, compress)
            lastref = d8lref( )
```

DFR8putimage or **DFR8getimage** can be used instead of **DFR8addimage** with similar results.

6.5.3. Determining the Reference Number of the Palette of the Most-Recently-Accessed 8-Bit Raster Image: DFR8getpalref

DFR8getpalref returns the reference number of the palette associated with the most recently used in writing or reading an 8-bit raster image. The **DFR8getdims** routine must be called before **DFR8getpalref**, as **DFR8getdims** initializes internal structures required by **DFR8getpalref**.

There is currently no FORTRAN-77 version of the **DFR8getpalref** routine.

TABLE 6J **DFR8nimages Parameter List**

Routine Name [Return Type]	Parameter	Parameter Type	Description
		C	
DFR8getpalref [intn]	pal_ref	uint16 *	Pointer to the returned reference number of the palette.

6.6. RIS8 Backward Compatibility Issues

6.6.1. Attribute "long_name" Included in HDF for netCDF Compatibility

In several routines of the RIS8 interface, the value returned by *label* is the value of the attribute named "long_name" and that the value returned by *coordsys* is the value of the attribute named "coordsys".

This was done in order to provide HDF with the ability to read netCDF files. While this aspect of HDF functionality will not affect its ability to read HDF data files written by programs compiled with earlier versions of HDF, it is advisable for HDF users to know this to be aware of the significance of the "long_name" and "coordsys" attribute names in a list of attributes.

6.6.2. Raster Image Group Implementation with New RIS8 Tags

As HDF has evolved, a variety of structures have been used to store raster image sets. For instance, HDF first began grouping 8-bit raster images together with dimensions and palettes by insuring that their reference numbers fell in a certain pattern. This method of organizing raster images quickly lead to very complicated collections of images, dimension records, and palettes, and eventually was replaced by an specific grouping structure known as a ***Raster Image Group***, or ***RIG***, with a completely new set of tags.

To maintain backward compatibility with older versions of HDF, the RIS8 interface supported by HDF version 4.1 and later recognizes raster images stored using either set of HDF tags. Details on the different tags and structures used to store raster images can be found in the *HDF Specifications and Developer's Guide v3.2* from the HDF web site at `http://www.hdfgroup.org/`.

CHAPTER 7 --

24-bit Raster Images (DF24 API)

7.1. Chapter Overview

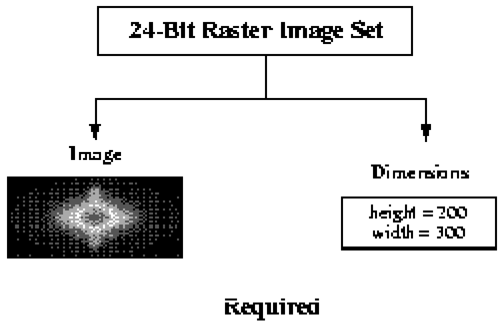
This chapter describes the 24-bit raster data model and the single-file DF24 routines available for storing and retrieving 24-bit raster images.

Note: This interface is now deprecated and superseded by the *General Raster Images (GR API)* interface (Chapter 8.)

7.2. The 24-Bit Raster Data Model

The ***24-bit raster image set***, or ***RIS24***, data model supports two primary data objects: two-dimensional 24-bit raster images and dimensions. The primary member of the set is the ***24-bit raster image***, a two-dimensional array of pixels or picture elements. Each pixel is represented by three 8-bit numbers of image data. An optional compression method describes the method used, if any, to compress the image. Figure 7a shows the contents of a 24-bit raster image set.

FIGURE 7a **24-Bit Raster Image Set Contents**



7.2.1. Required 24-Bit Raster Image Data Set Objects

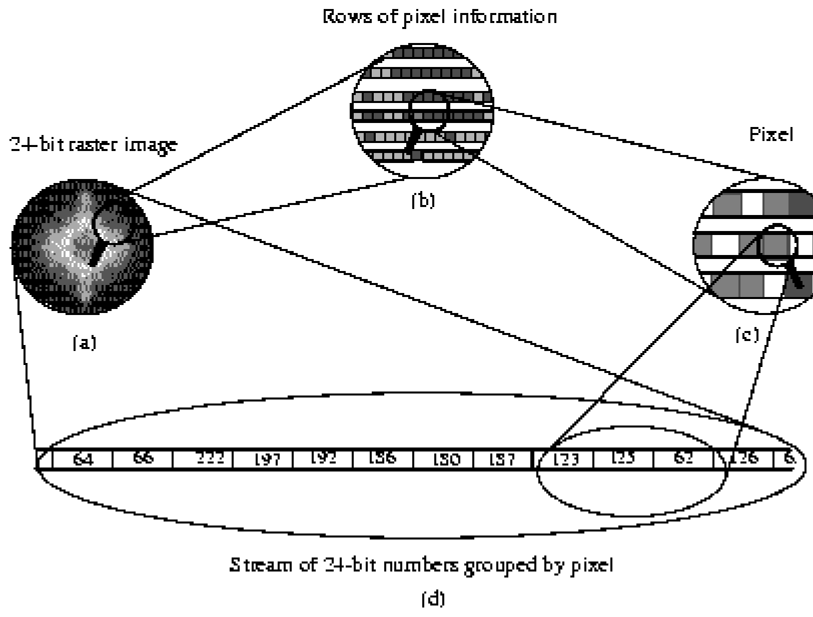
All 24-bit raster images must contain image data and a dimension record. These objects are created by the HDF library using information provided at the time the image is written to file.

7.2.1.1. 24-Bit Raster Image Data Representation

The ***24-bit raster image object contains a set of*** 24-bit pixel values, each of which has three 8-bit components; one for the red, one for the green, and one for the blue color component of the image. These values, referred to as ***RGB values***, are arranged in one of three specific ways, as described in Section "*Interlace Modes"*. The pixel values are arranged in rows, painted from left-to-right, top-to-bottom. As each pixel in a 24-bit image is represented in the image data by three 8-bit numbers, palettes are unnecessary and are not included in the 24-bit raster data model.

As an example, consider a stream of 24-bit numbers representing a raster image (Fig. 4.1a). To display the image, the color associated with the first number in the data stream appears 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 7b **The Numerical Representation of a 24-Bit Raster Image**



A 24-bit raster image (a) is a set of rows displayed from left-to-right consisting of rows of pixels (b) whose values are stored as three 8-bit numbers (c) in a stream of data (d). In this figure, the image is interleaved by pixel.

7.2.1.2. 24-Bit Raster Image Dimension

The ***dimensions*** of an image are the height and width of the image in pixels.

7.2.2. Optional 24-Bit Raster Image Data Set Objects

7.2.2.1. Compression Method

The only 24-bit compression method currently available in HDF is the JPEG algorithm . The applicable HDF compression tags are `COMP_JPEG`, and `COMP_NONE`. (See TABLE 7A) The HDF tags `COMP_JPEG` and `COMP_NONE` are defined as the values `2`, and `0` respectively in the "hdf.h" header file.

TABLE 7A **24-Bit Raster Image Compression Method List**

Compression Method	Type	Compression Code	Requirements
None	Lossless	`COMP_NONE`	Image data only (default setting).
JPEG	Lossy	`COMP_JPEG`	Image data, quality factor and compatibility factor.

JPEG Compression

The ***JPEG*** compression method is a lossy compression algorithm whose use is recommended for photographic or scanned images. Using JPEG compression to reduce the size of an image changes the values of the pixels and therefore may alter the meaning of the corresponding data.

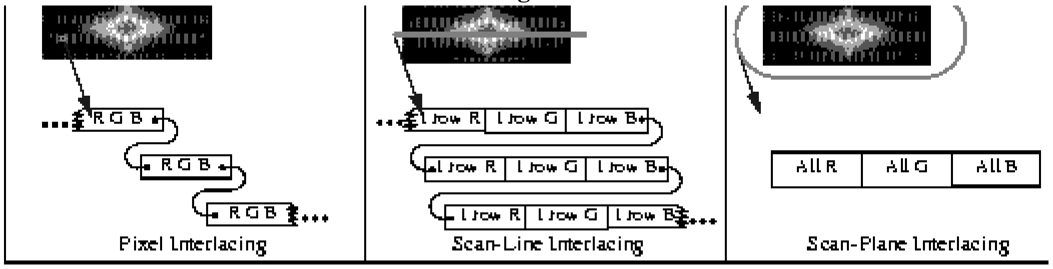
For more information on the JPEG algorithm, refer to Chapter 6, *8-Bit Raster Images (DFR8 API)*.

7.2.2.2. Interlace Modes

Because graphics applications and hardware devices vary in the way they access image data, HDF supports three interlace formats. By storing an image using a format that is consistent with the expected application or device, it is possible to achieve substantial improvements in performance.

HDF provides three options for organizing the color components in 24-bit raster images. These options consist of pixel interlacing, scan-line interlacing, and scan-plane interlacing. (See Figure 7c.) 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 one row of red, one row of green, one row of blue, one row red, etc., is called ***scan-line interlacing***. 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 7c **RGB Interlace Format for 24-Bit Raster Images**



An interlace format describes both the physical format of an image as it is stored in memory and in the file. When writing to a file, HDF stores a 24-bit image using the same interlace format as it has in memory. However, when reading from a file, it is possible to make the in-core interlacing mode different from that used in the file. The following table contains a summary of the interlacing format available in the DF24 interface.

TABLE 7B **24-Bit Raster Image Interlace Format**

format	Description	DF24setil or d2setil Parameter	Size of Image Array
Pixel	Components grouped by pixel.	`DFIL_PIXEL`	Width x Height x 3
Scan-line	Components grouped by row.	`DFIL_LINE`	Width x 3 x Height
Scan-plane	Components grouped by plane.	`DFIL_PLANE`	3 x Width x Height

7.3. The 24-Bit Raster Interface

The HDF library currently contains several routines for storing 24-bit raster images in the HDF format. The **DF24addimage**, **DF24putimage**, and **DF24getimage** routines are sufficient for most reading and writing operations.

7.3.1. 24-Bit Raster Image Library Routines

The names of all C routines in the 24-bit raster image interface are prefaced by "DF24". The equivalent FORTRAN-77 routines are prefaced by "d2". These routines are divided into the following categories:

- ***Write routines*** create raster image sets and store them in new files or append them to existing files.
- ***Read routines*** determine the dimensions and interlace format of an image set, read the actual image data, and provide sequential or random read access to any raster image set.

The DF24 function calls are more explicitly defined in Table 7C and on their respective pages in the *HDF Reference Manual*.

TABLE 7C **DF24 Library Routines**

Purpose	Routine Name		Description
	C	FORTRAN-77	
Write	`DF24addimage`	`d2aimg`	Appends a 24-bit raster image to a file.
	`DF24lastref`	`d2lref`	Reports the last reference number assigned to a 24-bit raster image.
	`DF24putimage`	`d2pimg`	Writes a 24-bit raster image to file by overwriting all existing data.
	`DF24setcom-press`	`d2scomp`	Sets the compression method for the next raster image written to the file.
	`DF24setdims`	`d2sdims`	Sets the dimensions for the next raster image written to the file.
	`DF24setil`	`d2setil`	Sets the interlace format of the next raster image written to the file.
	`None`	`d2sjpeg`	Fortran-specific routine for setting the parameters needed for the JPEG compression algorithm.
Read	`DF24getdims`	`d2gdims`	Retrieves the dimensions before reading the next raster image.
	`DF24getimage`	`d2gimg`	Reads the next 24-bit raster image.
	`DF24nimage`	`d2nimg`	Reports the number of 24-bit raster images in a file.
	`DF24readref`	`d2rref`	Reads 24-bit raster image with the specified reference number.
	`DF24reqil`	`d2reqil`	Retrieves the interlace format before reading the next raster image.
	`DF24restart`	`d2first`	Returns to the first 24-bit raster image in the file.

7.4. Writing 24-Bit Raster Images

The DF24 programming model for writing a 24-bit raster image set is as follows:

1. Set the interlace format if the interlacing is to be different from pixel interlacing. (optional)
2. Set the compression type if the image is to be compressed. (optional)
3. Write the raster data to the file.

Steps 1 and 2 can be invoked in any order, as long as they are executed before Step 3. By default, images are stored uncompressed using pixel interlacing.

7.4.1. Writing a 24-Bit Raster Image: DF24putimage and DF24addimage

To write a raster image to an HDF file, the calling program must contain one of the following function calls:

```
C:          status = DF24putimage(filename, image, width, height);

FORTRAN:    status = d2pimg(filename, image, width, height)

    OR

C:          status = DF24addimage(filename, image, width, height);

FORTRAN:    status = d2aimg(filename, image, width, height)
```

DF24putimage and **DF24addimage** write a 24-bit raster images to the HDF file specified by the *filename* parameter. When given a new file name, **DF24putimage** and **DF24addimage** create a new file and write the raster image as the first raster image in the file. If a file with the specified filename exists, **DF24putimage** overwrites the previous contents of the file whereas **DF24addimage** appends data to the end of the file.

DF24putimage and **DF24addimage** passes the raster data in the *image* parameter and the width and height of the image in the *width* and *height* parameters. The array *image* is assumed to be the width times the height times three bytes in length for each color component. The parameters for **DF24putimage** and **DF24addimage** are further defined in Table 7D.

TABLE 7D **DF24putimage and DF24addimage Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DF24putimage [intn] **(d2pimg)**	filename	char *	character*(*)	Name of file to store the raster image.
	image	VOIDP	<valid numeric data type>	Raster image to be written.
	width	int32	integer	Number of columns in the image.
	height	int32	integer	Number of rows in the image.
DF24addimage [intn] **(d2aimg)**	filename	char *	character*(*)	Name of file to store the raster image.
	image	VOIDP	<valid numeric data type>	Raster image to be written.
	width	int32	integer	Number of columns in the image.
	height	int32	integer	Number of rows in the image.

EXAMPLE 1. **Writing a 24-Bit Raster Image to an HDF File**

In the following examples, **DF24addimage** and **d2aimg** are used to write a 24-bit image to an HDF file named "Example1.hdf." **DF24addimage** assumes row-major order. Therefore, the FORTRAN-77 declaration requires the width (rows) before the height (columns), whereas the C declaration requires the height before the width. The interlace format setting is the default (pixel interlacing).

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

main( )
{

    /* Initialize the image array. */
    static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
    {  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,
      31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
      46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
      61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
      76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 };
    intn status;

    /* Write the 24-bit raster image to the HDF file. */
    status = DF24addimage("Example1.hdf", (VOIDP)raster_data, WIDTH, \
                          HEIGHT);
    }
```

FORTRAN:

```
      PROGRAM WRITE RIS24

       integer status, d2aimg
       integer*4 WIDTH, HEIGHT, PIXEL_DEPTH
       parameter (WIDTH = 5,
      +         HEIGHT = 6,
      +         PIXEL_DEPTH = 3)

       character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

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,
      +     31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
      +     46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
      +     61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
      +     76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 /

C      Write the 24-bit raster image to the file.
       status = d2aimg('Example1.hdf', raster_data, WIDTH,
      +                HEIGHT)

       end
```

7.4.2. Setting the Interlace Format: DF24setil

DF24setil indicates the interlace format to be used for all subsequent write operations. **DF24setil** changes the default setting from pixel interlacing to the selected format. When the format is set, it acts as the default until it is reset by another call to **DF24setil**. To change the default interlace format , the calling program must contain the following routines:

```
C:          status = DF24setil(il);
            status = DF24addimage(filename, image, width, height);

FORTRAN:    status = d2setil(il)
```

DF24setil takes *il* as its only parameter. Valid values for *il* are `DFIL_PIXEL`, `DFIL_LINE`, and `DFIL_PLANE`. The parameters for **DF24setil** are further defined in Table 7E

EXAMPLE 2. **Writing 24-Bit Raster Images Using Scan-plane Interlacing**

In the following examples, **DF24addimage** is used to write a 24-bit image to an HDF file called "Example2.hdf". The **DF24setil** function used here to change the default format setting from pixel interlacing to scan-plane interlacing.

C:

```
#include "hdf.h"
#include "hcomp.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

main( )
```

```
{

    /* Initialize the image array. */
    static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
    {  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,
    31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
    46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
    61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
    76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 };
    intn status;

    /* Change interlace from pixel to scan-plane. */
    status = DF24setil(DFIL_PLANE);

    /* Write the 24-bit image data to file. */
    status = DF24addimage("Example2.hdf", (VOIDP)raster_data,
                       WIDTH, HEIGHT);

}
```

FORTRAN:

```
      PROGRAM CHANGE INTERLACE

       integer status, d2aimg, d2setil
       integer*4 WIDTH, HEIGHT, PIXEL_DEPTH, DFIL_PLANE
       parameter (WIDTH = 5,
      +    HEIGHT = 6,
      +    PIXEL_DEPTH = 3,
      +    DFIL_PLANE = 2)

       integer raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

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,
      +  31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
      +  46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
      +  61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
      +  76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90  /

C      Change interlace from pixel to scan plane.
       status = d2setil(DFIL_PLANE)

C      Write the 24-bit raster image to the file.
       status = d2aimg('Example2.hdf', raster_data, WIDTH,
      +                 HEIGHT)

       end
```

7.4.3. Compressing Image Data: DF24setcompress and d2sjpeg

DF24setcompress invokes JPEG compression and sets the JPEG quality and baseline options. To store a 24-bit raster image using JPEG compression, the calling program must contain the following function calls:

```
C:          status = DF24setcompress(type, c_info);
            status = DF24addimage(filename, image, width, height);
```

```
FORTRAN:   status = d2scomp(type)
    OR     status = d2sjpeg(quality, baseline)
           status = d2aimg(filename, image, width, height, compress)
```

Notice that the calling sequence for C is different from the calling sequence for FORTRAN-77. Once it is set, the parameter *type* in the **DF24setcompress routine,** or **d2scomp** in FORTRAN-77, routine specifies the compression method that will be used to store the raster images. However, the *c_info* parameter in **DF24setcompress** is missing from **d2scomp** which is a pointer to a structure that contains information specific to the compression method indicated by the *type* parameter. Because data structures of variable size are not supported in FORTRAN-77, a second compression-specific routine (**d2sjpeg**) is required in the FORTRAN-77 calling sequence.

For more information about the *c_info* structure refer to Chapter 6, *8-Bit Raster Images (DFR8 API)*.

Default values for quality and baseline (*quality=75%*, *baseline=on*) are used if *c_info* is a null structure or **d2sjpeg** is omitted. Parameters for **DF24setcompress** and **d24sjpeg** are further described in Table 7E below.

TABLE 7E **DF24setil and DF24setcompress Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DF24setil [intn] **(d2sil)**	il	int32	integer	Interlace format to be set.
DF24setcompress [intn] **(d2scomp)**	type	int32	integer	COMP_JPEG
	c_info	comp_info *	None	Pointer to JPEG information structure.
(d2sjpeg)	quality	None	integer	JPEG compression quality specification.
	baseline	None	integer	JPEG compression baseline specification.

EXAMPLE 3. **Compressing and Writing a 24-Bit Raster Image**

In the following examples, **DF24addimage** and **DF24compress** are used to compress a 24-bit image and write it to an HDF file named "Example2.hdf". Notice that compressing an image in C requires only one function call, whereas compressing an image using FORTRAN-77 requires two. The second FORTRAN-77 call is required because it is not valid to pass a structure as a parameter in FORTRAN-77.

C:

```
#include "hdf.h"
#include "hcomp.h"

#define WIDTH 3
#define HEIGHT 5
#define PIXEL_DEPTH 3

main( )
{
    /* Initialize the image array. */
    static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
            {  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, 31,32,33, 34,35,36,
              37,38,39, 40,41,42, 43,44,45 };
    static comp_info compress_info;
    intn status;

    /* Initialize JPEG compression structure. */
    compress_info.jpeg.quality = 60;
    compress_info.jpeg.force_baseline = 1;

    /* Set JPEG compression for storing the image. */
    status = DF24setcompress(COMP_JPEG, &compress_info);

    /* Write the 24-bit image data to file. */
    status = DF24addimage("Example2.hdf", (VOIDP)raster_data,
                       WIDTH, HEIGHT);

}
```

FORTRAN:

```
      PROGRAM COMPRESS RIS24

      integer d2aimg, d2scomp, d2sjpeg, status
      integer*4 WIDTH, HEIGHT, PIXEL_DEPTH
      parameter(WIDTH = 3,
     +          HEIGHT = 5,
     +          PIXEL_DEPTH = 3)
      character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

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, 31,32,33, 34,35,36,
     +   37,38,39, 40,41,42, 43,44,45  /

C     Set compression.
      status = d2scomp(COMP_JPEG)

C     Set JPEG parameters to quality = 60, and turn compatibility on.
      status = d2sjpeg(60, 1)

C     Write the 24-bit image data to the HDF file.
      status = d2aimg('Example2.hdf', raster_data, WIDTH, HEIGHT)
      end
```

7.5. Reading 24-Bit Raster Images

The DF24 programming model for reading a 24-bit raster image set is as follows:

1. Determine the dimensions for an image if necessary.
2. Specify the interlace format to use when reading the image. (optional)
3. Read the image data from the file.

7.5.1. Reading a Raster Image: DF24getimage

If the dimensions and interlace format of the image are known, **DF24getimage** is the only function call required to read a raster image. If a file is being opened for the first time, **DF24getimage** returns the first image in the file. Additional calls will return successive images in the file, therefore images are read in the same order in which they were written to the file. Normally, **DF24getdims** and **DF24getil** are called before **DF24getimage** so that, if necessary, space allocations and interlace format for the image can be checked and the dimensions verified. If this information is already known, both function calls may be omitted.

The syntax of the **DF24getimage** routine is as follows:

```
C:          status = DF24getimage(filename, image, width, height);

FORTRAN:    status = d2gimg(filename, image, width, height)
```

DF24getimage retrieves the next 24-bit image from the HDF file specified by the *filename* parameter. If the image is compressed, **DF24getimage** decompresses it and places it in memory at the location pointed to by the *image* parameter. **DF24getimage** assumes the data is stored using pixel interlacing. The space allocated to hold the image is specified by the *width* and *height* parameters and may be larger than the actual image.The parameters for **DF24getimage** are further defined below. Table 7F

7.5.2. Determining the Dimensions of an Image: DF24getdims

DF24getdims opens a named file, finds the next image or the first image if the file is being opened for the first time, retrieves the dimensions of the image, then determines the interlace format of the image. Images are read in the order they were written.

To determine the dimensions and interlace format for an image, the calling program must call the following routines:

```
C:          status = DF24getdims(filename, width, height, il);
            status = DF24getimage(filename, image, width, height);

FORTRAN:    status = d2gdim(filename, width, height, il)
            status = d2gimg(filename, image, width, height)
```

DF24getdims takes four parameters: *filename*, *width*, *height*, and *il*. It retrieves dimension and interlace format information of the next 24-bit image stored in the HDF file specified by the *filename* parameter. The width and height are returned in the space pointed to by the *width* and *height* parameters respectively. The *il* parameter is used to determine the interlace format. The parameters for **DF24getdims** are further defined below. (See TABLE 7F)

7.5.3. Modifying the Interlacing of an Image: DF24reqil

DF24reqil specifies an interlace format to be used when reading a 24-bit image from a file into memory. Regardless of what interlace format is used to store the image, **DF24reqil** forces the image to be loaded into memory using the specified interlace format.

To set or reset the interlace format, the calling program should call the following routines:

```
C:          status = DF24reqil(il);
            status = DF24getimage(filename, image, width, height);

FORTRAN:    status = d2reqil(il)
            status = d2gimg(filename, image, width, height)
```

DF24reqil takes *il* as its only parameter. Valid *il* values are `DFIL_PIXEL`, `DFIL_LINE` and `DFIL_PLANE`. As a call to **DF24reqil** may require a substantial reordering of the data, a much slower I/O performance than would be achieved if the interlace format wasn't reset may result.

The parameters of **DF24reqil** is further defined below. (See TABLE 7F)

TABLE 7F

DF24getimage, DF24getdims and DF24reqil Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DF24getimage [intn] **(d2gimg)**	filename	char *	character*(*)	Name of the HDF file containing the raster image.
	image	VOIDP	<valid numeric data type>	Buffer for the raster image.
	width	int32	integer	Width of the raster image buffer.
	height	int32	integer	Height of the raster image buffer.
DF24getdims **[intn]** **(d2gdims)**	filename	char *	character*(*)	Name of HDF file containing the raster image.
	width	int32 *	integer	Pointer to the number of columns in the raster image.
	height	int32 *	integer	Pointer to the number of rows in the raster image.
	il	intn	integer	Pointer to the interlace format of the raster image.
DF24reqil **[intn]** **(d2reqil)**	il	intn	integer	Pointer to the interlace format of the raster image.

EXAMPLE 4.

Reading a 24-Bit Raster Image from an HDF File

The following examples read a 24-bit image from the "Example2.hdf" HDF file created in Example 2. Although the **DF24getdims** function call is optional, it is included as a demonstration of how to verify the image dimensions and interlace format before reading the image data. If the image dimensions and interlace format are known, only the **DF24getimage** call is required.

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

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

    /* Get the image dimensions from the HDF file. */
    status = DF24getdims("Example2.hdf", &width, &height,
                         &interlace);

    /*
    * Read raster data if the dimensions are
    * correct.
```

```
      */
      if (width <= WIDTH && height <= HEIGHT)
        status = DF24getimage("Example2.hdf", (VOIDP)raster_data,
                         width, height);
}
```

FORTRAN:

```
      PROGRAM READ RIS24

      integer d2gimg, d2gdims, status, width, height, interlace
      integer*4 X_LENGTH, Y_LENGTH, PIXEL_DEPTH
      parameter(X_LENGTH = 5, Y_LENGTH = 6, PIXEL_DEPTH = 3)
      integer raster_data(PIXEL_DEPTH, X_LENGTH, Y_LENGTH)

C     Read the dimensions raster image.
      status = d2gdims('Example2.hdf', width, height, interlace)

C     Read image data from the HDF file if the dimensions are
C     correct.
      if (width .eq. X_LENGTH .and. height .eq. Y_LENGTH) then
       status = d2gimg('Example2.hdf', raster_data, width, height)
      endif

      end
```

7.5.4. Reading a 24-Bit Raster Image with a Given Reference Number: DF24readref

DF24readref is used to access specific images stored in files containing multiple raster image sets. It is optionally used before **DF24getimage**. **DF24readref** can be used in connection with vgroups, which identify their members by tag/reference number pairs. See Chapter 5, *Vgroups (V API)*, for a discussion of vgroups and tag/reference number pairs.

To access a specific raster image set, use the following sequence of routine calls:

```
C:         status = DF24readref(filename, ref);
           status DF24getimage(filename, image, width, height);

FORTRAN:   status = d2rref(filename, ref)
           status = d2gimg(filename, image, width, height)
```

DF24readref sets the reference number for the next read operation performed on the HDF file *filename* to the reference number contained in *ref*. Because reference numbers are not always assigned in sequence, it is not guaranteed that a reference number represents the location of the image in the file.

The parameters of **DF24readref** are further described in the following table.

TABLE 7G

DF24readref Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DF24readref [intn] **(d2rref)**	filename	char *	character*(*)	Name of HDF file containing the raster image.
	ref	uint16	integer	Reference number for the next call to **DF24getimage**.

7.5.5. Specifying that the Next Image Read to be the First 24-Bit Raster Image in the File: DF24restart

DF24restart causes the next call to **DF24getimage** or **DF24getdims** to read from the first raster image set in the file, rather than the RIS24 following the one that was most recently read. Use the following call to invoke **DF24restart**:

```
C:          status = DF24restart( );

FORTRAN:    status = d2first( )
```

TABLE 7H

DF24restart Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DF24restart [intn] **(d2first)**	None	None	None	None.

7.6. 24-Bit Raster Image Information Retrieval Routines

7.6.1. Querying the Total Number of Images in a File: DF24nimages

DF24nimages returns the total number of 24-bit raster image sets in a file, and has the following syntax:

```
C:          num_of_images = DF24nimages(filename);

FORTRAN:    num_of_images = d2nimg(filename)
```

TABLE 7I **DF24nimages Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DF24nimages [intn] **(d2nimg)**	filename	char *	character*(*)	Name of the HDF file.

7.6.2. Querying the Reference Number of the Most Recently Accessed 24-Bit Raster Image: DF24lastref

DF24lastref returns the reference number of the 24-bit raster image most recently read or written. This routine is used for attaching annotations to images and adding images to vgroups. (See Chapter 5, *Vgroups (V API)* and Chapter 10, *Annotations (AN API)* for details on how to use reference numbers in connection with these applications.

The following calling sequence uses **DF24lastref** to find the reference number of the RIS24 most recently added to an HDF file:

```
C:         status = DF24addimage(filename, image, width, height, compress);
           lastref = DF24lastref( );

FORTRAN:   status = d2aimg(filename, image, width, height, compress)
           lastref = d2lref( )
```

DF24putimage or **DF24getimage** can be used in place of **DF24addimage** with similar results.

TABLE 7J **DF24lastref Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DF24lastref [uint16] **(d2lref)**	filename	None	None	None.

CHAPTER 8 --

General Raster Images (GR API)

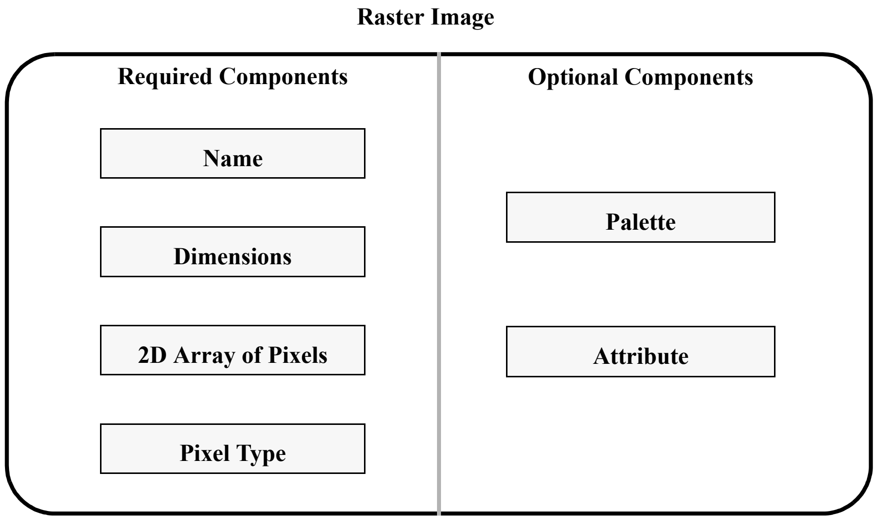
8.1. Chapter Overview

This chapter describes the general raster (GR) data model, the GR interface (also called the GR API), and the interface routines used to manipulate GR data objects. The GR data model is designed to provide a flexible means of manipulating raster images. There were two other interfaces that worked with raster images, the DFR8 interface (Chapter 6, *8-Bit Raster Images (DFR8 API)*) and the DF24 interface (Chapter 7, *24-bit Raster Images (DF24 API)*) but the GR interface supersedes them.

8.2. The GR Data Model

HDF users familiar with the SD interface will find certain aspects of the GR data model similar to the SD data model. The interfaces are similar in that both interfaces support data storage in multiple files, attributes, compression, and chunking. They are dissimilar in that palettes can be created and attached to an image through GR interface routines, customized dimension information is not supported in the GR interface, and GR dataset chunking is constrained to two dimensions.

FIGURE 8a **GR Data Set Contents**



The terms ***GR data set***, ***raster image***, and ***image*** are used interchangeably in this chapter.

Refer to Figure 8a for a graphical overview of the raster image, or GR data set, structure. Note that GR data sets consist of required and optional components.

8.2.1. Required GR Data Set Components

Every GR data set must contain the following components: ***image array***, ***name***, ***pixel type***, and ***dimensions***. The name, dimensions, and pixel type must be supplied by the user at the time the GR data set is defined.

Image Array

An ***image array*** is a two-dimensional array of pixels. This is the primary data component of the GR model and will be discussed later in this section; it can be compressed, chunked, and/or stored in external files. Refer to Section "*Compressing Raster Images: GRsetcompress"* for a description of raster image compression and Section "*External File Operations Using the GR Interface"* for a description of external image storage.

A raster image has an index and a reference number associated with it. The ***index*** is a non-negative integer that describes the relative position of the raster image in the file. A valid index ranges from 0 to the total number of images in the file minus 1. The ***reference number*** is a unique positive integer assigned to the raster image by the GR interface when the image is created. Various GR interface routines can be used to obtain an image index or reference number depending on the available information about the raster image. The index can also be determined if the sequence in which the images are created in the file is known.

In the GR interface, a ***raster image identifier*** uniquely identifies a raster image within the file. The identifier is generated by the GR interface access routines when a new GR data set is created or an existing one is selected. The identifier is then used by other GR interface routines to access the raster image until the access to this image is terminated. For an existing raster image, the index of the image can be used to obtain the identifier.

Image Array Name

Each image array has a ***name*** consisting of a string of case-sensitive alphanumeric characters. The name must be provided by the calling program at the time the image is created, and cannot be changed afterward. Image array names do not have to be unique within a file, but if they are not it can be difficult to distinguish among the raster images in the file.

Pixels and Pixel Type

Each element in an image array corresponds to one ***pixel*** and each pixel can consist of a number of color component values or ***pixel components***, e.g., Red-Green-Blue or RGB, Cyan-Magenta-Yellow-Black or CMYK, etc. Pixel components can be represented by different methods (8-bit lookup table or 24-bit direct representation, graphically depicted by Figure 6a and Figure 7b, respectively) and may have different data types.

The data type of pixel components and the number of components in each pixel are collectively known as the ***pixel type***. The GR data model supports all of the HDF-supported data types. A list of these data types appears provided in Table 2F.

Pixels can be composed of any number of components.

Dimensions

Image array ***dimensions*** specify the shape of the image array. A raster image array has two limited dimensions. The size of each dimension must be specified at the creation of the image and must be greater than 0.

The GR library does not allow the HDF user to add attributes to a dimension or to set dimension scale.

8.2.2. Optional GR Data Set Components

There are two types of optional components available for inclusion in a GR data set: ***palettes*** and ***attributes***. These components are only created when specifically requested by the calling program; the GR interface does not provide predefined palettes or attributes.

Palettes

Palettes are lookup tables attached to images and define a set of color values for each pixel value in the image array. The GR interface provides similar capabilities for storing and manipulating palettes as the DFP interface described in Chapter 9, *Palettes (DFP API)*. However, the DFP interface is restricted to single-file operations while the GR interface allows multifile palette operations.

Eventually, all palette manipulation functionality will reside only within the GR interface. In the meantime, the single-file DFP routines are fully compatible with palettes created with the GR palette routines. The GR palette routines are described in Section "*Reading and Writing Palette Data Using the GR Interface"*.

Attributes

Attributes contain auxiliary information about a file, a raster image, or both. The concept of attributes is described in Chapter 3, *Scientific Data Sets (SD API)*.

The GR interface does not support dimension attributes.

8.3. The GR Interface

The GR consists of routines for storing, retrieving, and manipulating the data in GR data sets.

8.3.1. GR Interface Routines

All C routine names in the GR interface have the prefix **GR** and the equivalent FORTRAN-77 routine names are prefaced by **mg**. All GR routines are classifiable within one of the following categories:

- ***Access routines*** initialize and terminate access to the GR interface and raster images.
- ***Raster image manipulation routines*** modify the data and metadata contained in a GR data set.
- ***LUT manipulation routines*** modify the palettes, also called color lookup tables or LUTs, contained in a GR data set.
- ***Maintenance routines*** create the data and metadata contained in a GR data set and modify global settings governing the format of the stored data.
- ***Inquiry routines*** return information about data contained in a GR data set.
- ***Chunking routines*** are used to define data chunking parameters, to retrieve chunking information, and to write and read chunked GR data sets.

The GR routines are listed in the following table and described further in subsequent sections of this chapter.

TABLE 8A

GR Library Routines

Purpose	Routine Name7		Description
	C	FOR-TRAN-77	
Access	`GRstart`	`mgstart`	Initializes the GR interface (Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate"*)
	`GRcreate`	`mgcreat`	Creates a new raster image (Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate"*)
	`GRselect`	`mgselct`	Selects the raster image (Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate"*)
	`GRendaccess`	`mgendac`	Terminates access to the raster image (Section "*Terminating Access to Images and Files: GRendaccess and GRend"*)
	`GRend`	`mgend`	Terminates access to the GR interface (Section "*Terminating Access to Images and Files: GRendaccess and GRend"*)
Raster Image Manipulation	`GRgetattr`	`mggnatt/ mggcatt`	Reads an attribute of a raster image or a file (Section "*Reading User-defined Attributes: GRgetattr"*)
	`GRidtoref`	`mgid2rf`	Maps a raster image identifier to a reference number (Section "*Obtaining the Reference Number of a Raster Image from Its Identifier: GRidtoref*")
	`GRnametoindex`	`mgn2ndx`	Maps the name of a raster image name to an index (Section "*Obtaining the Index of a Raster Image from Its Name: GRnametoindex"*)
	`GRreadimage`	`mgrdimg/ mgrcimg`	Reads raster image data (Section "*Reading Data from an Image: GRreadimage"*)
	`GRreftoindex`	`mgr2idx`	Maps the reference number of a raster image to its index (Section "*Obtaining the Index of a Raster Image from Its Reference Number: GRreftoindex"*)
	`GRsetattr`	`mgsnatt/ mgscatt`	Assigns an attribute to a raster image or a file (Section "*Setting User-defined Attributes: GRsetattr"*)
	`GRwriteimage`	`mgwrimg/ mgwcimg`	Writes raster image data (Section "*Writing Raster Images: GRwriteimage"*)
	`GRreqimageil`	`mgrimil`	Sets the interlace mode of the image read for subsequent read operations (Section "*Setting the Interlace Mode for an Image Read: GRreqimageil"*)
LUT Manipulation	`GRgetlutid`	`mggltid`	Gets a palette identifier given the palette's index (Section "*Obtaining a Palette Identifier: GRgetlutid"*)
	`GRluttoref`	`mglt2rf`	Maps a palette identifier to a reference number (Section "*Obtaining the Reference Number of a Specified Palette: GRluttoref*")
	`GRreadlut`	`mgrdlut/ mgrclut`	Reads palette data from a raster image (Section "*Reading Palette Data: GRreadlut"*)
	`GRwritelut`	`mgwrlut/ mgwclut`	Writes palette data to a raster image (Section "*Writing Palette Data: GRwritelut"*)
	`GRreqlutil`	`mgrltil`	Sets the interlace mode of the next palette for subsequent read operations (Section "*Setting the Interlace Mode for an Image Read: GRreqimageil"*)
	`GRgetnluts`	`mggnluts`	Retrieves the number of palettes associated with an image (See the *HDF Reference Manual*)

Miscenlaneous	`GRsetcompress`	`mgscomp`	Specifies whether the raster image will be stored in a file as a compressed raster image (Section "*Compressing Raster Images: GRsetcompress"*)
	`GRgetcompinfo`	`mggcom-press`	Retrieves image compression type and compression information (Section "*Obtaining Compression Information for a Raster Image: GRgetcompinfo"*)
	`GRsetexternal-file`	`mgsxfil`	Specifies that the raster image will be written to an external file (Section "*Creating a Raster Image in an External File: GRsetexternalfile"*)
	`GRsetaccess-type`	`nmgsactp`	Sets the access for an RI to be either serial or parallel I/O ()
Inquiry	`GRattrinfo`	`mgatinf`	Retrieves information about an attribute (Section "*Querying User-Defined Attributes: GRfindattr and GRattrinfo"*)
	`GRfindattr`	`mgfndat`	Finds the index of a data object's attribute given an attribute name (Section "*Querying User-Defined Attributes: GRfindattr and GRattrinfo"*)
	`GRfileinfo`	`mgfinfo`	Retrieves the number of raster images and the number of global attributes in the file (Section "*Obtaining Information about the Contents of a File: GRfileinfo"*)
	`GRgetiminfo`	`mggiinf`	Retrieves general information about a raster image (Section "*Obtaining Information about an Image: GRgetiminfo"*)
	`GRgetlutinfo`	`mgglinf`	Retrieves information about a palette (Section "*Obtaining Palette Information: GRgetlutinfo"*)
Chunking	`GRsetchunk`	`mgschnk`	Creates chunked raster image (Section "*Making a Raster Image a Chunked Raster Image: GRsetchunk"*)
	`GRgetchunkinfo`	`mggichnk`	Retrieves information about a chunked raster image (Section "*Obtaining Information about a Chunked Raster Image: GRgetchunkinfo"*)
	`GRsetchunk-cache`	`mgscchnk`	Sets maximum number of chunks to be cached (Section "*Setting the Maximum Number of Chunks in the Cache: GRsetchunkcache"*)
	`GRreadchunk`	`mgrchnk/ mgrcchnk`	Reads a data chunk from a chunked raster image (pixel-interlace only) (Section "*Reading a Chunked Raster Image: GRreadchunk"*)
	`GRwritechunk`	`mgwchnk/ mgwcchnk`	Writes a data chunk to a chunked raster image (pixel-interlace only) (Section "*Writing a Chunked Raster Image: GRwritechunk"*)

8.4. Header Files Required by the GR Interface

The header file "hdf.h" must be included in any program that utilizes GR interface routines.

8.5. Programming Model for the GR Interface

As with the SD interface, the GR interface relies on the calling program to initiate and terminate access to files and data sets to support multifile access. The GR programming model for accessing a raster image is as follows:

1. Open an HDF file.
2. Initialize the GR interface.
3. Open an existing raster image or create a new raster image.
4. Perform desired operations on the raster image.
5. Terminate access to the raster image.
6. Terminate access to the GR interface by disposing of the interface identifier.
7. Close the HDF file.

To access a single raster image data set in an HDF file, the calling program must contain the following calls:

```
C:          file_id = Hopen(filename, access_mode, n_dds_block);
            gr_id = GRstart(file_id);

            ri_id = GRselect(gr_id, ri_index);
    OR      ri_id = GRcreate(gr_id, name, n_comps, data_type, interlace_mode,
                          dim_sizes);

            <Optional operations>
            status = GRendaccess(ri_id);
            status = GRend(gr_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, access_mode, n_dds_block)
            gr_id = mgstart(file_id)

            ri_id = mgselct(gr_id, ri_index)
    OR      ri_id = mgcreat(gr_id, name, n_comps, data_type, interlace_mode,
                          dim_sizes)

            <Optional operations>
            status = mgendac(ri_id)
            status = mgend(gr_id)
            status = hclose(file_id)
```

To access several files at the same time, a calling program must obtain a separate interface identifier for each file to be opened. Similarly, to access more than one raster image, a calling program must obtain a separate data set identifier for each data set.

Because every file and raster image is assigned its own identifier, the order in which files and data sets are accessed is very flexible as long as all file and raster image identifiers are individually discarded before the end of the calling program.

8.5.1. Accessing Images and Files: GRstart, GRselect, and GRcreate

In the GR interface, **Hopen** opens the files and **GRstart** initiates the GR interface. Note the contrast to the SD interface, where **SDstart** performs both tasks. For information on the use of **Hopen**, refer to Chapter 2, *HDF Fundamentals.* For information on **SDstart**, refer to Chapter 3, *Scientific Data Sets (SD API)*.

GRstart initializes the GR interface and must be called once after **Hopen** and before any other GR routines are called. It takes one argument, *file_id*, the file identifier returned by **Hopen**, and returns the interface identifier *gr_id* or `FAIL` (or `-1`) upon unsuccessful completion. **Hopen** and **GRstart** can be called several times to access more than one file.

GRselect specifies the given image as the current image to be accessed. It takes two arguments, the GR interface identifier *gr_id* and the raster image index *ri_index*, and returns the raster image identifier *ri_id* or `FAIL` (or `-1`) upon unsuccessful completion. The GR interface identifier is returned by **GRstart**. The raster image index specifies the position of the image relative to the beginning of the file; it is zero-based, meaning that the index of the first image in the file is `0`. The index of a raster image can be obtained from the image's name using the routine **GRnametoindex** or from the image's reference number using **GRreftoindex**. These routines are discussed in Section "*Obtaining the Index of a Raster Image from Its Reference Number: GRreftoindex"* and Section "*Obtaining the Index of a Raster Image from Its Name: GRnametoindex"*. The index value must be less than the total number of raster images in the file; that number can be obtained using **GRfileinfo**, described in Section "*Obtaining Information about the Contents of a File: GRfileinfo"*.

The parameters for **GRstart** and **GRselect** are further defined in Table 8B.

GRcreate defines a new raster image using the arguments *gr_id*, *name*, *n_comps*, *data_type*, *interlace_mode*, and *dim_sizes*. Once a data set is created, you cannot change its name, data type, dimension, or number of components. **GRcreate** does not actually write the image to the file; this occurs only when **GRendaccess** is called. Thus, failing to call **GRendaccess** properly will cause a loss of data.

The buffer *name* contains the name of the image; it must not exceed `H4_MAX_GR_NAME` (or `256`). The parameter *n_comps* specifies the number of pixel components in the raster image; it must have a value of at least 1. The parameter *data_type* specifies the data type of the image data; it can be any of the data types supported by the HDF library. The HDF supported data type are defined in the header file "hntdefs.h" and listed in Table 2F.

The parameter *interlace_mode* specifies the interlacing in which the raster image is to be written; it can be set to either `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`), or `MFGR_INTERLACE_COMPONENT` (or `2`). These definitions respectively correspond to pixel interlacing, line interlacing, and component interlacing. The first two interlacing modes are illustrated for the instance of 24-bit pixel representation in Figure 7c of Chapter 7, *24-bit Raster Images (DF24 API)*. Component interlacing, as the name implies, describes interlacing raster data by color component. Note that images created with the GR interface are actually written to disk in pixel interlace mode; any user-specified interlace mode is stored in the file with the image and the image is automatically converted to that mode when it is read with a GR interface function.

The parameter *dim_sizes* specifies the size of the two dimensions of the image. The dimension sizes must be specified; their values must be at least 1.

GRcreate returns the value of the raster image identifier if successful or `FAIL` (or `-1`) otherwise. The parameters for **GRstart**, **GRselect**, and **GRcreate** are further defined in (See TABLE 8B).

8.5.2. Terminating Access to Images and Files: GRendaccess and GRend

GRendaccess disposes of the raster image identifier *ri_id* and terminates access to the data set initiated by the corresponding call to **GRselect** or **GRcreate**. The calling program must make one **GRendaccess** call for every **GRselect** or **GRcreate** call made during its execution. Failing to call **GRendaccess** for each call to **GRselect** or **GRcreate** may result in a loss of data.

GRend disposes of the GR interface identifier *gr_id* and terminates access to the GR interface initiated by the corresponding call to **GRstart**. The calling program must make one **GRend** call for every **GRstart** call made during its execution; failing to call **GRend** for each **GRstart** may result in a loss of data.

GRendaccess and **GRend** return `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of these routines are further defined in Table 8B.

Hclose terminates access to an HDF file and should only be called after **GRend** has been called properly. Refer to Chapter 2, *HDF Fundamentals*, for a description of **Hclose**.

TABLE 8B **GRstart, GRselect, GRcreate, GRendaccess, and GRend, Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
GRstart [int32] **(mgstart)**	file_id	int32	integer	File identifier
GRselect [int32] **(mgselct)**	gr_id	int32	integer	GR interface identifier
	ri_index	int32	integer	Position of the raster image within the file
GRcreate [int32] **(mgcreat)**	gr_id	int32	integer	GR interface identifier
	name	char *	character*(*)	Name of the image
	n_comps	int32	integer	Number of components in each pixel
	data_type	int32	integer	Data type of the pixel component
	interlace_mode	int32	integer	Interlace mode to be used when writing to the data set
	dim_sizes	int32 [2]	integer (2)	Array defining the size of both dimensions
GRendaccess [intn] **(mgendac)**	ri_id	int32	integer	Raster image identifier
GRend [intn] **(mgend)**	gr_id	int32	integer	GR interface identifier

8.6. Writing Raster Images

A raster image can be written partially or entirely. Partial writing includes writing to a contiguous region of the image and writing to selected locations in the image according to patterns defined by the user. This section describes the routine **GRwriteimage** and how it can write data to part of an image or to an entire image. The section also illustrates the concepts of compressing raster images and the use of external files to store image data.

8.6.1. Writing Raster Images: GRwriteimage

GRwriteimage is used to either completely or partially fill an image array.

Writing data to an image array involves the following steps:

1. Open a file and initialize the GR interface.
2. Select an existing raster image or create a new one.
3. Write data to the image array.
4. Terminate access to the raster image.
5. Terminate access to the GR interface and close the file.

The calling program must contain the following sequence of calls:

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

            ri_id = GRselect(gr_id, ri_index);
     OR     ri_id = GRcreate(gr_id, name, n_comps, number_type, interlace_mode,
                          dim_sizes);
```

```
                status = GRwriteimage(ri_id, start, stride, edges, data);
                status = GRendaccess(gr_id);
                status = GRend(ri_id);
                status = Hclose(file_id);

     FORTRAN:   file_id = hopen(filename, access_mode, num_dds_block)
                gr_id = mgstart(file_id)

                ri_id = mgselct(gr_id, ri_index);
          OR    ri_id = mgcreat(gr_id, name, n_comps, number_type, interlace_mode,
                                dim_sizes);

                status = mgwrimg(ri_id, start, stride, edges, data)
          OR    status = mgwrcmg(ri_id, start, stride, edges, data)

                status = mgendac(ri_id)
                status = mgend(gr_id)
                status = hclose(file_id)
```

As with SD arrays, whole raster images, subsamples, and slabs can be written. The data to be written is defined by the values of the parameters *start*, *stride*, and *edges*, which correspond to the coordinate location of the data origin, number of values to be skipped along each dimension during write operation, and number of elements to be written along each dimension.

The array *start* specifies the starting location of the data to be written. Valid values of each element in the array *start* are `0` to the size of the corresponding raster image dimension - 1. The first element of the array *start* specifies an offset from the beginning of the array *data* along the fastest-changing dimension, which is the second dimension in C and the first dimension in FORTRAN-77. The second element of the array *start* specifies an offset from the beginning of the array *data* along the second fastest-changing dimension, which is the first dimension in C and the second dimension in FORTRAN-77. For example, if the first value of the array *start* is 2 and the second value is 3, the starting location of the data to be written is at the fourth row and third column in C, and at the third row and fourth column in FORTRAN-77. Note that the correspondence between elements in the array *start* and the raster image dimensions in the GR interface is different from that in the SD interface. See Section "*Reading Data from an SDS Array: SDreaddata"* on **SDreaddata** for an example of this.

The array *stride* specifies the writing pattern along each dimension. For example, if one of the elements of the array *stride* is 1, then every element along the corresponding dimension of the array *data* will be written. If one of the elements of the *stride* array is 2, then every other element along the corresponding dimension of the array *data* will be written, and so on. The correspondence between elements of the array *stride* and the dimensions of the array *data* is the same as described above for the array *start*.

Note that the FORTRAN-77 version of **GRwriteimage** has two routines; **mgwrimg** writes buffered numeric data and **mgwcimg** writes buffered character data.

GRwriteimage returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **GRwriteimage** are described in Table 8C.

TABLE 8C

GRwriteimage Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parame-ter	Parameter Type		Description
		C	FORTRAN-77	
GRwriteimage [intn] **(mgwrimg/ mgwcimg)**	ri_id	int32	integer	Raster image identifier returned by **GRcreate**
	start	int32 [2]	integer (2)	Array containing the x,y-coordinate location where the write will start for each dimension
	stride	int32 [2]	integer (2)	Array containing the number of data locations the current location is to be moved forward before the next write
	edges	int32 [2]	integer (2)	Array containing the number of data elements that will be written along each dimension
	data	VOIDP	<valid numeric data type>(*)/ character(*)	Buffer for the image data to be written

EXAMPLE 1.

Creating and Writing a Raster Image

This example illustrates the use of the routines **Hopen**/**hopen**, **GRstart**/**mgstart**, **GRcreate**/ **mgcreat**, **GRwriteimage**/**mgwrimg**, **GRendaccess**/**mgendac**, **GRend**/**mgend**, and **Hclose**/ **hclose** to create an HDF file and store a raster image in it.

In this example, the program creates the HDF file called “General_RImages.hdf” and a raster image in the file. The image created is of size 5x10 and named “Image Array 1", and has data of the int16 data type, 2 components, and interlace mode `MFGR_INTERLACE_PIXEL`. Then the program writes the image data, terminates access to the image and the GR interface, and closes the file.

C:

```
#include "hdf.h"

#define  FILE_NAME     "General_RImages.hdf"
#define  IMAGE_NAME    "Image Array 1"
#define  X_LENGTH      10    /* number of columns in the image */
#define  Y_LENGTH      5     /* number of rows in the image */
#define  N_COMPS       2     /* number of components in the image */

main( )
{
   /************************ Variable declaration *************************/

   intn  status;         /* status for functions returning an intn */
   int32 file_id,        /* HDF file identifier */
         gr_id,          /* GR interface identifier */
         ri_id,          /* raster image identifier */
         start[2],       /* start position to write for each dimension */
         edges[2],       /* number of elements to be written
                           along each dimension */
         dim_sizes[2],   /* dimension sizes of the image array */
         interlace_mode, /* interlace mode of the image */
         data_type,      /* data type of the image data */
         i, j;
   int16 image_buf[Y_LENGTH][X_LENGTH][N_COMPS];

   /********************* End of variable declaration *********************/
   /*
   * Create and open the file.
   */
   file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

   /*
   * Initialize the GR interface.
   */
   gr_id = GRstart (file_id);

   /*
   * Set the data type, interlace mode, and dimensions of the image.
   */
   data_type = DFNT_INT16;
   interlace_mode = MFGR_INTERLACE_PIXEL;
   dim_sizes[0] = X_LENGTH;
   dim_sizes[1] = Y_LENGTH;

   /*
   * Create the raster image array.
   */
   ri_id = GRcreate (gr_id, IMAGE_NAME, N_COMPS, data_type,
                     interlace_mode, dim_sizes);

   /*
   * Fill the image data buffer with values.
   */
   for (i = 0; i < Y_LENGTH; i++)
   {
      for (j = 0; j < X_LENGTH; j++)
      {
         image_buf[i][j][0] = (i + j) + 1;     /* first component */
         image_buf[i][j][1] = (i + j) + 1;     /* second component */
      }
    }

   /*
   * Define the size of the data to be written, i.e., start from the origin
   * and go as long as the length of each dimension.
   */
   start[0] = start[1] = 0;
   edges[0] = X_LENGTH;
   edges[1] = Y_LENGTH;

   /*
   * Write the data in the buffer into the image array.
   */
   status = GRwriteimage(ri_id, start, NULL, edges, (VOIDP)image_buf);

   /*
   * Terminate access to the raster image and to the GR interface and,
   * close the HDF file.
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

FORTRAN:

```
      program create_raster_image
    implicit none
```

```
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*13 IMAGE_NAME
      integer      X_LENGTH
      integer      Y_LENGTH
      integer      N_COMPS
C
      parameter (FILE_NAME  = 'General_RImages.hdf',
     +           IMAGE_NAME = 'Image Array 1',
     +           X_LENGTH   = 10,
     +           Y_LENGTH   = 5,
     +           N_COMPS    = 2)
      integer DFACC_CREATE, DFNT_INT16, MFGR_INTERLACE_PIXEL
      parameter (DFACC_CREATE = 4,
     +           DFNT_INT16   = 22,
     +           MFGR_INTERLACE_PIXEL = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgcreat, mgwrimg, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer gr_id, ri_id, num_type, interlace_mode
      integer start(2), stride(2), edges(2), dimsizes(2)
      integer i, j, k
      integer*2  image_buf(N_COMPS, X_LENGTH, Y_LENGTH)
C
C**** End of variable declaration ************************************
C
C
C     Create and open the file.
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Set the number type, interlace mode, and dimensions of the image.
C
      num_type = DFNT_INT16
      interlace_mode = MFGR_INTERLACE_PIXEL
      dimsizes(1) = X_LENGTH
      dimsizes(2) = Y_lENGTH
C
C     Create the raster image array.
C
      ri_id = mgcreat(gr_id, IMAGE_NAME, N_COMPS, num_type,
     +                interlace_mode, dimsizes)
C
C     Fill the image data buffer with values.
C
      do 30 i = 1, Y_LENGTH
         do 20 j = 1, X_LENGTH
            do 10 k = 1, N_COMPS
               image_buf(k,j,i) = (i+j) - 1
```

```
10          continue
20       continue
30    continue

C
C     Define the size of the data to be written, i.e., start from the origin
C     and go as long as the length of each dimension.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Write the data in the buffer into the image array.
C
      status = mgwrimg(ri_id, start, stride, edges, image_buf)

C
C     Terminate access to the raster image and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

EXAMPLE 2.

Modifying an Existing Raster Image

This example illustrates the use of the routines **GRselect**/**mgselct** to obtain an existing raster image and **GRwrite**/**mgwrimg** to modify image data.

In this example, the program selects the only raster image in the file "General_RImages.hdf" created and written in Example 1, and modifies image data. The program also creates another raster image that is named "Image Array 2" and has 3 components with dimension size of 4x6, data type of `DFNT_CHAR8`, and interlace mode of `MFGR_INTERLACE_PIXEL`.

C:

```
#include "hdf.h"

#define  FILE_NAME    "General_RImages.hdf"
#define  X1_LENGTH    5     /* number of columns in the first image
                               being modified */
#define  Y1_LENGTH    2     /* number of rows in the first image
                               being modified */
#define  N1_COMPS     2     /* number of components in the first image */
#define  IMAGE1_NAME  "Image Array 1"
#define  IMAGE2_NAME  "Image Array 2"
#define  X2_LENGTH    6     /* number of columns in the second image */
#define  Y2_LENGTH    4     /* number of rows in the second image */
#define  N2_COMPS     3     /* number of components in the second image */

main( )
{
   /************************* Variable declaration **************************/

   intn  status;         /* status for functions returning an intn */
   int32 file_id,        /* HDF file identifier */
         gr_id,          /* GR interface identifier */
```

```
      ri1_id,         /* raster image identifier */
      start1[2],      /* start position to write for each dimension */
      edges1[2],      /* number of elements to be written along
                        each dimension */
      ri2_id,         /* raster image identifier */
      start2[2],      /* start position to write for each dimension */
      edges2[2],      /* number of elements to be written along
                        each dimension */
      dims_sizes[2],  /* sizes of the two dimensions of the image array */
      data_type,      /* data type of the image data */
      interlace_mode; /* interlace mode of the image */
int16 i, j;           /* indices for the dimensions */
int16 image1_buf[Y1_LENGTH][X1_LENGTH][N1_COMPS]; /* data of first image */
char  image2_buf[Y2_LENGTH][X2_LENGTH][N2_COMPS]; /* data of second image*/

/********************** End of variable declaration **********************/

/*
* Open the HDF file for writing.
*/
file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

/*
* Initialize the GR interface.
*/
gr_id = GRstart (file_id);

/*
* Select the first raster image in the file.
*/
ri1_id = GRselect (gr_id, 0);

/*
* Fill the first image data buffer with values.
*/
for (i = 0; i < Y1_LENGTH; i++)
{
   for (j = 0; j < X1_LENGTH; j++)
   {
      image1_buf[i][j][0] = 0;  /* first component */
      image1_buf[i][j][1] = 0;  /* second component */
   }
 }

/*
* Define the size of the data to be written, i.e., start from the origin
* and go as long as the length of each dimension.
*/
start1[0] = start1[1] = 0;
edges1[0] = X1_LENGTH;
edges1[1] = Y1_LENGTH;

/*
* Write the data in the buffer into the image array.
*/
status = GRwriteimage (ri1_id, start1, NULL, edges1, (VOIDP)image1_buf);


/*
* Set the interlace mode and dimensions of the second image.
*/
data_type = DFNT_CHAR8;
interlace_mode = MFGR_INTERLACE_PIXEL;
```

```
   dims_sizes[0] = X2_LENGTH;
   dims_sizes[1] = Y2_LENGTH;

   /*
   * Create the second image in the file.
   */
   ri2_id = GRcreate (gr_id, IMAGE2_NAME, N2_COMPS, data_type,
                               interlace_mode, dims_sizes);

   /*
   * Fill the second image data buffer with values.
   */
   for (i = 0; i < Y2_LENGTH; i++)
   {
      for (j = 0; j < X2_LENGTH; j++)
      {
         image2_buf[i][j][0] = 'A';     /* first component */
         image2_buf[i][j][1] = 'B';     /* second component */
         image2_buf[i][j][2] = 'C';     /* third component */
      }
    }

   /*
   * Define the size of the data to be written, i.e., start from the origin
   * and go as long as the length of each dimension.
   */
   for (i = 0; i < 2; i++) {
      start2[i] = 0;
      edges2[i] = dims_sizes[i];
   }

   /*
   * Write the data in the buffer into the second image array.
   */
   status = GRwriteimage (ri2_id, start2, NULL, edges2, (VOIDP)image2_buf);

   /*
   * Terminate access to the raster images and to the GR interface, and
   * close the HDF file.
   */
   status = GRendaccess (ri1_id);
   status = GRendaccess (ri2_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

FORTRAN:

```
      program modify_image
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*13 IMAGE1_NAME
      integer      X1_LENGTH
      integer      Y1_LENGTH
      integer      N1_COMPS
      character*13 IMAGE2_NAME
      integer      X2_LENGTH
      integer      Y2_LENGTH
      integer      N2_COMPS
C
```

```
      parameter (FILE_NAME   = 'General_RImages.hdf',
     +           IMAGE1_NAME = 'Image Array 1',
     +           IMAGE2_NAME = 'Image Array 2',
     +           X1_LENGTH   = 5,
     +           Y1_LENGTH   = 2,
     +           N1_COMPS    = 2,
     +           X2_LENGTH   = 6,
     +           Y2_LENGTH   = 4,
     +           N2_COMPS    = 3)
      integer DFACC_WRITE, DFNT_INT16, DFNT_CHAR8,
     +        MFGR_INTERLACE_PIXEL
      parameter (DFACC_WRITE  = 2,
     +           DFNT_CHAR8   = 4,
     +           DFNT_INT16   = 22,
     +           MFGR_INTERLACE_PIXEL = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgselct, mgcreat, mgwrimg, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer gr_id, ri1_id, ri2_id, data_type, interlace_mode
      integer start1(2), stride1(2), edges1(2)
      integer start2(2), stride2(2), edges2(2), dim_sizes(2)
      integer i, j, k
      integer*2  image1_buf(N1_COMPS, X1_LENGTH, Y1_LENGTH)
      character  image2_buf(N2_COMPS, X2_LENGTH, Y2_LENGTH)
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for writing.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Select the first raster image in the file.
C
      ri1_id = mgselct(gr_id, 0)
C
C     Fill the buffer with values.
C
      do 20 i = 1, Y1_LENGTH
         do 10 j = 1, X1_LENGTH
               image1_buf(1,j,i) = 0
               image1_buf(2,j,i) = 0
10       continue
20    continue
C
C     Define the part of the data in the first image that will be overwritten
C     with the new values from image1_buf.
C
      start1(1) = 0
      start1(2) = 0
      edges1(1) = X1_LENGTH
      edges1(2) = Y1_LENGTH
      stride1(1) = 1
      stride1(2) = 1
C
C     Write the data in the buffer into the image array.
C
      status = mgwrimg(ri1_id, start1, stride1, edges1, image1_buf)

C
C    Set the number type, interlace mode, and dimensions of the second image.
C
      data_type = DFNT_CHAR8
      interlace_mode = MFGR_INTERLACE_PIXEL
      dim_sizes(1) = X2_LENGTH
      dim_sizes(2) = Y2_LENGTH
C
C     Create the second image in the file.
C
      ri2_id = mgcreat(gr_id, IMAGE2_NAME, N2_COMPS, data_type,
     +                interlace_mode, dim_sizes)
C
C     Fill the image data buffer with values.
C
      do 60 i = 1, Y2_LENGTH
         do 50 j = 1, X2_LENGTH
            do 40 k = 1, N2_COMPS
               image2_buf(k,j,i) = char(65 + k - 1)
40          continue
50       continue
60    continue

C
C     Define the size of the data to be written, i.e., start from the origin
C     and go as long as the length of each dimension.
C
      start2(1) = 0
      start2(2) = 0
      edges2(1) =  dim_sizes(1)
      edges2(2) =  dim_sizes(2)
      stride2(1) = 1
      stride2(2) = 1
C
C     Write the data in the buffer into the image array.
C
      status = mgwrimg(ri2_id, start2, stride2, edges2, image2_buf)

C
C     Terminate access to the raster images and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri1_id)
      status = mgendac(ri2_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

8.6.2. Compressing Raster Images: GRsetcompress

Images can be compressed using the routine **GRsetcompress**. **GRsetcompress** compresses the image data at the time it is called and supports all standard HDF compression algorithms. The syntax of the routine **GRsetcompress** is as follows:

```
C:          status = GRsetcompress(ri_id, comp_type, c_info);

FORTRAN:    status = mgscompress(ri_id, comp_type, comp_prm)
```

The compression method is specified by the parameter *comp_type*. Valid values of the parameter *comp_type* are:

`COMP_CODE_NONE` (or `0`) for no compression
`COMP_CODE_RLE` (or `1`) for RLE run-length encoding
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for Szip compression (not for Fortran)
`COMP_CODE_JPEG` (or `7`) for JPEG compression

The compression parameters are specified by the parameter *c_info* in C and the parameter *comp_prm* in FORTRAN-77. The parameter *c_info* has type *comp_info* and contains algorithm-specific information for the library compression routines. The type *comp_info* is described in the header file `hcomp.h` and in the reference manual page for **SDsetcompress**. Compression parameters are only needed when Skipping Huffman, GZIP, and Szip compression methods are applied.

If *comp_type* is set to `COMP_CODE_NONE` or `COMP_CODE_RLE`, the parameters *c_info* and *comp_prm* are not used; *c_info* can be set to `NULL` and *comp_prm* can be undefined.

If *comp_type* is set to `COMP_CODE_SKPHUFF`, then the structure *skphuff* in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) must be provided with the size, in bytes, of the data elements.

If *comp_type* is set to `COMP_CODE_DEFLATE`, the deflate structure in the union *comp_info* in C (*comp_prm(1)* in FORTRAN-77) must be provided with the information about the compression effort.

Note that, as of HDF 4.2.2, Szip is not supported in Fortran GR interface yet.

GRsetcompress returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The **GRsetcompress** parameters are further described in Table 8D.

8.6.3. Setting I/O Access Type for a Raster Image: GRsetaccesstype

GRsetaccesstype sets the access type to be either serial or parallel I/O for the raster image specified by *ri_id*.

The syntax of the routine **GRsetaccesstype** is as follows:

```
C:          status = GRsetaccesstype(ri_id, access_type);

FORTRAN:    status = mgsactp(ri_id, access_type)
```

The access type is specified by the parameter *access_type* and its valid values are `DFACC_SERIAL` (or `1`), `DFACC_PARALLEL` (or `11`), and `DFACC_DEFAULT` (or `0`.)

GRsetaccesstype returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The **GRsetaccesstype** parameters are further described in Table 8D.

TABLE 8D **GRsetcompress and GRsetaccesstype Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRsetcompress [intn] (`mgscompress`)	ri_id	int32	integer	Raster image identifier
	comp_type	int32	integer	Compression method
	c_info	comp_info*	N/A	Pointer to compression information structure
	comp_prm	N/A	integer	Compression parameters array
GRsetaccesstype [intn] (`mgsactp`)	ri_id	int32	integer	Raster image identifier
	access_type	int32	integer	I/O access type

8.6.4. External File Operations Using the GR Interface

An ***external image array*** is one that is stored in a file that is not the file containing the metadata for the image. The HDF file containing the metadata is known as the primary HDF file; the file containing the external image array is known as an ***external file***. The concept of externally stored data is described in Chapter 3, *Scientific Data Sets (SD API)*. The GR interface supports the same external file functionality as the SD interface.

8.6.4.1. Creating a Raster Image in an External File: GRsetexternalfile

Creating an image with the data stored in an external file involves the same general steps as with the SD interface:

1. Create the image array.
2. Specify that an external data file is to be used.
3. Write data to the image array.
4. Terminate access to the image.

To create a data set containing image array stored in an external file, the calling program must make the following calls.

```
C:         ri_id = GRcreate(gr_id, name, n_comps, data_type, interlace_mode,
                            dim_sizes);
           status = GRsetexternalfile(ri_id, filename, offset);
           status = GRwriteimage(ri_id, start, stride, edges, image_data);
           status = GRendaccess(ri_id);

FORTRAN:   ri_id = mgcreat(gr_id, name, n_comps, data_type, interlace_mode,
                            dim_sizes)
           status = mgsxfil(ri_id, filename, offset)
           status = mgwrimg(ri_id, start, stride, edges, image_data)
           status = mgendac(ri_id)
```

GRsetexternalfile marks the image identified by the parameter *ri_id* as one whose data is to be written to an external file. The parameter *filename* is the name of the external file, and the parameter *offset* specifies the number of bytes from the beginning of the external file to the location where the first byte of data will be written.

GRsetexternalfile can only be called once per data set. If a file with the same name as *filename* exists in the current directory, HDF will use it as the external file. If the file does not exist, HDF will create one. Once the name of the external file is specified, it is impossible to change it without breaking the association between the raster image and its data.

Use caution when writing to existing files because the routine **GRwriteimage** begins its write at the specified offset without checking whether existing data is being overwritten. When different data sets have arrays being stored the same external file, the calling program is responsible for avoiding any overlap between them.

GRsetexternalfile returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **GRsetexternalfile** are further defined in Table 8E.

TABLE 8E **GRsetexternalfile Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
GRsetexternalfile [intn] **(mgsxfil)**	ri_id	int32	integer	Raster image identifier
	filename	char *	character*(*)	Name of the external file
	offset	int32	integer	Offset in bytes from the beginning of the external file to the image data

8.6.4.2. Moving Raster Images to an External File

Images can be moved from the primary HDF file to an external file. To do so requires the following steps:

1. Select the image.
2. Specify the external data file.
3. Terminate access to the image.

The calling program must make the following calls:

```
C:          ri_id = GRselect(gr_id, ri_index);
            status = GRsetexternalfile(ri_id, filename, offset);
            status = GRendaccess(ri_id);

FORTRAN:    ri_id = mgselct(gr_id, ri_index);
            status = mgsxfil(ri_id, filename, offset)
            status = mgendac(ri_id);
```

When **GRsetexternalfile** is used in conjunction with **GRselect**, it will immediately write the existing data to the external file; any data in the external file that occupies the space reserved for the external array will be overwritten as a result of this operation. A data set can only be moved to an external file once.

During the operation, the data is written to the external file as a contiguous stream regardless of how it is stored in the primary file. Because data is moved "as is," any unwritten locations in the data set are preserved in the external file. Subsequent read and write operations performed on the data set will access the external file.

8.7. Reading Raster Images

Image array data can be read as an entire array or as a subsample of the array. Raster image data is read from an external file in the same way that it is read from a primary file; whether the image array is stored in an external file is transparent to the user. This section describes how **GRreadim-**

age is used to read an entire image and part of an image. The section also describes the routine **GRreqimageil** that sets the interlacing for reading image data.

8.7.1. Reading Data from an Image: GRreadimage

Reading data subsamples from an image array involves the following steps:

1. Select a data set.
2. Read data from the image array.
3. Terminate access to the data set.

To read data from an image array, the calling program must contain the following function calls:

```
C:          ri_id = GRselect(gr_id, ri_index);
            status = GRreadimage(ri_id, start, stride, edges, data);
            status = GRendaccess(ri_id);

FORTRAN:    ri_id = mgselct(gr_id, ri_index)

            status = mgrdimg(ri_id, start, stride, edges, data)
    OR      status = mgrcimg(ri_id, start, stride, edges, data)

            status = mgendac(gr_id)
```

GRreadimage can be used to read either an entire image or a subsample of the image. The *ri_id* argument is the raster image identifier returned by **GRselect**. As with **GRwriteimage**, the arguments *start*, *stride*, and *edges* respectively describe the starting location for the read operation, the number of locations the current image array location will be moved forward after each read, and the length of each dimension to be read. Refer to Section "*Writing Raster Images: GRwriteimage"* for detailed descriptions of the parameters *start*, *stride*, and *edges*. If the image array is smaller than the *data* argument array, the amount of data read will be limited to the maximum size of the image array.

Note that the FORTRAN-77 version of **GRreadimage** has two routines; **mgrdimg** reads numeric image data and **mgrcimg** reads character image data.

GRreadimage returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **GRreadimage** are further defined in (See TABLE 8F).

8.7.2. Setting the Interlace Mode for an Image Read: GRreqimageil

The **GRreqimageil** routine sets the interlace mode for the next image read. The syntax of this routine is as follows:

```
C:          status = GRreqimageil(ri_id, interlace_mode);

FORTRAN:    status = mgrimil(ri_id, interlace_mode)
```

GRreqimageil can be called at any time before the read operation and takes two parameters, *ri_id* and *interlace_mode*. The parameter *ri_id* is the raster image identifier returned by the **GRselect** routine and the parameter *interlace_mode* specifies the interlace mode that will be in effect for the image read operation. Refer to Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate"* for a description of the GR interlace modes.

GRreqimagetil may be called more than once; the interlace mode setting specified by the last call to the routine will be used for the next read operation.

GRreqimagetil returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 8F.

TABLE 8F

GRreadimage and GRreqimageil Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRreadimage [intn] **(mgrdimg/ mgrcimg)**	ri_id	int32	integer	Raster image identifier
	start	int32[2]	integer (2)	Array containing the starting read coordinates
	stride	int32[2]	integer (2)	Array specifying the interval between the values that will be read along each dimension
	edges	int32[2]	integer (2)	Array containing the number of data elements that will be read along each dimension
	data	VOIDP	<valid numeric data type>(*)/character*(*)	Buffer for the image data to be read
GRreqimageil [intn] **(mgrimil)**	ri_id	int32	integer	Raster image identifier
	interlace_mode	intn	integer	Interlace mode for the next image read operation

EXAMPLE 3.

Reading a Raster Image.

This example illustrates the use of the routine **GRreadimage/mgrdimg** to read an image and its subsets.

In this example, the program reads the image written by Example 1 and modified by Example 2 in the file "General_RImages.hdf". Recall that this image has two components and has 5 rows and 10 columns. The program first reads the entire image, then reads a subset of the image, 3 rows and 2 columns starting at the 2nd row and the 4th column, and finally reads the image skipping all the even rows and all the odd columns. Reading patterns are applied to all components.

C:

```
#include "hdf.h"

#define  FILE_NAME       "General_RImages.hdf"
#define  N_COMPS         2
#define  X_LENGTH        10   /* number of columns of the entire image */
#define  Y_LENGTH        5    /* number of rows of the entire image */
#define  PART_COLS       2    /* number of columns read for partial image */
#define  PART_ROWS       3    /* number of rows read for partial image */
#define  SKIP_COLS       5    /* number of columns read for skipped image */
#define  SKIP_ROWS       3    /* number of rows read for skipped image */
#define  COLS_PART_START 3    /* starting column to read partial image */
#define  ROWS_PART_START 1    /* starting row to read partial image */
#define  COLS_SKIP_START 1    /* starting column to read skipped image */
#define  ROWS_SKIP_START 0    /* starting row to read skipped image */
#define  N_STRIDES       2    /* number of elements to skip on each dim. */

main( )
{
   /************************ Variable declaration *************************/

   intn  status;        /* status for functions returning an intn */
   int32 index;
   int32 file_id, gr_id, ri_id,
         start[2],      /* start position to write for each dimension */
```

```
      edges[2],      /* number of elements to bewritten along
                        each dimension */
      stride[2],     /* number of elements to skip on each dimension */
      dim_sizes[2];  /* dimension sizes of the image array */
int16 entire_image[Y_LENGTH][X_LENGTH][N_COMPS],
      partial_image[PART_ROWS][PART_COLS][N_COMPS],
      skipped_image[SKIP_ROWS][SKIP_COLS][N_COMPS];
int32 i, j;

/********************** End of variable declaration **********************/

/*
* Open the HDF file for reading.
*/
file_id = Hopen (FILE_NAME, DFACC_READ, 0);

/*
* Initialize the GR interface.
*/
gr_id = GRstart (file_id);

/*
* Select the first raster image in the file.
*/
ri_id = GRselect (gr_id, 0);

/*
* Define the size of the data to be read, i.e., start from the origin
* and go as long as the length of each dimension.
*/
start[0] = start[1] = 0;
edges[0] = X_LENGTH;
edges[1] = Y_LENGTH;

/*
* Read the data from the raster image array.
*/
status = GRreadimage (ri_id, start, NULL, edges, (VOIDP)entire_image);

/*
* Display only the first component of the image since the two components
* have the same data in this example.
*/
printf ("First component of the entire image:\n");
for (i = 0; i < Y_LENGTH; i++)
{
   for (j = 0; j < X_LENGTH; j++)
      printf ("%d ", entire_image[i][j][0]);
   printf ("\n");
}

/*
* Define the size of the data to be read.
*/
start[0] = COLS_PART_START;
start[1] = ROWS_PART_START;
edges[0] = PART_COLS;
edges[1] = PART_ROWS;

/*
* Read a subset of the raster image array.
*/
status = GRreadimage (ri_id, start, NULL, edges, (VOIDP)partial_image);
```

```
      /*
      * Display the first component of the read sample.
      */
      printf ("\nThree rows & two cols at 2nd row and 4th column");
      printf (" of the first component:\n");
      for (i = 0; i < PART_ROWS; i++)
      {
         for (j = 0; j < PART_COLS; j++)
            printf ("%d ", partial_image[i][j][0]);
         printf ("\n");
      }

      /*
      * Define the size and the pattern to read the data.
      */
      start[0] = COLS_SKIP_START;
      start[1] = ROWS_SKIP_START;
      edges[0] = SKIP_COLS;
      edges[1] = SKIP_ROWS;
      stride[0] = stride[1] = N_STRIDES;

      /*
      * Read all the odd rows and even columns of the image.
      */
      status = GRreadimage (ri_id, start, stride, edges, (VOIDP)skipped_image);

      /*
      * Display the first component of the read sample.
      */
      printf ("\nAll odd rows and even columns of the first component:\n");
      for (i = 0; i < SKIP_ROWS; i++)
      {
         for (j = 0; j < SKIP_COLS; j++)
            printf ("%d ", skipped_image[i][j][0]);
         printf ("\n");
      }

      /*
      * Terminate access to the raster image and to the GR interface, and
      * close the HDF file.
      */
      status = GRendaccess (ri_id);
      status = GRend (gr_id);
      status = Hclose (file_id);
   }
```

FORTRAN:

```
      program read_raster_image
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      integer      X_LENGTH
      integer      Y_LENGTH
      integer      N_COMPS
C
      parameter (FILE_NAME  = 'General_RImages.hdf',
     +           X_LENGTH   = 10,
     +           Y_LENGTH   = 5,
     +           N_COMPS    = 2)
```

```
      integer PART_COLS, PART_ROWS, SKIP_COLS, SKIP_ROWS
      integer COLS_PART_START, ROWS_PART_START
      integer COLS_SKIP_START, ROWS_SKIP_START
      integer N_STRIDES
      parameter (PART_COLS = 3, PART_ROWS = 2,
     +           SKIP_COLS = 3, SKIP_ROWS = 5,
     +           COLS_PART_START = 1, ROWS_PART_START = 3,
     +           COLS_SKIP_START = 0, ROWS_SKIP_START = 1,
     +           N_STRIDES = 2)
      integer DFACC_READ
      parameter (DFACC_READ = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgselct, mgrdimg, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id
      integer gr_id, ri_id
      integer start(2), stride(2), edges(2)
      integer i, j
      integer*2  entire_image(N_COMPS, X_LENGTH, Y_LENGTH)
      integer*2  partial_image(N_COMPS, PART_ROWS, PART_COLS)
      integer*2  skipped_image(N_COMPS, SKIP_ROWS, SKIP_COLS)
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Select the first raster image in the file.
C
      ri_id = mgselct(gr_id, 0)
C
C     Define the size of the data to be read, i.e., start from the origin
C     and go as long as the length of each dimension.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Read the data from the raster image array.
C
      status = mgrdimg(ri_id, start, stride, edges, entire_image)
C
C     Display only the first component of the image since the two components
C     have the same data in this example.
C
      write(*,*) 'First component of the entire image'
      write(*,*)
      do 10 i = 1, X_LENGTH
         write(*,1000) (entire_image(1,i,j), j = 1, Y_LENGTH)
10    continue
      write(*,*)
C
C     Define the size of the data to be read.
C
      start(1) = ROWS_PART_START
      start(2) = COLS_PART_START
      edges(1) = PART_ROWS
      edges(2) = PART_COLS
      stride(1) = 1
      stride(2) = 1
C
C     Read a subset of the raster image array.
C
      status = mgrdimg(ri_id, start, stride, edges, partial_image)
C
C     Display only the first component of the read sample.
C
      write(*,*)
     +  'Two rows and three columns at 4th row and 2nd column',
     +  ' of the first component'
      write(*,*)
      do 20 i = 1, PART_ROWS
         write(*,1000) (partial_image(1,i,j), j = 1, PART_COLS)
20    continue
      write(*,*)
C
C     Define the size and the pattern to read the data.
C
      start(1) = ROWS_SKIP_START
      start(2) = COLS_SKIP_START
      edges(1) = SKIP_ROWS
      edges(2) = SKIP_COLS
      stride(1) = N_STRIDES
      stride(2) = N_STRIDES
C
C     Read all the odd rows and even columns of the image.
C
      status = mgrdimg(ri_id, start, stride, edges, skipped_image)
C
C     Display only the first component of the read sample.
C
      write(*,*) 'All even rows and odd columns of the first component'
      write(*,*)
      do 30 i = 1, SKIP_ROWS
         write(*,1000) (skipped_image(1,i,j), j = 1, SKIP_COLS)
30    continue
      write(*,*)
C
C     Terminate access to the raster image and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
1000  format(1x, 5(I4))
      end
```

8.8. Difference between the SD and GR Interfaces

There is a difference between the SD and GR interfaces that becomes important in applications or tools that must manipulate both images and two-dimensional SDs.

The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and a GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- For **SDreaddata**, those parameters are expressed in (*y*,*x*) or [*row*,*column*] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For **GRreadimage**, those parameters are expressed in (*x*,*y*) or [*column*,*row*] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

8.9. Obtaining Information about Files and Raster Images

The routines covered in this section provide methods for obtaining information about all of the images in a file, for identifying images that meet certain criteria, and for obtaining information about specific raster images.

GRfileinfo retrieves the number of images and file attributes in a file. **GRgetiminfo** provides information about individual images. To retrieve information about all images in a file, a calling program can use **GRfileinfo** to determine the number of images, followed by repeated calls to **GRgetiminfo** to obtain information about each image.

GRnametoindex or **GRreftoindex** can be used to obtain the index of a raster image in a file knowing its name or reference number, respectively. Refer to Section "*Required GR Data Set Components"* for a description of the raster image index and reference number. **GRidtoref** is used when the reference number of an image is required by another routine and the raster image identifier is available.

These routines are described individually in the following subsections.

8.9.1. Obtaining Information about the Contents of a File: GRfileinfo

GRfileinfo retrieves the number of raster images and the number of file attributes contained in a file. This information is often useful in index validation, sequential searches, or memory allocation. The syntax of **GRfileinfo** is as follows:

```
C:          status = GRfileinfo(gr_id, &n_images, &n_file_attrs);

FORTRAN:    status = mgfinfo(gr_id, n_images, n_file_attrs)
```

The number of images in the file and the total number of file attributes will be stored in the arguments *n_images* and *n_file_attrs*, respectively.

GRfileinfo returns `SUCCEED` (or `0`) if successful or `FAIL` (or `-1`) otherwise. The parameters for **GRfileinfo** are further described in Table 8G.

8.9.2. Obtaining Information about an Image: GRgetiminfo

It is impossible to allocate the proper amount of memory to buffer the image data when the number of components, dimension sizes, and/or data type of the image are unknown. The routine **GRgetiminfo** retrieves this required information. To access information about an image, the calling program must contain the following:

```
C:          status = GRgetiminfo(ri_id, name, &n_comps, &data_type, &interlace_-
                           mode, dim_sizes, &n_attrs);

FORTRAN:    status = mggiinf(ri_id, name, n_comps, data_type, interlace_mode,
                           dim_sizes, n_attrs)
```

GRgetiminfo takes a raster image identifier as input, and returns the name, number of components, data type, interlace mode, dimension size, and number of attributes for the corresponding image in the arguments *name*, *n_comps*, *data_type*, *interlace_mode*, *dim_sizes*, and *n_attrs* respectively. The number of components of an image array element corresponds to the order of a vdata field, therefore this implementation of image components in the GR interface is flexible enough to accommodate any representation of pixel data. The calling program determines this representation; the GR interface recognizes only the raw byte configuration of the data. The attribute count will only reflect the number of attributes assigned to the image array; file attributes are not included.

GRgetiminfo returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 8G.

TABLE 8G **GRfileinfo and GRgetiminfo Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
GRfileinfo [intn] **(mgfinfo)**	gr_id	int32	integer	GR interface identifier
	n_images	int32 *	integer	Number of raster images in the file
	n_file_attrs	int32 *	integer	Number of global attributes in the file
GRgetiminfo [intn] **(mggiinf)**	ri_id	int32	integer	Raster image identifier
	name	char *	character*(*)	Name of the raster image
	n_comps	int32 *	integer	Number of pixel components in the pixel
	data_type	int32 *	integer	Pixel data type
	interlace_mode	int32 *	integer	Interlace mode of the data in the raster image
	dim_sizes	int32 [2]	integer (2)	Array containing the size of each dimension in the raster image
	n_attrs	int32 *	integer	Number of raster image attributes

8.9.3. Obtaining the Reference Number of a Raster Image from Its Identifier: GRidtoref

GRidtoref returns either the reference number of the raster image identified by the parameter *ri_id*, or `FAIL` (or `-1`) upon unsuccessful completion. The syntax of **GRidtoref** is as follows:

```
C:          ref = GRidtoref(ri_id);

FORTRAN:    ref = mgid2rf(ri_id)
```

This routine is further defined in Table 8H.

8.9.4. Obtaining the Index of a Raster Image from Its Reference Number: GRreftoindex

GRreftoindex returns either the index of the raster image specified by its reference number, *ref*, or `FAIL` (or `-1`) upon unsuccessful completion. The syntax of **GRreftoindex** is as follows:

```
C:          ri_index = GRreftoindex(gr_id, ref);

FORTRAN:    ri_index = mgr2idx(gr_id, ref)
```

This routine is further defined in Table 8H.

8.9.5. Obtaining the Index of a Raster Image from Its Name: GRnametoindex

GRnametoindex returns the index of the raster image specified by its name or `FAIL` (or `-1`) upon unsuccessful completion. The syntax of **GRnametoindex** is as follows:

```
C:          ri_index = GRnametoindex(gr_id, name);

FORTRAN:    ri_index = mgr2idx(gr_id, name)
```

This routine is further defined in Table 8H.

8.9.6. Obtaining Compression Information for a Raster Image: GRgetcompinfo

GRgetcompinfo retrieves the type of compression used to store a raster image and, when appropriate, the required compression parameters. **GRgetcompinfo** replaces **GRgetcompress** because this function has flaws, causing failure for some chunked and chunked/compressed data.

GRgetcompinfo takes one input parameter, *ri_id*, a raster image identifier, and two output parameters, *comp_type*, for the type of compression used when the image was written, and either *c_info* (a C struct) or *comp_prm* (a FORTRAN-77 array) for the returned compression parameters.

Valid *comp_type* values are as follows:

`COMP_CODE_NONE` (or `0`) for no compression
`COMP_CODE_RLE` (or `1`) for RLE run-length encoding
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for Szip compression (not for Fortran)
`COMP_CODE_JPEG` (or `7`) for JPEG compression

The *c_info* struct is of type `comp_info`, contains algorithm-specific information for the library compression routines, and is described in the `hcomp.h` header file.

The *comp_prm* parameter is an array of several elements.

For Skipping Huffman compression, *comp_prm(1)* contains the skip value, `skphuff_skp_size`.

For GZIP compression, *comp_prm(1)* contains the deflation value, `deflate_value`.

For other compression types, *comp_prm* is ignored. Currently, Szip is not yet supported in Fortran GR interface.

GRgetcompinfo returns `SUCCESS` (or `0`) if it is successful or `FAIL` (or `-1`) upon unsuccessful completion.

The syntax of **GRgetcompinfo** is as follows:

```
C:          status = GRgetcompinfo(ri_id, comp_type, c_info);

FORTRAN:    status = mggcompress(ri_id, comp_type, comp_prm)
```

This routine is further defined in Table 8H.

8.9.7. Checking Whether a Raster Image Is To Be Mapped: GR2bmapped

This function was originally added to support the HDF4 File Content Project. The tool, produced from the project, maps the contents of HDF4 files. Supporting for raster images was limited as requested by the project's sponsor. Thus, only certain types of images, which satisfy a set of conditions, are to be mapped.

GR2bmapped will set *tobe_mapped* to `TRUE` if the given raster image, *ri_id*, satisfies the following conditions:

- being an 8-bit raster image,
- having one component,
- being non-special or RLE compressed only, i.e., no other compressions or chunking,

or `FAIL` (or `-1`), otherwise. The syntax of **GR2bmapped** is as follows:

```
C:          status = GR2bmapped(ri_id, &tobe_mapped, &name_generated);

FORTRAN:    Unavailable
```

Another characteristic of the image to be reported by **GR2bmapped** is whether the image has name that was generated by the library and, if so, *name_generated* will be set to `TRUE`. Old images (or images created with pre-GR API) do not have a name and the library would generate a name for it while reading in the file. The tool from the HDF4 File Content Project needs to make this distinction.

GR2bmapped returns `SUCCEED` (or `0`), if successful, or `FAIL` (or `-1`), otherwise. When failure occurs, *tobe_mapped* and *name_generated* will be undefined. This routine is further defined in Table 8H.

TABLE 8H

GRidtoref, GRreftoindex, GRnametoindex, and GRgetcompinfo Parameter Lists

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRidtoref [uint16] **(mgid2rf)**	ri_id	int32	integer	Raster image identifier
GRreftoindex [int32] **(mgr2idx)**	gr_id	int32	integer	GR interface identifier
	ref	uint16	integer	Reference number of the raster image
GRnametoindex [int32] **(mgn2ndx)**	gr_id	int32	integer	GR interface identifier
	name	char *	character *(*)	Name of the raster image
GRgetcompinfo [intn] **(mggcompress)**	ri_id	int32	integer	Raster image identifier
	comp_type	comp_coder_t	integer	Type of compression
	c_info	comp_info	N/A	Pointer to compression information structure
	comp_prm(1)	N/A	integer	Compression parameter in array format
GR2bmapped [intn] **(unavailable)**	ri_id	int32	integer	Raster image identifier
	tobe_mapped	intn *	integer	TRUE if the image should be mapped
	name_generated	intn *	N/A	TRUE if the image's name was generated by the GR API, i.e., not given by applications

EXAMPLE 4.

Obtaining File and Image Information.

This example illustrates the use of the routines **GRfileinfo/mgfinfo** and **GRgetiminfo/mggiinf** to obtain information such as the number of images and attributes in an HDF file and the characteristics of a raster image in the file.

In this example, the program gets the number of images in the file using the routine **GRfileinfo/mgfinfo**. For each image, the program then obtains and displays its name, number of components, data type, interlace mode, dimension sizes, and number of attributes using the routine **GRgetiminfo/mggiinf**.

C:

```
#include "hdf.h"

#define  FILE_NAME    "General_RImages.hdf"

main( )
{
   /************************* Variable declaration **************************/

   intn  status;            /* status for functions returning an intn */
   int32 file_id, gr_id, ri_id,
         n_rimages,         /* number of raster images in the file */
         n_file_attrs,      /* number of file attributes */
         ri_index,          /* index of a image */
         dim_sizes[2],      /* dimensions of an image */
         n_comps,           /* number of components an image contains */
         interlace_mode,    /* interlace mode of an image */
         data_type,         /* number type of an image */
         n_attrs;           /* number of attributes belong to an image */
   char  name[MAX_GR_NAME], /* name of an image */
        *type_string,       /* mapped text of a number type */
```

```
     *interlace_string;  /* mapped text of an interlace mode */

/********************** End of variable declaration **********************/

/*
* Open the file for reading.
*/
file_id = Hopen (FILE_NAME, DFACC_READ, 0);

/*
* Initialize the GR interface.
*/
gr_id = GRstart (file_id);

/*
* Determine the contents of the file.
*/
status = GRfileinfo (gr_id, &n_rimages, &n_file_attrs);

/*
* For each image in the file, get and display the image information.
*/
printf ("RI#    Name       Components  Type         Interlace     \
Dimensions   Attributes\n\n");
for (ri_index = 0; ri_index < n_rimages; ri_index++)
{
   ri_id = GRselect (gr_id, ri_index);
   status = GRgetiminfo (ri_id, name, &n_comps, &data_type,
                     &interlace_mode, dim_sizes, &n_attrs);
   /*
   * Map the number type and interlace mode into text strings for output
   * readability.  Note that, in this example, only two possible types
   * are considered because of the simplicity of the example.  For real
   * problems, all possible types should be checked and, if reading the
   * data is desired, the size of the type must be determined based on the
   * machine where the program resides.
   */
   if (data_type == DFNT_CHAR8)
      type_string = "Char8";
   else if (data_type == DFNT_INT16)
      type_string = "Int16";
   else
      type_string = "Unknown";

   switch (interlace_mode)
   {
      case MFGR_INTERLACE_PIXEL:
         interlace_string = "MFGR_INTERLACE_PIXEL";
         break;
      case MFGR_INTERLACE_LINE:
         interlace_string = "MFGR_INTERLACE_LINE";
         break;
      case MFGR_INTERLACE_COMPONENT:
         interlace_string = "MFGR_INTERLACE_COMPONENT";
         break;
      default:
         interlace_string = "Unknown";
         break;
   } /* switch */

   /*
   * Display the image information for the current raster image.
   */
```

```
              printf ("%d  %s       %d      %s   %s     %2d,%2d         %d\n",
                     ri_index, name, n_comps, type_string, interlace_string,
                     dim_sizes[0], dim_sizes[1], n_attrs);

          /*
          * Terminate access to the current raster image.
          */
          status = GRendaccess (ri_id);
       }

       /*
       * Terminate access to the GR interface and close the HDF file.
       */
       status = GRend (gr_id);
       status = Hclose (file_id);
    }
```

FORTRAN:

```
      program image_info
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
C
      parameter (FILE_NAME = 'General_RImages.hdf')
      integer DFACC_READ
      parameter (DFACC_READ = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgselct, mgfinfo, mggiinf, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, gr_id, ri_id
      integer n_rimages, n_file_attrs, ri_index
      integer n_comps, interlace_mode, n_attrs, data_type
      integer dim_sizes(2)
      character*10 type_string
      character*24 interlace_string
      character*64 name
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Determine the contents of the file.
C
      status = mgfinfo(gr_id, n_rimages, n_file_attrs)
C
C     For each image in the file, get and display image information.
```

```
C
      do 100 ri_index = 0, n_rimages-1
         ri_id = mgselct(gr_id, ri_index)
         status = mggiinf(ri_id, name, n_comps, data_type,
     +                    interlace_mode, dim_sizes, n_attrs)
C
C     Map the number type and interlace mode into text strings for
C     output readability.
C
      if(data_type .eq. 4) then
         type_string = 'DFNT_CHAR8'
      else if(data_type .eq. 22) then
         type_string = 'DFNT_INT16'
      else
         type_string = 'Unknown'
      endif
      if (interlace_mode .eq. 0) then
          interlace_string = 'MFGR_INTERLACE_PIXEL'
      else if(interlace_mode .eq. 1) then
          interlace_string = 'MFGR_INTERLACE_LINE'
      else if(interlace_mode .eq. 2) then
          interlace_string = 'MFGR_INTERLACE_COMPONENT'
      else
         interlace_string = 'Unknown'
      endif
C
C     Display the image information for the current image.
C
      write(*,*) 'Image index: ', ri_index
      write(*,*) 'Image name: ', name
      write(*,*) 'Number of components: ', n_comps
      write(*,*) 'Number type: ', type_string
      write(*,*) 'Interlace mode: ', interlace_string
      write(*,*) 'Dimnesions: ', dim_sizes(1), dim_sizes(2)
      write(*,*) 'Number of image attributes: ', n_attrs
      write(*,*)
C
C     Terminate access to the current raster image.
C
      status = mgendac(ri_id)
100   continue
C
C     Terminate access to the GR interface and close the HDF file.
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

8.10. GR Data Set Attributes

The GR interface provides tools that attach attributes to particular images. This capability is similar to, though more limited than, attribute function capabilities of the SD interface. The concepts of user-defined and predefined attributes are explained in Chapter 3, *Scientific Data Sets (SD API)*. The GR implementation of attributes is similar to the SD implementation. Attributes are not written out to a file until access to the object the attribute is attached to is terminated.

8.10.1. Predefined GR Attributes

The GR API library has only one predefined attribute: `FILL_ATTR`. This attribute defines a fill pixel, which is analogous to a fill value in the SD interface. It represents the default value that is written to each element of an image array not explicitly written to by the calling program, i.e., when only a portion of the entire image array is filled with data. This value must of the same data type as the rest of the initialized image data. The routine used to set the fill value, **GRsetattr**, is explained in the next section.

8.10.2. Setting User-defined Attributes: GRsetattr

GRsetattr creates or modifies an attribute for either a file or a raster image. If the attribute with the specified name does not exist, **GRsetattr** creates a new one. If the named attribute already exists, **GRsetattr** resets all the values that are different from those provided in its argument list. The syntax of **GRsetattr** is as follows:

```
C:          status = GRsetattr(obj_id, attr_name, data_type, n_values,
                               attr_value);

FORTRAN:   status = mgsnatt(obj_id, attr_name, data_type, n_values, attr_value)
    OR     status = mgscatt(obj_id, attr_name, data_type, n_values, attr_value)
```

The first argument, *obj_id*, can either be the GR interface identifier or raster image identifier. The argument *attr_name* contains the name of the attribute and can be no more than `H4_MAX_GR_NAME` (or `256`) characters in length. Passing the name of an existing attribute will overwrite the value portion of that attribute.

The arguments *data_type*, *n_values*, and *attr_value* describe the right side of the *label=value* equation. The *attr_value* argument contains one or more values of the same data type. The *data_-type* argument describes the data type for all values in the attribute and *n_values* contains the total number of values in the attribute.

Note that the FORTRAN-77 version of **GRsetattr** has two routines; **mgsnatt** writes numeric attribute data and **mgscatt** writes character attribute data.

GRsetattr returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters for **GRsetattr** are further described in Table 8I.

EXAMPLE 5. **Operations on File and Raster Image Attributes.**

This example illustrates the use of the routines **GRsetattr/mgsnatt/mgscatt** to assign attributes to an HDF file and to an image.

In this example, the program sets two attributes to the existing file "General_RImages.hdf" and two attributes to the image named "Image Array 2". The file is created by the program in Example 1 and the image is created by the program in Example 2. The values of the second attribute of the image are of type *int16* and the values of the other three attributes are of type *char8*.

C:

```
#include "hdf.h"

#define  FILE_NAME          "General_RImages.hdf"
#define  IMAGE_NAME         "Image Array 2"
#define  F_ATT1_NAME        "File Attribute 1"
#define  F_ATT2_NAME        "File Attribute 2"
#define  RI_ATT1_NAME       "Image Attribute 1"
#define  RI_ATT2_NAME       "Image Attribute 2"
```

```
#define  F_ATT1_VAL         "Contents of First FILE Attribute"
#define  F_ATT2_VAL         "Contents of Second FILE Attribute"
#define  F_ATT1_N_VALUES    32
#define  F_ATT2_N_VALUES    33
#define  RI_ATT1_VAL        "Contents of IMAGE's First Attribute"
#define  RI_ATT1_N_VALUES   35
#define  RI_ATT2_N_VALUES   6

main( )
{
   /************************* Variable declaration *************************/

   intn  status;         /* status for functions returning an intn */
   int32 gr_id, ri_id, file_id,
         ri_index;
   int16 ri_attr_2[RI_ATT2_N_VALUES] = {1, 2, 3, 4, 5, 6};

   /********************* End of variable declaration *********************/

   /*
   * Open the HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_WRITE, 0);

   /*
   * Initialize the GR interface.
   */
   gr_id = GRstart (file_id);

   /*
   * Set two file attributes to the file with names, data types, numbers of
   * values, and values of the attributes specified.
   */
   status = GRsetattr (gr_id, F_ATT1_NAME, DFNT_CHAR8, F_ATT1_N_VALUES,
                    (VOIDP)F_ATT1_VAL);

   status = GRsetattr (gr_id, F_ATT2_NAME, DFNT_CHAR8, F_ATT2_N_VALUES,
                    (VOIDP)F_ATT2_VAL);

   /*
   * Obtain the index of the image named IMAGE_NAME.
   */
   ri_index = GRnametoindex (gr_id, IMAGE_NAME);

   /*
   * Obtain the identifier of this image.
   */
   ri_id = GRselect (gr_id, ri_index);

   /*
   * Set two attributes to the image with names, data types, numbers of
   * values, and values of the attributes specified.
   */
   status = GRsetattr (ri_id, RI_ATT1_NAME, DFNT_CHAR8, RI_ATT1_N_VALUES,
                    (VOIDP)RI_ATT1_VAL);

   status = GRsetattr (ri_id, RI_ATT2_NAME, DFNT_INT16, RI_ATT2_N_VALUES,
                    (VOIDP)ri_attr_2);

   /*
   * Terminate access to the image and to the GR interface, and close the
   * HDF file.
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

FORTRAN:

```
      program  set_attribute
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*13 IMAGE_NAME
      character*16 F_ATT1_NAME
      character*16 F_ATT2_NAME
      character*17 RI_ATT1_NAME
      character*17 RI_ATT2_NAME
      character*32 F_ATT1_VAL
      character*33 F_ATT2_VAL
      integer      F_ATT1_N_VALUES
      integer      F_ATT2_N_VALUES
      character*35 RI_ATT1_VAL
      integer      RI_ATT1_N_VALUES
      integer      RI_ATT2_N_VALUES
C
      parameter (FILE_NAME    = 'General_RImages.hdf',
     +           IMAGE_NAME   = 'Image Array 2',
     +           F_ATT1_NAME  = 'File Attribute 1',
     +           F_ATT2_NAME  = 'File Attribute 2',
     +           RI_ATT1_NAME = 'Image Attribute 1',
     +           RI_ATT2_NAME = 'Image Attribute 2',
     +           F_ATT1_VAL   = 'Contents of First FILE Attribute',
     +           F_ATT2_VAL   = 'Contents of Second FILE Attribute',
     +           F_ATT1_N_VALUES = 32,
     +           F_ATT2_N_VALUES = 33,
     +           RI_ATT1_VAL = 'Contents of IMAGE''s First Attribute',
     +           RI_ATT1_N_VALUES = 35,
     +           RI_ATT2_N_VALUES = 6)
      integer DFACC_WRITE, DFNT_INT16, DFNT_CHAR8
      parameter (DFACC_WRITE  = 2,
     +           DFNT_CHAR8   = 4,
     +           DFNT_INT16   = 22)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgscatt, mgsnatt , mgn2ndx,
     +        mgselct, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer   status
      integer   file_id, gr_id, ri_id, ri_index
      integer*2 ri_attr_2(RI_ATT2_N_VALUES)
      integer   i

      do 10 i = 1, RI_ATT2_N_VALUES
         ri_attr_2(i) = i
10    continue
C
C**** End of variable declaration ************************************
```

```
C
C
C     Open the HDF file.
C
      file_id = hopen(FILE_NAME, DFACC_WRITE, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Set two file attributes to the file with names, data type, numbers of
C     values, and values of attributes specified.
C
      status = mgscatt(gr_id, F_ATT1_NAME, DFNT_CHAR8,
     +                 F_ATT1_N_VALUES, F_ATT1_VAL)
      status = mgscatt(gr_id, F_ATT2_NAME, DFNT_CHAR8,
     +                 F_ATT2_N_VALUES, F_ATT2_VAL)
C
C     Obtain the index of the image named IMAGE_NAMR.
C
      ri_index = mgn2ndx(gr_id, IMAGE_NAME)
C
C     Obtain the identifier of this image.
C
      ri_id = mgselct(gr_id, ri_index)
C
C     Set two attributes of the image with names, data types, number of
C     values, and values of the attributes specified.
C
      status = mgscatt(ri_id, RI_ATT1_NAME, DFNT_CHAR8,
     +                 RI_ATT1_N_VALUES, RI_ATT1_VAL)
      status = mgsnatt(ri_id, RI_ATT2_NAME, DFNT_INT16,
     +                 RI_ATT2_N_VALUES, ri_attr_2)
C
C     Terminate access to the image and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

8.10.3. Querying User-Defined Attributes: GRfindattr and GRattrinfo

Each attribute associated with an object has a unique ***attribute index***, a value ranging from 0 to the total number of attributes attached to the object - 1. Given a GR interface or raster image identifier and an attribute name, **GRfindattr** will return a valid attribute index of the file or raster image attribute if the attribute exists. The attribute index can then be used to retrieve information about the attribute or its values. Given a GR interface or raster image identifier and a valid attribute index, **GRattrinfo** returns the name, data type, and number of values for the file or raster image attribute if the attribute exists.

The syntax for **GRfindattr** and **GRattrinfo** is as follows:

```
C:          attr_index = GRfindattr(obj_id, attr_name);
            status = GRattrinfo(obj_id, attr_index, attr_name, &data_type,
                                &n_values);

FORTRAN:    attr_index = mgfndat(obj_id, attr_name)
            status = mgatinf(obj_id, attr_index, attr_name, data_type, n_values)
```

The parameter *obj_id* is either a GR interface identifier or a raster image identifier. The parameter *attr_name* specifies the name of the attribute. The parameter *attr_index* specifies the index of the attribute to be read. The attribute index is a zero-based integer and must be less than the total number of attributes assigned to the specified object. The parameter *data_type* specifies the data type of the attribute. And the parameter *n_values* specifies the number of attribute values.

GRfindattr returns the attribute index if successful and *FAIL* (or *-1*) otherwise. **GRattrinfo** returns *SUCCEED* (or *0*) if successful and *FAIL* (or *-1*) otherwise. The parameters for **GRfindattr** and **GRattrinfo** are further described in Table 8I.

8.10.4. Reading User-defined Attributes: GRgetattr

GRgetattr reads the values of an attribute assigned to the object identified by the parameter *obj_id*. The syntax for **GRgetattr** is as follows:

```
C:          status = GRgetattr(obj_id, attr_index, values);

FORTRAN:    status = mggnatt(obj_id, attr_index, values)
    OR      status = mggcatt(obj_id, attr_index, values)
```

The parameter *obj_id* is either a GR interface identifier or a raster image identifier. The parameter *attr_index* specifies the index of the attribute to be read. The attribute index is a zero-based integer and must be less than the total number of attributes assigned to the specified object.

It is assumed that the buffer *values*, allocated to hold the attribute values, is large enough to hold the data; if not, the data read will be truncated to the size of the buffer. The size of the buffer should be at least *n_values*sizeof(data_type)* bytes long. If an attribute contains multiple values, **GRgetattr** will return all of them. It is not possible to read a subset of values.

Note that the FORTRAN-77 version of **GRgetattr** has two routines; **mggnatt** reads numeric attribute data and **mggcatt** reads character attribute data.

GRgetattr returns *SUCCEED* (or *0*) if successful and *FAIL* (or *-1*) otherwise. The parameters for **GRgetattr** are further described in Table 8I.

TABLE 8I

GRsetattr, GRfindattr, GRattrinfo, and GRgetattr Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
GRsetattr [intn] **(mgsnatt/mgscatt)**	obj_id	int32	integer	GR interface or raster image identifier
	attr_name	char *	character*(*)	Name assigned to the attribute
	data_type	int32	integer	Data type of the attribute
	n_values	int32	integer	Number of values in the attribute
	values	VOIDP	<valid numeric data type>(*)/character*(*)	Buffer with the attribute values
GRfindattr [int32] **(mgfndat)**	obj_id	int32	integer	GR interface or raster image identifier
	attr_name	char *	character*(*)	Name of the attribute
GRattrinfo [intn] **(mgatinf)**	obj_id	int32	integer	GR interface or raster image identifier
	attr_index	int32	integer	Index for the attribute to be read
	attr_name	char *	character*(*)	Name of the attribute
	data_type	int32 *	integer	Data type of the attribute values
	n_values	int32 *	integer	Total number of values in the attribute
GRgetattr **[intn]** **(mggnatt/ mggcatt)**	obj_id	int32	integer	GR interface or raster image identifier
	attr_index	int32	integer	Index for the attribute to be read
	values	VOIDP	<valid numeric data type>(*)/character*(*)	Buffer for the attribute values

EXAMPLE 6.

Obtaining File and Image Attributes.

This example illustrates the use of the routines **GRattrinfo/mgatinf**, **GRfindattr/mgfndat**, and **GRgetattr/mggnatt/mggcatt** to extract information and values of file and image attributes that were set by the program in Example 5.

In this example, the program gets the information about each file attribute, then extracts its values. The program then selects the second image in the file, finds the attribute named "Image Attribute 2", obtains the data type and the number of values in the attribute, and extracts its stored values.

C:

```
#include "hdf.h"

#define  FILE_NAME       "General_RImages.hdf"
#define  RI_ATTR_NAME    "Image Attribute 2"

main( )
{
   /************************ Variable declaration *************************/

   intn   status;          /* status for functions returning an intn */
   int32  gr_id, ri_id, file_id,
          f_att_index,     /* index of file attributes */
          ri_att_index,    /* index of raster image attributes */
          data_type,       /* image data type */
          n_values,        /* number of values in an attribute */
          value_index,     /* index of values in an attribute */
          n_rimages,       /* number of raster images in the file */
          n_file_attrs;    /* number of file attributes */
   char   attr_name[MAX_GR_NAME];  /* buffer to hold the attribute name     */
```

```
VOIDP  data_buf;                /* buffer to hold the attribute values   */
int16 *int_ptr;      /* int16 pointer to point to a void data buffer     */
char8 *char_ptr;     /* char8 pointer to point to a void data buffer     */

/********************* End of variable declaration *********************/

/*
* Open the HDF file.
*/
file_id = Hopen (FILE_NAME, DFACC_READ, 0);

/*
* Initialize the GR interface.
*/
gr_id = GRstart (file_id);

/*
* Determine the number of attributes in the file.
*/
status = GRfileinfo (gr_id, &n_rimages, &n_file_attrs);

if (status != FAIL && n_file_attrs > 0)
{
   for (f_att_index = 0; f_att_index < n_file_attrs; f_att_index++)
   {
      /*
      * Get information about the current file attribute.
      */
      status = GRattrinfo (gr_id, f_att_index, attr_name, &data_type,
                           &n_values);

      /*
      * Allocate a buffer to hold the file attribute data.  In this example,
      * knowledge about the data type is assumed to be available from
      * the previous example for simplicity.  In reality, the size
      * of the type must be determined based on the machine where the
      * program resides.
      */
      if (data_type == DFNT_CHAR8)
      {
         data_buf = malloc (n_values * sizeof (char8));
         if (data_buf == NULL)
         {
            printf ("Unable to allocate space for attribute data.\n");
            exit (1);
         }
      }
      else
      {
        printf ("Unable to determine data type to allocate data buffer.\n");
         exit (1);
      }

      /*
      * Read and display the attribute values.
      */
      status = GRgetattr (gr_id, f_att_index, (VOIDP)data_buf);
      char_ptr = (char8 *) data_buf;
      printf ("Attribute %s: ", attr_name);
      for (value_index = 0; value_index < n_values; value_index++)
         printf ("%c", char_ptr[value_index]);
      printf ("\n");
```

```
            /*
            * Free the space allocated for the data buffer.
            */
            free (data_buf);
         } /* for */
      } /* if */

      /*
      * Select the second image in the file.
      */
      ri_id = GRselect (gr_id, 1);

      /*
      * Find the image attribute named RI_ATTR_NAME.
      */
      ri_att_index = GRfindattr (ri_id, RI_ATTR_NAME);

      /*
      * Get information about the attribute.
      */
      status = GRattrinfo (ri_id, ri_att_index, attr_name, &data_type, &n_values);

      /*
      * Allocate a buffer to hold the file attribute data.  As mentioned above,
      * knowledge about the data type is assumed to be available from
      * the previous example for simplicity.  In reality, the size of the
      * type must be determined based on the machine where the program resides.
      */
      if (data_type == DFNT_INT16)
         data_buf = malloc (n_values * sizeof (int16));

      /*
      * Read and display the attribute values.
      */
      status = GRgetattr (ri_id, ri_att_index, (VOIDP)data_buf);
      printf ("\nAttribute %s: ", RI_ATTR_NAME);
      int_ptr = (int16 *)data_buf;
      for (value_index = 0; value_index < n_values; value_index++)
         printf ("%d ", int_ptr[value_index]);
      printf ("\n");

      /*
      * Free the space allocated for the data buffer.
      */
      free (data_buf);

      /*
      * Terminate access to the raster image and to the GR interface, and
      * close the file.
      */
      status = GRendaccess (ri_id);
      status = GRend (gr_id);
      status = Hclose (file_id);
   }
```

FORTRAN:

```
      program  get_attribute
      implicit none
C
C     Parameter declaration
C
      character*19 FILE_NAME
      character*17 RI_ATTR_NAME
C
      parameter (FILE_NAME    = 'General_RImages.hdf',
     +           RI_ATTR_NAME  = 'Image Attribute 2')
      integer DFACC_READ, DFNT_INT16, DFNT_CHAR8
      parameter (DFACC_READ   = 1,
     +           DFNT_CHAR8   = 4,
     +           DFNT_INT16   = 22)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgfinfo, mgatinf, mggcatt, mggnatt , mgfndat,
     +        mgselct, mgendac, mgend

C
C**** Variable declaration *******************************************
C
      integer      status
      integer      file_id, gr_id, ri_id
      integer      f_att_index, ri_att_index, data_type, n_values
      integer      n_rimages, n_file_attrs
      integer*2    int_buf(10)
      character*17 attr_name
      character*80 char_buf
      integer      i
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Determine the number of attributes in the file.
C
      status = mgfinfo(gr_id, n_rimages, n_file_attrs)
      if ((status .NE. -1) .AND. (n_file_attrs .GT. 0)) then

         do 10 f_att_index = 0, n_file_attrs-1
C
C        Get information about the current file attribute.
C
         status = mgatinf(gr_id, f_att_index, attr_name, data_type,
     +                    n_values)
C
C       Check whether data type is DFNT_CHAR8 in order to use allocated buffer.
C
         if(data_type .NE. DFNT_CHAR8) then
            write(*,*)
     +      'Unable to determine data type to use allocated buffer'
         else
C
C           Read and display the attribute values.
C
            status = mggcatt(gr_id, f_att_index, char_buf)
            write(*,*) 'Attribute ', attr_name, ' : ',
     +                 char_buf(1:n_values)
         endif
```

```
10        continue

      endif

C
C     Select the second image in the file.
C
      ri_id = mgselct(gr_id, 1)
C
C     Find the image attribute named RI_ATTR_NAME.
C
      ri_att_index = mgfndat(ri_id, RI_ATTR_NAME)
C
C     Get information about the attribute.
C
      status = mgatinf(ri_id, ri_att_index, attr_name, data_type,
     +                 n_values)
C
C     Read and display attribute values.
C
      status = mggnatt(ri_id, ri_att_index, int_buf)
      write(*,*) 'Attributes :', (int_buf(i), i = 1, n_values)
C
C     Terminate access to the image and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

8.11. Reading and Writing Palette Data Using the GR Interface

The GR API library includes routines that read, write, and access information about palette data attached to GR images. Although this functionality is also provided by the HDF Palette API library, it is not a recommended practice to use the Palette API to access and manipulate palette objects created by GR interface routines.

The routines are named **GRgetlutid**, **GRluttoref**, **GRgetlutinfo**, **GRwritelut**, **GRreqlutil**, and **GRreadlut**. Note that the routine names use the term ***LUT*** to refer to palettes; LUT stands for color ***lookup tables***.

8.11.1. Obtaining a Palette Identifier: GRgetlutid

Given a palette index, the routine **GRgetlutid** is used to get the palette identifier for the specified palette.

The **GRgetlutid** function takes two arguments, *ri_id*, the raster image identifier of the image that has the palette attached to it, and *lut_index*, the index of the palette, and returns the value of the palette identifier corresponding to the specified image. The syntax of **GRgetlutid** is as follows:

```
C:          pal_id = GRgetlutid(ri_id, lut_index);

FORTRAN:    pal_id = mggltid(ri_id, lut_index)
```

GRgetlutid returns the value of the palette identifier if successful and `FAIL` (or `-1`) otherwise. The **GRgetlutid** parameters are further discussed in Table 8J.

8.11.2. Obtaining the Number of Palettes Associated with an Image: GRgetnluts

Given an image identifier, **GRgetnluts** is used to determne the number of palettes currently associated with an image.

The **GRgetnluts** function takes one argument, *ri_id*, a raster image identifier, and returns the number of palettes associated with that imare. The syntax of GRgetnluts is as follows:

```
C:          n_luts = GRgetnluts(ri_id);

FORTRAN:    n_luts = mggnluts(ri_id)
```

GRgetnluts returns the number of palettes associated with the identified image if successful and `FAIL` (or `-1`) otherwise. The **GRgetnluts** parameters are further discussed in Table 8J.

8.11.3. Obtaining the Reference Number of a Specified Palette: GRluttoref

Given a palette identifier, **GRluttoref** can be used to obtain the reference number of the specified palette.

The **GRluttoref** routine takes one argument, *pal_id*, a palette identifier, and returns the reference number of the palette. **GRluttoref** is commonly used to annotate the palette or to include the palette within a vgroup. The syntax of **GRgetlutid** is as follows:

```
C:          pal_ref = GRluttoref(pal_id);

FORTRAN:    pal_ref = mglt2rf(pal_id)
```

GRluttoref returns the reference number of the palette if successful and `0` otherwise. The **GRluttoref** parameters are further discussed in Table 8J.

TABLE 8J **GRgetlutid, GRgetlutinfo, and GRluttoref Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRgetlutid [int32] **(mggltid)**	ri_id	int32	integer	Raster image identifier
	lut_index	int32	integer	Palette index
GRluttoref [uint16] **(mglt2rf)**	pal_id	int32	integer	Palette identifier
GRgetnluts [intn] **(mggnluts)**	ri_id	int32	integer	Raster image identifier

8.11.4. Obtaining Palette Information: GRgetlutinfo

Given a palette identifier, **GRgetlutinfo** retrieves information about the palette and its components.

The **GRgetlutinfo** function takes one input argument, *pal_id*, the identifier of the palette, and several return parameters. The return parameters are *n_comps*, the number of components of the palette; *data_type*, the data type of the palette data; *interlace_mode*, the interlace mode of the stored palette data; and *num_entries*, the number of entries in the palette. The syntax of **GRgetlutinfo** is as follows:

```
C:          status = GRgetlutinfo(pal_id, &n_comps, &data_type, &interlace_mode,
                                  &num_entries);

FORTRAN:    status = mgglinf(pal_id, n_comps, data_type, interlace_mode, num_en-
                             tries)
```

GRgetlutinfo returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The **GRgetlutinfo** parameters are further discussed in Table 8J.

8.11.5. Writing Palette Data: GRwritelut

GRwritelut writes palette data into the palette identified by the parameter *pal_id*. The syntax of **GRwritelut** is as follows:

```
C:          status = GRwritelut(pal_id, n_comps, data_type, interlace_mode,
                                num_entries, pal_data);

FORTRAN:    status = mgwrlut(pal_id, n_comps, data_type, interlace_mode, num_en-
                             tries, pal_data)
     OR     status = mgwclut(pal_id, n_comps, data_type, interlace_mode, num_en-
                             tries, pal_data)
```

The parameter *n_comps* specifies the number of pixel components in the palette; it must have a value of at least 1. The parameter *data_type* specifies the data type of the palette data. Refer to Table 2F for all data types supported by HDF.

The parameter *interlace_mode* specifies the interlacing in which the palette is to be written. The valid values of *interlace_mode* are: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`). Refer to Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate*" for further information.

The parameter *num_entries* specifies the number of entries in the palette. The buffer *pal_data* contains the palette data.

Note that the FORTRAN-77 version of **GRwritelut** has two routines; **mgwrlut** writes buffered numeric palette data and **mgwclut** writes buffered character palette data.

GRwritelut returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 8K.

8.11.6. Setting the Interlace Mode for a Palette: GRreqlutil

GRreqlutil sets the interlace mode for the next palette to be read. The syntax of **GRreqlutil** is as follows:

```
C:          status = GRreqlutil(pal_id, interlace_mode);

FORTRAN:    status = mgrltil(pal_id, interlace_mode)
```

The parameter *interlace_mode* specifies the interlacing that will be in effect for the next palette read operation. The valid values of *interlace_mode* are: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`). Refer to Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate*" for further information.

GRreqlutil may be called at anytime before the read operation of the specified palette. In addition, it may be called more than once; the interlace mode setting specified by the last call to the routine will be used for the next read operation.

GRreqlutil returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 8K.

8.11.7. Reading Palette Data: GRreadlut

GRreadlut reads data from the palette identified by the parameter *pal_id*. The syntax of **GRreadlut** is as follows:

```
C:          status = GRreadlut(pal_id, pal_data);

FORTRAN:    status = mgrdlut(pal_id, pal_data)
    OR      status = mgrclut(pal_id, pal_data)
```

The read data will be stored in the buffer *pal_data*, which is assumed to be sufficient to store the read palette data. The sufficient amount of space needed can be determined using the routine **GRgetlutinfo**. The palette data is read according to the interlacing mode set by the last call to **GRreqlutil**.

Note that the FORTRAN-77 version of **GRreadlut** has two routines; **mgrdlut** reads numeric palette data and **mgrclut** reads character palette data.

GRreadlut returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of this routine are further defined in Table 8K.

TABLE 8K

GRgetlutid, GRwritelut, GRreqlutil, and GRreadlut Parameter Lists

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRgetlutinfo [intn] **(mgglinf)**	pal_id	int32	integer	Palette identifier
	n_comps	int32*	integer	Number of components in each palette element
	data_type	int32*	integer	Data type of the palette data
	interlace_mode	int32*	integer	Interlace mode of the palette data
	num_entries	int32*	integer	Buffer for the size of the palette
GRwritelut [intn] **(mgwrlut/ mgwclut)**	pal_id	int32	integer	Palette identifier
	n_comps	int32	integer	Number of components in each palette element
	data_type	int32	integer	Type of the palette data
	interlace_mode	int32	integer	Interlace mode of the palette data
	num_entries	int32	integer	Number of entries in the palette
	pal_data	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer for the palette data to be written
GRreqlutil [intn] **(mgrltil)**	pal_id	int32	integer	Palette identifier
	interlace_mode	intn	integer	Interlace mode for the next palette read operation
GRreadlut [intn] **(mgrdlut/ mgrclut)**	pal_id	int32	integer	Palette identifier
	pal_data	VOIDP	<valid numeric data type>(*)/ character*(*)	Buffer for the palette data to be read

EXAMPLE 7.

Writing a Palette.

This example illustrates the use of the routines **GRgetlutid/mggltid** and **GRwritelut/mgwclut** to attach a palette to a raster image and write data to it.

In this example, the program creates an image named "Image with Palette" in the file "Image_with_Palette.hdf". A palette is then attached to the image and data is written to it.

C:

```
#include "hdf.h"

#define  FILE_NAME         "Image_with_Palette.hdf"
#define  NEW_IMAGE_NAME    "Image with Palette"
#define  N_COMPS_IMG       2       /* number of image components */
#define  X_LENGTH          5
#define  Y_LENGTH          5
#define  N_ENTRIES         256     /* number of entries in the palette */
#define  N_COMPS_PAL       3       /* number of palette's components */

main( )
{
   /************************* Variable declaration *************************/

   intn  status,         /* status for functions returning an intn */
         i, j;
   int32 file_id, gr_id, ri_id, pal_id,
         interlace_mode,
         start[2],     /* holds where to start to write for each dimension  */
```

```
      edges[2],     /* holds how long to write for each dimension */
      dim_sizes[2];  /* sizes of the two dimensions of the image array   */
uint8 image_buf[Y_LENGTH][X_LENGTH][N_COMPS_IMG]; /* data of first image */
uint8 palette_buf[N_ENTRIES][N_COMPS_PAL];

/********************* End of variable declaration *********************/

/*
* Open the HDF file.
*/
file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

/*
* Initialize the GR interface.
*/
gr_id = GRstart (file_id);

/*
* Define the dimensions and interlace mode of the image.
*/
dim_sizes[0] = X_LENGTH;
dim_sizes[1] = Y_LENGTH;
interlace_mode = MFGR_INTERLACE_PIXEL;

/*
* Create the image named NEW_IMAGE_NAME.
*/
ri_id = GRcreate (gr_id, NEW_IMAGE_NAME, N_COMPS_IMG, DFNT_UINT8,
                  interlace_mode, dim_sizes);

/*
* Fill the image data buffer with values.
*/
for (i = 0; i < Y_LENGTH; i++)
{
   for (j = 0; j < X_LENGTH; j++)
   {
      image_buf[i][j][0] = (i + j) + 1;
      image_buf[i][j][1] = (i + j) + 2;
   }
 }

/*
* Define the size of the data to be written, i.e., start from the origin
* and go as long as the length of each dimension.
*/
start[0] = start[1] = 0;
edges[0] = X_LENGTH;
edges[1] = Y_LENGTH;

/*
* Write the data in the buffer into the image array.
*/
status = GRwriteimage (ri_id, start, NULL, edges, (VOIDP)image_buf);

/*
* Initialize the palette to grayscale.
*/
for (i = 0; i < N_ENTRIES; i++) {
   palette_buf[i][0] = i;
   palette_buf[i][1] = i;
   palette_buf[i][2] = i;
}
```

```
   /*
   * Define palette interlace mode.
   */
   interlace_mode = MFGR_INTERLACE_PIXEL;

   /*
   * Get the identifier of the palette attached to the image NEW_IMAGE_NAME.
   */
   pal_id = GRgetlutid (ri_id, 0);

   /*
   * Write data to the palette.
   */
   status = GRwritelut (pal_id, N_COMPS_PAL, DFNT_UINT8, interlace_mode,
                        N_ENTRIES, (VOIDP)palette_buf);

   /*
   * Terminate access to the image and to the GR interface, and
   * close the HDF file.
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

FORTRAN:

```
      program  write_palette
      implicit none
C
C     Parameter declaration
C
      character*22 FILE_NAME
      character*18 NEW_IMAGE_NAME
      integer      X_LENGTH
      integer      Y_LENGTH
      integer      N_ENTRIES
      integer      N_COMPS_IMG
      integer      N_COMPS_PAL
C
      parameter (FILE_NAME       = 'Image_with_Palette.hdf',
     +           NEW_IMAGE_NAME  = 'Image with Palette',
     +           X_LENGTH        = 5,
     +           Y_LENGTH        = 5,
     +           N_ENTRIES       = 256,
     +           N_COMPS_IMG     = 2,
     +           N_COMPS_PAL     = 3)
      integer DFACC_CREATE, DFNT_CHAR8, DFNT_UINT8, MFGR_INTERLACE_PIXEL
      parameter (DFACC_CREATE = 4,
     +           DFNT_CHAR8   = 4,
     +           DFNT_UINT8   = 21,
     +           MFGR_INTERLACE_PIXEL = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgcreat, mgwcimg, mggltid, mgwclut,
     +        mgendac, mgend
C
C**** Variable declaration *******************************************
C
      integer    file_id, gr_id, ri_id, pal_id
      integer    interlace_mode
      integer    start(2), stride(2), edges(2), dim_sizes(2)
      integer    status
      integer    i, j
      character  image_buf(N_COMPS_IMG, X_LENGTH, Y_LENGTH)
      character  palette_buf(N_COMPS_PAL, N_ENTRIES)
C
C**** End of variable declaration ***********************************
C
C
C     Create and open the file.
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Define interlace mode and dimensions of the image.
C
      interlace_mode = MFGR_INTERLACE_PIXEL
      dim_sizes(1) = X_LENGTH
      dim_sizes(2) = Y_lENGTH
C
C     Create the raster image array.
C
      ri_id = mgcreat(gr_id, NEW_IMAGE_NAME, N_COMPS_IMG, DFNT_CHAR8,
     +                interlace_mode, dim_sizes)
C
C     Fill the image data buffer with values.
C
      do 20 i = 1, Y_LENGTH
         do 10 j = 1, X_LENGTH
               image_buf(1,j,i) = char(i + j - 1 )
               image_buf(2,j,i) = char(i + j)
10       continue
20    continue

C
C     Define the size of the data to be written, i.e., start from the origin
C     and go as long as the length of each dimension.
C
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
C
C     Write the data in the buffer into the image array.
C
      status = mgwcimg(ri_id, start, stride, edges, image_buf)
C
C     Initilaize the palette buffer to grayscale.
C
      do 40 i = 1, N_ENTRIES
          do 30 j = 1, N_COMPS_PAL
             palette_buf(j,i) = char(i)
30        continue
40    continue
C
C     Get the identifier of the palette attached to the image NEW_IMAGE_NAME.
C
```

```
      pal_id = mggltid(ri_id, 0)
C
C     Set palette interlace mode.
C
      interlace_mode = MFGR_INTERLACE_PIXEL
C
C     Write data to the palette.
C
      status = mgwclut(pal_id, N_COMPS_PAL, DFNT_UINT8, interlace_mode,
     +                 N_ENTRIES, palette_buf)
C
C     Terminate access to the raster image and to the GR interface,
C     and close the HDF file.
C
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      end
```

EXAMPLE 8.

Reading a Palette.

This example illustrates the use of the routines **GRgetlutinfo/mgglinf** and **GRreadlut/mgrclut** to obtain information about a palette and to read palette data.

In this example, the program finds and selects the image named "Image with Palette" in the file "Image_with_Palette.hdf". Then the program obtains information about the palette and reads the palette data.

C:

```
#include "hdf.h"

#define  FILE_NAME      "Image_with_Palette.hdf"
#define  IMAGE_NAME     "Image with Palette"
#define  N_ENTRIES      256     /* number of elements of each color */

main( )
{
   /************************* Variable declaration *************************/

   intn  status,         /* status for functions returning an intn */
         i, j;
   int32 file_id, gr_id, ri_id, pal_id, ri_index;
   int32 data_type, n_comps, n_entries, interlace_mode;
   uint8 palette_data[N_ENTRIES][3];        /* static because of fixed size */

   /************************* Variable declaration *************************/

   /*
   * Open the file.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initiate the GR interface.
   */
   gr_id = GRstart (file_id);

   /*
   * Get the index of the image IMAGR_NAME.
   */
   ri_index = GRnametoindex (gr_id, IMAGE_NAME);

   /*
   * Get image identifier.
   */
   ri_id = GRselect (gr_id, ri_index);

   /*
   * Get the identifier of the palette attached to the image.
   */
   pal_id = GRgetlutid (ri_id, ri_index);

   /*
   * Obtain and display information about the palette.
   */
   status = GRgetlutinfo (pal_id, &n_comps, &data_type, &interlace_mode,
                          &n_entries);
   printf ("Palette: %d components; %d entries\n", n_comps, n_entries);

   /*
   * Read the palette data.
   */
   status = GRreadlut (pal_id, (VOIDP)palette_data);

   /*
   * Display the palette data.  Recall that HDF supports only 256 colors.
   * Each color is defined by its 3 components. Therefore,
   * verifying the value of n_entries and n_comps is not necessary and
   * the buffer to hold the palette data can be static.  However,
   * if more values or colors are added to the model, these parameters
   * must be checked to allocate sufficient space when reading a palette.
   */
   printf ("  Palette Data: \n");
   for (i=0; i< n_entries; i++)
   {
      for (j = 0; j < n_comps; j++)
         printf ("%i ", palette_data[i][j]);
      printf ("\n");
   }
   printf ("\n");

   /*
   * Terminate access to the image and to the GR interface, and
   * close the HDF file.
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

FORTRAN:

```
      program  read_palette
      implicit none
C
C     Parameter declaration
C
      character*22 FILE_NAME
      character*18 IMAGE_NAME
      integer      N_ENTRIES
      integer      N_COMPS_PAL
C
      parameter (FILE_NAME   = 'Image_with_Palette.hdf',
     +           IMAGE_NAME  = 'Image with Palette',
     +           N_COMPS_PAL = 3,
     +           N_ENTRIES   = 256)
      integer DFACC_READ, DFNT_CHAR8, DFNT_UINT8, MFGR_INTERLACE_PIXEL
      parameter (DFACC_READ  = 1,
     +           DFNT_CHAR8  = 4,
     +           DFNT_UINT8  = 21,
     +           MFGR_INTERLACE_PIXEL = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer mgstart, mgn2ndx, mgselct, mggltid, mgglinf,
     +        mgrclut, mgendac, mgend
C
C**** Variable declaration *******************************************
C
      integer    file_id, gr_id, ri_id, ri_index, pal_id, pal_index
      integer    interlace_mode
      integer    data_type, n_comps, n_entries_out
      integer    status
      integer    i, j
      character  palette_data(N_COMPS_PAL, N_ENTRIES)
C
C**** End of variable declaration ************************************
C
C
C     Open the file.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the GR interface.
C
      gr_id = mgstart(file_id)
C
C     Get the index of the image IMAGE_NAME.
C
      ri_index = mgn2ndx(gr_id, IMAGE_NAME)
C
C     Get the image identifier.
C
      ri_id = mgselct(gr_id, 0)
C
C     Get the identifier of the palette attached to the image.
C
      pal_index = 0
      pal_id = mggltid(ri_id, pal_index)
C
C     Obtain information about the palette.
C
      status = mgglinf(pal_id, n_comps, data_type, interlace_mode,
     +                 n_entries_out)
      write(*,*) ' Palette: ', n_comps, ' components;  ',
     +           n_entries_out, ' entries'
C
C     Read the palette.
C
      status = mgrclut(pal_id, palette_data)
C
C     Display the palette data.
C
      write(*,*) "Palette data"
      do 10 i = 1, n_entries_out
```

```
            write(*,*) (ichar(palette_data(j,i)), j = 1, n_comps)
10      continue
C
C       Terminate access to the raster image and to the GR interface,
C       and close the HDF file.
C
        status = mgendac(ri_id)
        status = mgend(gr_id)
        status = hclose(file_id)
        end
```

8.12. Chunked Raster Images

The GR interface also supports chunking in a manner similar to that of the SD interface. There is one restriction on a raster image: it must be created with MFGR_INTERLACE_PIXEL (or `0`) in the call to **GRcreate**. We refer the reader to Section 3.11 of Chapter 3, *Scientific Data Sets (SD API)*, and to Chapter 14, *HDF Performance Issues*, for discussions of chunking concepts and performance related topics. The GR interface provides three routines, **GRsetchunk**, **GRsetchunkcache**, and **GRgetchunkinfo**, to create and maintain chunked raster images. The generic functions for reading and writing GR images, **GRwriteimage** and **GRreadimage**, will write and read chunked raster images as well. However, the GR interface provides special write and read routines, **GRwritechunk** and **GRreadchunk,** which are similar to **SDwritechunk** and **SDreadchunk**. Compared to **GRwriteimage** and **GRreadimage**, **GRwritechunk** and **GRreadchunk** are low-overhead but are only sutable for writing or reading complete chunks.

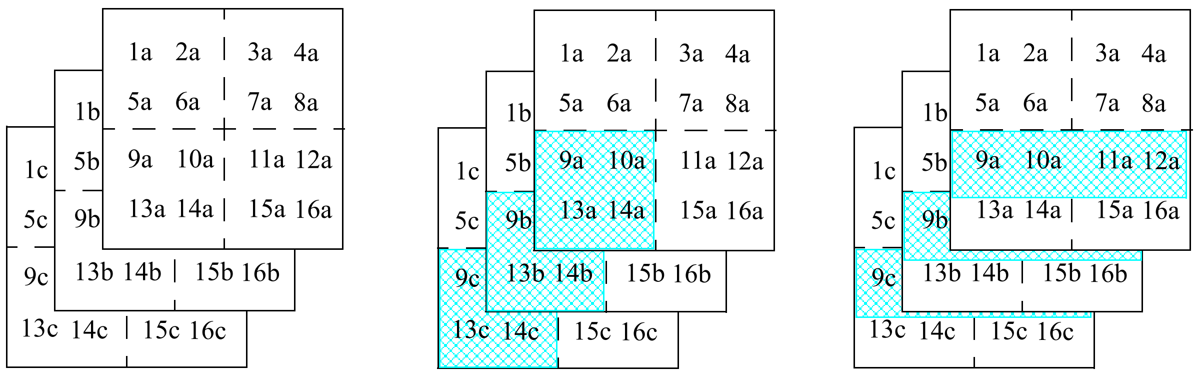
8.12.1. Difference between a Chunked Raster Image and a Chunked SDS

Chunks of scientific datasets (SDSs) have the same dimensionality as the SDS itself and the chunks can divide the SDS along any dimension. While raster images under the GR interface are actually 3-dimensional arrays, 2 dimensions define the image while the third dimension (the stack of 2-dimensional image planes) provides the composite definition of the color at each pixel of the 2-dimensional image. Chunking can be applied only across the 2-dimensions of the image; chunking cannot divide the array across the third dimension. In other words, all of the elements of the raster image that define a single pixel must remain together in the same chunk.

FIGURE 8b **Chunks in a GR raster image dataset**

Multiple layers of a GR raster image. For example, 1a, 1b, and 1c fully define the color of pixel 1.

GR dataset chunking can divide a dataset only across the 2 dimensions of the image; the chunks cannot divide the planes, which consitute the third dimension of the dataset.



Unchunked GR dataset Chunked GR dataset Alternate GR dataset chunking format

8.12.2. Making a Raster Image a Chunked Raster Image: GRsetchunk

GRsetchunk makes the raster image, identified by the parameter *ri_id*, a chunked raster image according to the provided chunking and compression information. The syntax of **GRsetchunk** is as follows:

```
C:          status = GRsetchunk(ri_id, c_def, flags);

FORTRAN:    status = mgschnk(ri_id, dim_length, comp_type, comp_prm)
```

The parameters *c_def* and *flags* in C or the parameters *comp_type* and *comp_prm* in FORTRAN-77 provide the chunking and compression information and are discussed below.

In C:

The parameter *c_def* is a union of type `HDF_CHUNK_DEF`, which is defined as follows:

```
typedef union hdf_chunk_def_u
    {
     int32 chunk_lengths[2];  /* chunk lengths along each dim */

     struct
          {
           int32 chunk_lengths[2];
           int32 comp_type;             /* compression type */
           struct comp_info cinfo;
          } comp;

     struct
          {
          /* is not used in GR interface */
          } nbit;
     } HDF_CHUNK_DEF
```

Valid values of the parameter *flags* are `HDF_CHUNK` for chunked and uncompressed data and (`HDF_CHUNK | HDF_COMP`) for chunked and compressed data. Data can be compressed using run-length encoding (RLE), Skipping Huffman, GZIP, or Szip compression algorithms.

If the parameter *flags* has a value of `HDF_CHUNK`, the chunk dimensions must be specified in the field `c_def.chunk_lengths[]`. If the parameter *flags* has a value of (`HDF_CHUNK | HDF_COMP`), the chunk dimensions must be specified in the field `c_def.comp.chunk_lengths[]` and the compression type in the field `c_def.comp.comp_type`. Valid values of compression type values are:

`COMP_CODE_NONE` (or `0`) for uncompressed data
`COMP_CODE_RLE` (or `1`) for RLE compression
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for Szip compression

For Skipping Huffman, GZIP, and Szip compression methods, parameters are passed in corresponding fields of the structure *cinfo*. Specify skipping size for Skipping Huffman compression in the field `c_def.comp.cinfo.skphuff.skp_size`; this value cannot be less than 1. Specify deflate level for GZIP compression in the field `c_def.comp.cinfo.deflate_level`. Valid values of deflate levels are integers from 0 to 9 inclusive. Specify the Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset in the fields `c_info.szip.options_mask` and `c_info.szip.pixels_per_block`, respectively.

Refer to the discussion of **SDsetcompress** routine in Section "*Compressing SDS Data: SDsetcompress"* for the definition of the structure *comp_info*.

In FORTRAN-77:

Chunk dimensions are specified in the array *dim_length* and the compression type in the parameter *comp_type*. Valid compression types and their values are defined in the `hdf.inc` file and are listed below:

`COMP_CODE_NONE` (or `0`) for uncompressed data
`COMP_CODE_RLE` (or `1`) for RLE compression
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression

The parameter *comp_prm* specifies the compression parameters for the Skipping Huffman and GZIP compression methods. It contains only one element which is set to the skipping size for Skipping Huffman compression or the deflate level for GZIP compression. Currently, Szip compression is not yet supported by Fortran GR interface.

GRsetchunk returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The **GRsetchunk** parameters are discussed further in Table 8L

8.12.3. Writing a Chunked Raster Image: GRwritechunk

GRwritechunk is used to write a chunk of a chunked raster image. The syntax of the **GRwritechunk** routine is as follows:

```
C:          status = GRwritechunk(ri_id, &origin, &datap);

FORTRAN:    status = mgwchnk(ri_id, origin, datap)
            status = mgwcchnk(ri_id, origin, datap)
```

GRwritechunk writes the entire chunk of data stored in the buffer *datap* to the chunked raster image identified by the parameter *ri_id*. Writing starts at the location specified by the parameter *origin*. This function has less overhead than **GRwriteimage** and should be used whenever an entire chunk of data is to be written.

The raster image must be stored in pixel-interlace mode.

The parameter *origin* is a two-dimensional array which specifies the coordinates of the chunk according to the chunk position in the overall chunk array.

The *datap* buffer contains the chunk data. The data must be organized in pixel-interlace mode.

Note that the FORTRAN-77 version of **GRwritechunk** has two routines; **mgwchnk** writes buffered numeric data and **mgwcchnk** writes buffered character data.

GRwritechunk returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The **GRwritechunk** parameters are discussed further in Table 8L.

EXAMPLE 9.

Creating and Writing a Chunked Raster Image

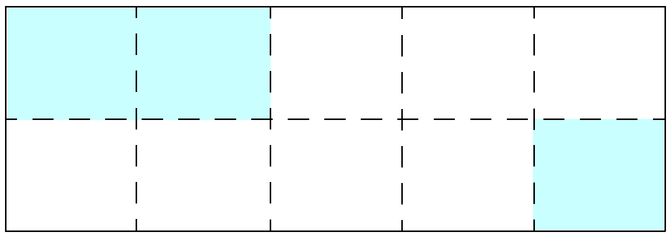
This example illustrates the use of the routines **Hopen**/**hopen**, **GRstart**/**mgstart**, **GRcreate**/**mgcreat**, **GRwritechunk**/**mgwchnk**, **GRendaccess**/**mgendac**, **GRend**/**mgend**, and **Hclose**/**hclose** to create an HDF file and store a raster image in it.

In this example, the program creates an image of 6 rows by 10 columns in C and 10 rows by 6 columns in FORTRAN. The image is set up to be chunked with a chunk size of 3x2 in C and 2x3 in FORTRAN and compressed with the GZIP method. Three chunks are then written to the image. See Figure 8c through Figure 8d for illustrations.

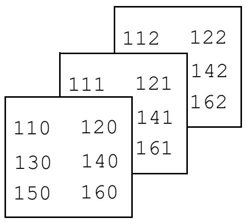
FIGURE 8c

Chunked GR image as written by C example

This image has 10 chunks, each 3x2 in size. This program writes data to the first, second, and last chunks, as indicated by the shading to the right.



Each chunk contains the data for all three planes of the images. The first chunk, for example would be illustrated as below.



Upon completion of the program, the three planes of the image contain the following data.

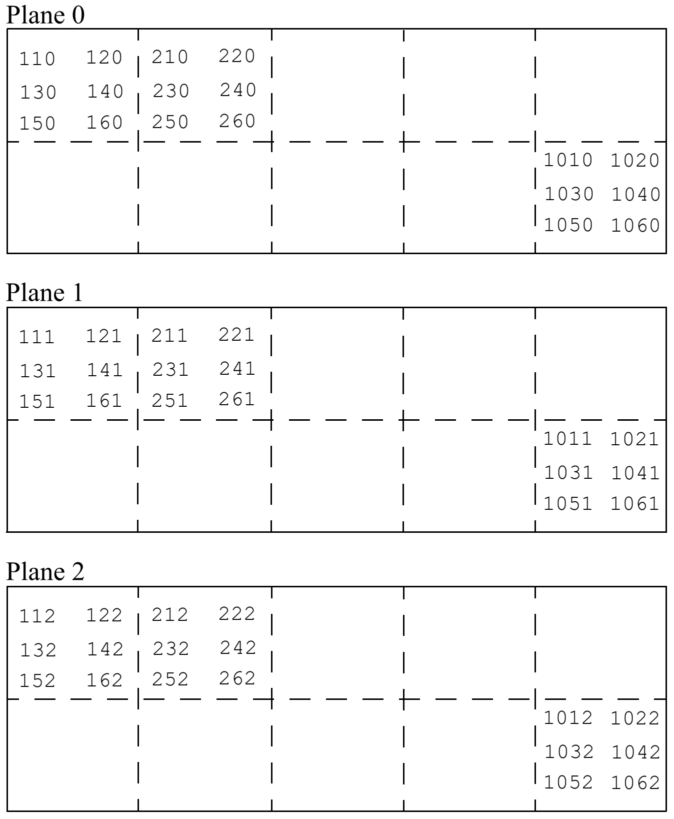
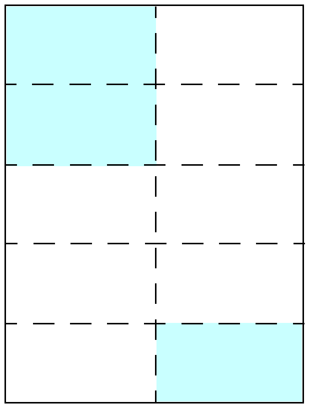
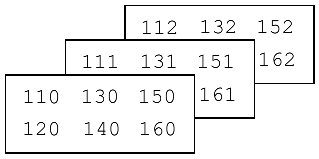


FIGURE 8d **Chunked GR image as written by FORTRAN example**

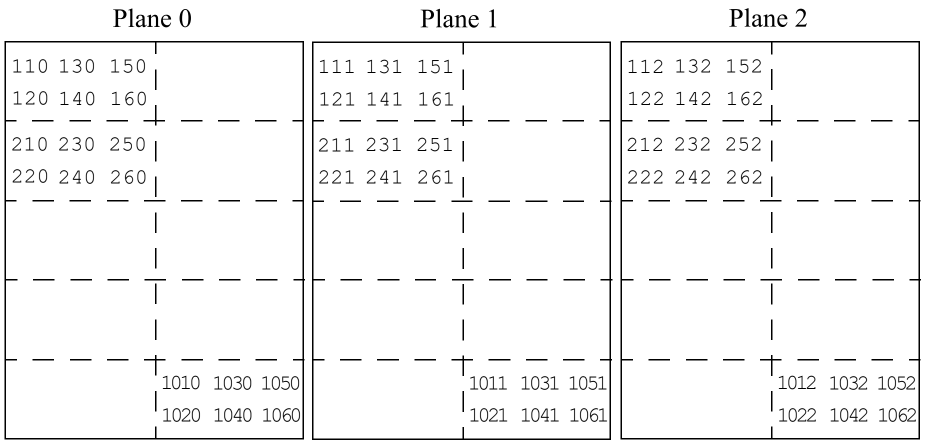
This image has 10 chunks, each 2x3 in size. This program writes data to the first, second, and last chunks, as indicated by the shading below.



Each chunk contains the data for all three planes of the images. The first chunk, for example would be illustrated as below.



Upon completion of the program, the three planes of the image contain the following data.



C:

```
#include "hdf.h"

#define FILE_NAME     "Image_Chunked.hdf"
#define IMAGE_NAME    "Image with Chunks"
#define X_LENGTH      10    /* number of rows in the image */
#define Y_LENGTH      6     /* number of columns in the image */
#define NCOMPS        3     /* number of components in the image */

int main()
{
   /************************* Variable declaration *************************/

   intn  status;         /* status for functions returning an intn */
   int32 file_id,        /* HDF file identifier */
         gr_id,          /* GR interface identifier */
         ri_id,          /* raster image identifier */
         dims[2],        /* dimension sizes of the image array */
         origin[2],      /* origin position to write each chunk */
         interlace_mode; /* interlace mode of the image */
   HDF_CHUNK_DEF chunk_def;     /* Chunk defintion set */
   int32 chunk00[] = {1, 2, 3, 4, 5, 6,
                      7, 8, 9, 10, 11, 12,
                      13, 14, 15, 16, 17, 18 };

   int32 chunk01[] = {210, 211, 212, 220, 221, 222,
                      230, 231, 232, 240, 241, 242,
                      250, 251, 252, 260, 261, 262};

   int32 chunk14[] = {1010, 1011, 1012, 1020, 1021, 1022,
                      1030, 1031, 1032, 1040, 1041, 1042,
                      1050, 1051, 1052, 1060, 1061, 1062};

   /********************** End of variable declaration **********************/

   /*
   * Create and open the file.
   */
   file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

   /*
   * Initialize the GR interface.
   */
   gr_id = GRstart (file_id);

   /*
   * Set dimensions of the image.
   */
   dims[0] = Y_LENGTH;
   dims[1] = X_LENGTH;

   /*
   * Create the raster image array.
   */
   ri_id = GRcreate (gr_id, IMAGE_NAME, NCOMPS, DFNT_INT32,
                     MFGR_INTERLACE_PIXEL, dims);
   /*
   * Define chunked image.
   */
   chunk_def.comp.comp_type = COMP_CODE_DEFLATE;
   chunk_def.comp.cinfo.deflate.level = 6;
   chunk_def.comp.chunk_lengths[0] = 3;
   chunk_def.comp.chunk_lengths[1] = 2;
   status = GRsetchunk (ri_id, chunk_def, HDF_CHUNK | HDF_COMP);

   /*
   * Write first chunk(0,0).
   */
   origin[0] = 0;
   origin[1] = 0;
   status = GRwritechunk (ri_id, origin, (VOIDP)chunk00);

   /*
   * Write second chunk(0,1).
   */
   origin[0] = 0;
   origin[1] = 1;
   status = GRwritechunk (ri_id, origin, (VOIDP)chunk01);

   /*
   * Write third chunk(1,4).
   */
   origin[0] = 1;
   origin[1] = 4;
   status = GRwritechunk (ri_id, origin, (VOIDP)chunk14);

   /*
   * Terminate access to the raster image and to the GR interface and,
   * close the HDF file.
```

```
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
   return 0;
}
```

FORTRAN:

```
      program gr_chunking_example
      implicit none
C
C     Parameter declaraction
C
      character*14 FILE_NAME
      character*14 DATASET_NAME
      parameter (FILE_NAME = ’gr_chunked.hdf’,
     .           DATASET_NAME = ’gzip_comp_data’)
      integer   NCOMP, MFGR_INTERLACE_PIXEL
      parameter(NCOMP = 3, MFGR_INTERLACE_PIXEL = 0)
      integer DFACC_CREATE, DFACC_READ, DFACC_WRITE
      parameter (DFACC_CREATE = 4,
     .           DFACC_READ   = 1,
     .           DFACC_WRITE  = 2)
      integer DFNT_INT32
      parameter (DFNT_INT32   = 24)
      integer X_LENGTH, Y_LENGTH, X_CH_LENGTH, Y_CH_LENGTH
      parameter (X_LENGTH     = 6,
     .           Y_LENGTH     = 10,
     .           X_CH_LENGTH  = 3,
     .           Y_CH_LENGTH  = 2)
C
C     Compression parameters.
C
      integer  COMP_CODE_DEFLATE, DEFLATE_LEVEL
      parameter( COMP_CODE_DEFLATE = 4, DEFLATE_LEVEL = 6)
C
C     Function declaration.
C
      integer mgstart, mgcreat, mgendac, mgend
      integer mgwchnk, mgschnk
      integer hopen, hclose
C
C**** Variable declaration *************************************************
C
      integer ri_id, gr_id, file_id
      integer dims(2), start(2)
      integer status, il
      integer comp_prm(1), comp_type
C
C     Data buffers.
C
      integer*4 chunk11(NCOMP* X_CH_LENGTH*Y_CH_LENGTH)
      integer*4 chunk21(NCOMP* X_CH_LENGTH*Y_CH_LENGTH)
      integer*4 chunk52(NCOMP* X_CH_LENGTH*Y_CH_LENGTH)
C
C     Chunking dimension arrays
C
      integer ch_dims(2)
C
C**** End of variable declaration *******************************************
C
C
```

```
C     Data initialization
C
      data  chunk11 / 110, 111, 112, 120, 121, 122,
     .                130, 131, 132, 140, 141, 142,
     .                150, 151, 152, 160, 161, 162
     .              /,
     .      chunk21 /
     .                210, 211, 212, 220, 221, 222,
     .                230, 231, 232, 240, 241, 242,
     .                250, 251, 252, 260, 261, 262
     .              /,
     .      chunk52 /
     .                1010, 1011, 1012, 1020, 1021, 1022,
     .                1030, 1031, 1032, 1040, 1041, 1042,
     .                1050, 1051, 1052, 1060, 1061, 1062
     .              /
C
C     Define chunk dimensions.
C
      ch_dims(1) = Y_CH_LENGTH
      ch_dims(2) = X_CH_LENGTH
C
C     Create and open the file and initiate GR interface..
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
      gr_id   = mgstart(file_id)
C
C     Define the number of components and dimensions of the image.
C
      il      = MFGR_INTERLACE_PIXEL
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH
C
C     Create GR dataset.
C
      ri_id = mgcreat(gr_id, DATASET_NAME, NCOMP, DFNT_INT32, il, dims)
C

C     Define chunked GR dataset using GZIP compression.
C
      comp_prm(1) =  DEFLATE_LEVEL
      comp_type = COMP_CODE_DEFLATE
      status = mgschnk (ri_id, ch_dims, comp_type, comp_prm)
C
C     Define the location of the first chunk and write the data.
C
      start(1) = 1
      start(2) = 1
      status = mgwchnk(ri_id, start, chunk11)
C
C     Define the location of the second chunk and write the data.
C
      start(1) = 2
      start(2) = 1
      status = mgwchnk(ri_id, start, chunk21)
C
C     Define the location of the third and write the data.
C
      start(1) = 5
      start(2) = 2
      status = mgwchnk(ri_id, start, chunk52)
C
C     Terminate access to the array.
```

```
C
      status = mgendac(ri_id)
C
C     Terminate access to the GR interface.
C
      status = mgend(gr_id)
C
C     Close the file.
C
      status = hclose(file_id)
      end
```

8.12.4. Reading a Chunked Raster Image: GRreadchunk

GRreadchunk is used to read an entire chunk of data from a chunked raster image. The syntax of the **GRreadchunk** routine is as follows:

```
C:          status = GRreadchunk(ri_id, &origin, datap);

FORTRAN:    status = mgrchnk(ri_id, origin, datap)

            status = mgrcchnk(ri_id, origin, datap)
```

GRreadchunk reads the entire chunk of data stored from the chunked raster image identified by the parameter *ri_id* and stores it in the buffer *datap*. The chunk to be read is specified by the parameter *origin*. This function has less overhead than **GRreadimage** and should be used whenever an entire chunk of data is to be read.

The raster image must be stored in pixel-interlace mode.

The parameter *origin* is a two-dimensional array which specifies the coordinates of the chunk according to the chunk position in the overall chunk array.

The *datap* buffer contains the chunk data. The data is organized in pixel-interlace mode.

Note that the FORTRAN-77 version of **GRreadchunk** has two routines; **mgrchnk** reads numeric data and **mgrcchnk** reads character data to the buffer.

GRreadchunk returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. **GRreadchunk** will return `FAIL` (or `-1`) when an attempt is made to read from a non-chunked image. The **GRreadchunk** parameters are discussed further in Table 8L.

EXAMPLE 10. **Reading a Chunked Raster Image.**

This example illustrates the use of the routines **GRreadchunk/mgrchnk** to read the raster image's chunked data.

In this example, the program finds and selects the image named "Image with Chunks" in the file "Image_Chunked.hdf". Then the program obtains information about the image and reads the image data. Only C example is available at this time.

C:

```
#include "hdf.h"

#define  FILE_NAME      "Image_Chunked.hdf"
#define  IMAGE_NAME     "Image with Chunks"
#define X_LENGTH      10    /* number of rows in the image */
#define Y_LENGTH      6     /* number of columns in the image */
```

```
#define NCOMPS        3     /* number of components in the image */

int main()
{
   /************************ Variable declaration ************************/

   intn  status;         /* status for functions returning an intn */
   int32 file_id,        /* HDF file identifier */
         gr_id,          /* GR interface identifier */
         ri_id,          /* raster image identifier */
         dims[2],        /* dimension sizes of the image array */
         origin[2],      /* origin position to write each chunk */
         interlace_mode; /* interlace mode of the image */
   HDF_CHUNK_DEF chunk_def;     /* Chunk defintion set */

   /************************ Variable declaration ************************/

   /*
   * Open the file.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initiate the GR interface.
   */
   gr_id = GRstart (file_id);

   /*
   * Get the index of the image IMAGR_NAME.
   */
   ri_index = GRnametoindex (gr_id, IMAGE_NAME);

   /*
   * Get image identifier.
   */
   ri_id = GRselect (gr_id, ri_index);

   /*
   * Set dimensions of the image.
   */
   dims[0] = X_LENGTH;
   dims[1] = Y_LENGTH;
   start[0] = start[1] = 0;
   edges[0] = dims[0];
   edges[1] = dims[1];

   /* Read the data in the image array. */
   status = GRreadimage (ri_id, start, NULL, edges, (VOIDP)image_data);

   /*
   * Terminate access to the image and to the GR interface, and
   * close the HDF file.
   */
   status = GRendaccess (ri_id);
   status = GRend (gr_id);
   status = Hclose (file_id);
}
```

8.12.5. Obtaining Information about a Chunked Raster Image: GRgetchunkinfo

GRgetchunkinfo is used to determine whether a raster image is chunked and how chunking is defined. The syntax of the **GRgetchunkinfo** routine is as follows:

```
C:          status = GRgetchunkinfo(ri_id, &c_def, &flag);

FORTRAN:    status = mggichnk(ri_id, dim_length, flag)
```

GRgetchunkinfo retrieves chunking information about the raster image into the parameters *c_def* and *flag* in C and into the parameters *dim_length* and *flag* in FORTRAN-77. Note that only chunk dimensions are retrieved; compression information is not available.

The value returned in the parameter *flag* indicates whether the raster image is not chunked, chunked, or chunked and compressed. `HDF_NONE` (or `-1`) indicates that the raster image is not chunked. `HDF_CHUNK` (or `0`) indicates that the raster image is chunked and not compressed. (`HDF_CHUNK | HDF_COMP`) (or `1`) indicates that raster image is chunked and compressed with one of the allowed compression methods: RLE, Skipping Huffman, or GZIP.

In C, if the raster image is chunked and not compressed, **GRgetchunkinfo** fills the array *chunk_lengths* in the union *c_def* with the values of the corresponding chunk dimensions. If the raster image is chunked and compressed, **GRgetchunkinfo** fills the array *chunk_lengths* in the structure *comp* of the union *c_def* with the values of the corresponding chunk dimensions. Refer to Section "*Making a Raster Image a Chunked Raster Image: GRsetchunk"* on **GRsetchunk** for specific information on the union `HDF_CHUNK_DEF`. In C, if the chunk length for each dimension is not needed, `NULL` can be passed in as the value of the parameter *c_def*.

In FORTRAN-77, chunk dimensions are retrieved into the array *dim_length*.

GRgetchunkinfo returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. The **GRgetchunkinfo** parameters are discussed further in Table 8L.

8.12.6. Setting the Maximum Number of Chunks in the Cache: GRsetchunkcache

GRsetchunkcache sets the maximum number of chunks to be cached for chunked raster image. **GRsetchunkcache** has similar behavior to **SDsetchunkcache**. Refer to Section "*Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache"* for specific information. The syntax of **GRsetchunkcache** is as follows:

```
C:          status = GRsetchunkcache(ri_id, maxcache, flags);

FORTRAN:    status = mgscchnk(ri_id, maxcache, flags)
```

The maximum number of chunks is specified by the parameter *maxcache*. Currently, the only valid value of the parameter *flags* is `0`.

If **GRsetchunkcache** is not called, the maximum number of chunks in the cache is set to the number of chunks along the fastest-changing dimension. Since **GRsetchunkcache** is similar to the routine **SDsetchunkcache**, refer to Section "*Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache"* for more detailed discussion of the routine's behavior.

GRsetchunkcache returns the value of the parameter *maxcache* if successful and `FAIL` (or `-1`) otherwise. The **GRsetchunkcache** parameters are discussed further in Table 8L.

TABLE 8L **GRsetchunk, GRgetchunkinfo, GRsetchunkcache, GRwritechunk, and**

GRreadchunk Parameter Lists

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
GRsetchunk [intn] **(mgschnk)**	ri_id	int32	integer	Raster image identifier
	c_def	HDF_CHUNK_DEF	N/A	Chunk definition
	flags	int32*	N/A	Compression flags
	dim_length	N/A	integer	Chunk dimensions array
	comp_type	N/A	integer	Type of compression
	comp_prm	N/A	integer	Compression parameters array
GRgetchunkinfo [intn] **(mggichnk)**	ri_id	int32	integer	Raster image identifier
	c_def	HDF_CHUNK_DEF	N/A	Chunk definition
	dim_length	N/A	integer	Chunk dimensions array
	flag	int32	integer	Compression flag
GRsetchunkcache [intn] **(mgscchnk)**	ri_id	int32	integer	Raster image identifier
	maxcache	int32	integer	Maximum number of chunks to cache
	flags	int32	integer	Flags determining routine behavior
GRreadchunk **(mgrchnk/ mgrcchnk)**	ri_id	int32	integer	Raster image identifier
	origin	int32	integer	Array specifying the coordinates of the chunk
	datap	VOIDP	<valid_numeric_or_char_data_type>	Buffer with chunk data in pixel interlace mode
GRwritechunk [intn] **(mgwchnk/ mgwcchnk)**	ri_id	int32	integer	Raster image identifier
	origin	int32	integer	Array specifying the coordinates of the chunk
	datap	const VOIDP	<valid_numeric_or_char_data_type>	Buffer with chunk data in pixel interlace mode

CHAPTER 9 --

Palettes (DFP API)

9.1. Chapter Overview

This chapter describes the routines available for storing and retrieving 8-bit palettes. An 8-bit palette is a look-up table with 256 entries, one entry for each of the 256 possible pixel values the system hardware associates with a particular color. This chapter introduces and describes the HDF palette data model and the DFP interface.

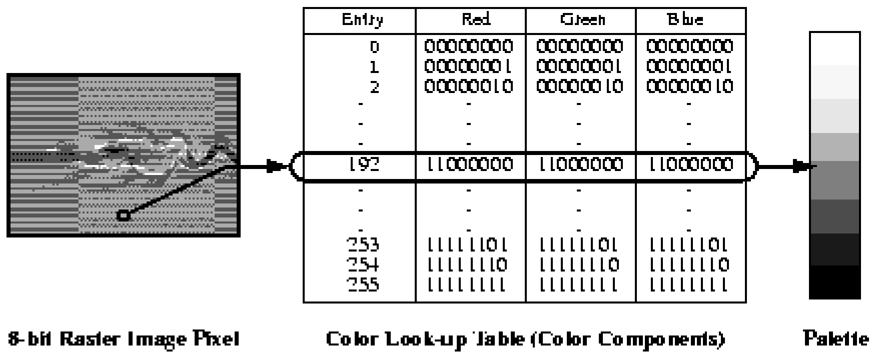
Note: This interface is now deprecated and superseded by the *General Raster Images (GR API)* interface (Chapter 8.)

9.2. The Palette Data Model

A ***palette*** is the means by which color is applied to an image and is also referred to as a ***color lookup table***. It is a table in which every row contains the numerical representation of a particular color. Palettes can be many different sizes, but HDF only supports palettes with 256 colors, corresponding to the 256 different possible pixel values (0 to 255) in 8-bit raster images.

For each of the 256 colors in a palette, there are three 8-bit numbers describing its appearance. (See Figure 9a) Each 8-bit ***color component*** represents the amount of red (or "R"), green (or "G"), or blue (or "B") used to create a particular color. In HDF, 8-bit palettes are assumed to be organized as follows; each entry consists of three bytes: one each for R, G, and B value. The first group of three bytes represent the R, G, and B values of the first color in the palette; the next three the R, G, and B values of the second color; and so forth. Therefore, the 256 possible different pixel values in an image serve as an index for the 256 color entries stored in the palette.

FIGURE 9a **Color Mapping Using a Palette**



Entry	Red	Green	Blue
0	00000000	00000000	00000000
1	00000001	00000001	00000001
2	00000010	00000010	00000010
-	-	-	-
-	-	-	-
-	-	-	-
192	11000000	11000000	11000000
-	-	-	-
-	-	-	-
-	-	-	-
253	11111101	11111101	11111101
254	11111110	11111110	11111110
255	11111111	11111111	11111111

In the HDF library, there are four interfaces that support the reading and writing of palette data; the raster image interfaces, covered in Chapter 6, *8-Bit Raster Images (DFR8 API)*, Chapter 7, *24-*

bit Raster Images (DF24 API), *Chapter 9, Palettes (DFP API)*, and the DFP palette interface covered in this chapter. The raster image interfaces store palettes with raster images and the palette interface reads and writes palettes outside of raster image sets. Palettes stored using the palette interface are stored as isolated data objects. In other words they are not included as members of any set, although they can be grouped with other objects using the Vgroup interface. For more information on the Vgroup interface, refer to Chapter 5, *Vgroups (V API)*.

9.3. The Palette API

The DFP interface consists of eight routines. The routines DFPaddpal and DFPgetpal are the primary routines for palette I/O and are used for most reading and writing operations.

9.3.1. Palette Library Routines

All C functions in the palette interface are prefaced by "DFP" and the equivalent FORTRAN-77 functions are prefaced by "dp". These routines are divided into the following categories:

- ***Write routines*** store palettes in new files or append them to existing files.
- ***Read routines*** sequentially or randomly locate palettes to be read from a named file.

The DFP function calls are more explicitly defined in the following table and in the *HDF Reference Manual*.

TABLE 9A **DFP Library Routines**

Category	Routine Names		Description
	C	FORTRAN-77	
Write	DFPaddpal	dpapal	Appends a palette to a file.
	DFPputpal	dpppal	Writes a palette to a file.
	DFPwrit-eref	dpwref	Sets the reference number for writing the next palette.
Read	DFPgetpal	dpgpal	Retrieves the next palette in a file.
	DFPlastref	dplref	Returns the value of the last reference number read or written.
	DFPnpals	dpnpals	Returns the number of palettes in a file.
	DFPreadref	dprref	Sets reference number for retrieving the next palette.
	DFPrestart	dprest	Specifies that the next read call will get the first palette in the file.

9.4. Writing Palettes

9.4.1. Writing a Palette: DFPaddpal and DFPputpal

To write a palette to an HDF file, the calling program must contain one of the following function calls:

```
C:          status = DFPaddpal(filename, palette);

FORTRAN:    status = dpapal(filename, palette)
    OR

C:          status = DFPputpal(filename, palette, overwrite, filemode);

FORTRAN:    status = dpppal(filename, palette, overwrite, filemode)
```

DFPaddpal and **DFPputpal** will write a palette to an HDF file named by *filename*. When given a new filename, **DFPputpal** and **DFPaddpal** creates a new file and writes the palette as the first object in the file. When given an existing filename, **DFPaddpal** appends the palette to the end of the file.

DFPputpal provides this functionality as well with additional options for how the data is handled, providing more control over how a palette is written to file than **DFPaddpal**. Specifically, the *overwrite* parameter determines whether or not to overwrite the last palette written to a file or to append a new palette onto the file. The *filemode* parameter determines whether to create a new file or to append the data to the previous file. Note the combination to overwrite a palette in a newly created file is invalid and will generate an error. To overwrite a palette, *filename* must be the same filename as the last file accessed through the DFP interface. The parameters for **DFPaddpal** and **DFPputpal** are more explicitly defined in the following table.

TABLE 9B **DFPputpal and DFPaddpal Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFPputpal [intn] **(dpppal)**	filename	char *	character*(*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte space for palette.
	overwrite		integer	Palette write specification.
	filemode	char *	character*(*)	File write specification.
DFPaddpal [intn] **(dpapal)**	filename	char *	character*(*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte space with palette.

Calling **DFPaddpal** or **DFPputpal** immediately after writing an 8-bit raster image will not group the palette with the preceding image. Palettes written to a 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 a good idea to explicitly group each palette with the image to which it belongs. To find out more about assigning a palette to an image, see Chapter 6, *8-Bit Raster Images (DFR8 API)*.

EXAMPLE 1. **Writing a Palette**

In the following code examples, **DFPaddpal** is used to write a palette to an HDF file named "Example1.hdf".

C:

```
#include "hdf.h"

main( )
{
    uint8 palette_data[768];
    intn i;
    int32 status;

    /* 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. */
    status = DFPaddpal("Example1.hdf", (VOIDP)palette_data);

}
```

FORTRAN:

```
      PROGRAM WRITE PALETTE

       integer dpapal, status, i
       character palette_data(768)

C      Initialize the palette to greyscale.
       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 the HDF file.
       status = dpapal('Example1.hdf', palette_data)

       end
```

9.4.2. Specifying the Reference Number of a Palette: DFPwriteref

DFPwriteref specifies the reference number of the palette to be written on the next call to **DFPaddpal** or **DFPputpal**:

```
C:          status = DFPwriteref(filename, ref);

            status = DFPaddpal(filename, palette);

FORTRAN:    status = dpwref(filename, ref)

            status = dpapal(filename, palette)
```

DFPwriteref assigns the specified reference number to the next palette written to the file *file-name*. If the value of *ref* is the same as the reference number of an existing palette, the existing palette will be overwritten.

The parameters of **DFPwriteref** are further described in the following table.

TABLE 9C **DFPwriteref Parameter List**

Routine Name [Return Type] **(FOR-TRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	**FOR-TRAN-77**	
DFPwriteref [intn] **(dpwref)**	filename	char *	character*(*)	Name of the HDF file containing the palette.
	ref	uint16	integer	Reference number for the next call to DFPaddpal or DFPputpal.

9.5. Reading a Palette

The DFP programming model for reading a palette is similar to that for writing a palette - only the palette read call is required.

9.5.1. Reading a Palette: DFPgetpal

DFPgetpal is the only function required to read a palette. If the file is being opened for the first time, **DFPgetpal** returns the first palette in the file. Subsequent calls will return successive palettes in the file. In this way palettes are read in the same order in which they were written to the file.

To read a palette from an HDF file, the calling program must contain the following routines:

```
C:          status = DFPgetpal(filename, palette);

FORTRAN:    status = dpgpal(filename,palette)
```

DFPgetpal retrieves the next palette from the HDF file specified by *filename*. The space allocated for the palette is specified by palette and must be at least 768 bytes. When **DFPgetpal** is first called, it returns the first palette in the file. Subsequent calls to **DFPgetpal** will return successive palettes in the order in which they are stored in the file, including those stored via the DFR8 interface.

The parameters of **DFPgetpal** are defined in the following table.

TABLE 9D **DFPgetpal Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFPgetpal [intn] **(dpapal)**	filename	char *	character*(*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte buffer for the palette.

EXAMPLE 2. **Reading a Palette**

The following examples demonstrate the method used to read a palette from the "Example1.hdf" HDF file created in Example 1.

C:

```
#include "hdf.h"

main( )
{

    uint8 palette_data[768];
    intn status;

    /* Read the palette data from a file. */
    status = DFPgetpal("Example1.hdf", (VOIDP)palette_data);

}
```

FORTRAN:

```
     PROGRAM READ PALETTE

      integer dpgpal, status
      character palette_data(768)

C     Read the palette from the HDF file.
      status = dpgpal('Example1.hdf', palette_data)
```

```
            end
```

9.5.2. Reading a Palette with a Given Reference Number: DFPreadref

DFPreadref is used to access specific palettes stored in files containing multiple palettes. It is the optionally called before **DFPgetpal** to set the next palette to be accessed to be the specified palette. **DFPreadref** can be used in connection with vgroups, which identify their members by tag/reference number pair.

To access a specific palette, use the following calling sequence:

```
C:          true_false = DFPreadref(filename, ref);
            status = DFPgetpal(filename, palette);

FORTRAN:    true_false = dprref(filename, ref)
            status = dpgpal(filename, palette)
```

DFPreadref specifies the reference number for the next read operation performed on the HDF file *filename* to the reference number specified by *ref*.

Due to an oversight in the library, in very rare cases, a palette may not be seen by the DFP API, the application may need to use GR API to obtain it. Please refer to Appendix D, *Issue of Missing Palettes* of this document for a detailed description of the issue and for help in determining which functions to use.

The parameters of **DFPreadref** are further defined in the following table.

TABLE 9E **DFPreadref Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFPreadref [intn] **(dprref)**	filename	char *	character*(*)	Name of the HDF file.
	ref	uint16	integer	Reference number of the next palette to be read.

9.5.3. Specifying the Next Palette to be Accessed to be the First Palette: DFPrestart

DFPrestart causes the next **DFPgetpal** to read from the first palette in the file, rather than the palette following the one that was most recently read. **DFPrestart** has the following syntax:

```
C:          status = DFPrestart( );

FORTRAN:    status = dprest( )
```

9.6. Other Palette Routines

9.6.1. Querying the Number of Palettes in a File: DFPnpals

DFPnpals returns the total number palettes in a file and has the following syntax:

```
C:          num_of_pals = DFPnpals(filename);

FORTRAN:    num_of_pals = dpnpals(filename)
```

The parameter of DFPnpals is further defined in the following table.

TABLE 9F **DFPnpals Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFPnpals [intn] **(dpnpals)**	filename	char *	character*(*)	Name of the HDF file.

9.6.2. Obtaining the Reference Number of the Most Recently Accessed Palette: DFPlastref

DFPlastref returns the reference number most recently used in writing or reading a palette. This routine is used for attaching annotations to palettes and adding palettes to vgroups.

The following calling sequence uses **DFPlastref** to find the reference number of the palette most recently written to an HDF file:

```
C:         status = DFPaddpal(filename, palette, width, height, compress);
           lastref = DFPlastref( );

FORTRAN:   status = dpapal(filename, palette, width, height, compress)
           lastref = dplref( )
```

DFPputpal or **DFPgetpal** can be used in place of **DFPaddpal** with similar results.

9.7. Backward Compatibility Issues

As HDF has evolved, a variety of internal structures have been used to store palettes, with different tags used to represent them. To maintain backward compatibility with older versions of HDF, the palette interface supported by HDF version 4.0 recognizes palettes stored using all previously-used HDF tags. A detailed description of the tags and structures used to store palettes is in the *HDF Specifications and Developer's Guide v3.2* which can be found from the HDF web site at `http://www.hdfgroup.org/`.

CHAPTER 10 -- Annotations (AN API)

10.1. Chapter Overview

The HDF annotation interface, the AN interface, supports the storage of labels and descriptions to HDF files and the data objects they contain. This chapter explains the methods used to read and write file and data object annotations using the AN interface.

Note that the AN interface works with multiple files and supersedes the single-file annotations interface, the DFAN interface, described in Chapter 11, *Single-file Annotations (DFAN API)*. Further note that the AN interface can also read files written by DFAN interface routines.

10.2. The Annotation Data Model

When working with different data types, it is often convenient to identify the contents of a file by adding a short text description or ***annotation***. An annotation serves as the explanation for a file or data object, as in "`COLLECTED 12/14/90`" or "`BLACK HOLE SIMULATION`". The annotation can be as short as a name or as long as a portion of source code. For example, if the data originated as satellite data, the annotation might include the source of the data, pertinent environmental conditions, or other relevant information. In the case of a hypothetical black hole simulation, the annotation might contain source code for the program that produced the data.

HDF annotations are designed to accommodate a wide variety of information including titles, comments, variable names, parameters, formulas, and source code. In fact, HDF annotations can encompass any textual information regarding the collection, meaning, or intended use of the data.

Annotations can be attached to files or data objects, and are themselves data objects identifiable by a tag/reference number pair. Refer to Chapter 2, *HDF Fundamentals*, for a description of tag/reference number pairs.

10.2.1. Labels and Descriptions

Annotations come in two forms: ***labels*** and ***descriptions***. ***Labels*** are short annotations used for assigning things like titles or time stamps to a file or its data objects. Longer annotations are called ***descriptions*** and typically contain more extensive information, such as a source code module or mathematical formulas.

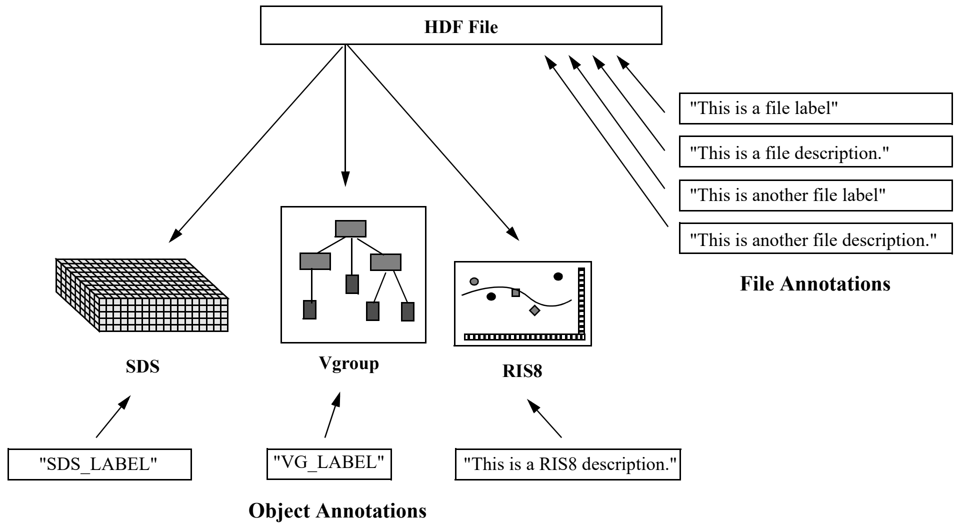
Labels are defined as a null-terminated string of characters. Descriptions may contain any sequence of ASCII characters.

In addition to the distinction made between labels and descriptions, HDF distinguishes between ***file annotations*** and ***object annotations***.

10.2.2. File Annotations

File annotations are assigned to a file to describe the origin, meaning, or intended use of its data. Any HDF file can be annotated with a label, description, or combination of both. (See Figure 10a) The number of labels or descriptions an HDF file may contain is limited to the maximum number of tag/reference number pairs. File annotations may be assigned in any order and at any time after a file is created.

FIGURE 10a **File and Object Annotations**


HDF File
"This is a file label"
"This is a file description."
"This is another file label"
"This is another file description."
File Annotations
SDS
Vgroup
RIS8
"SDS_LABEL"
"VG_LABEL"
"This is a RIS8 description."
Object Annotations

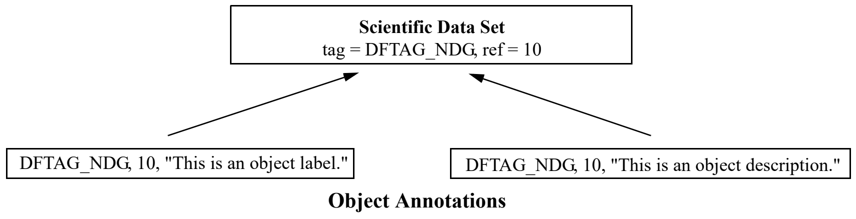
Although it is possible to use a file annotation to describe a data object in a file, this practice is not recommended. Each data object should be described by its own data object annotation as it is added to the file.

10.2.3. Object Annotations

Object annotations are assigned to individual data objects to explain their origin, meaning, or intended use. Because object annotations are assigned to individual objects, their use requires an understanding of HDF tags and reference numbers (see Chapter 2, *HDF Fundamentals*).

The annotation interface takes advantage of this identification scheme by including the object's tag/reference number pair with the text of the annotation. Consider a scientific data set identified by the tag `DFTAG_NDG` and the reference number `10`. (See Figure 10b) All object annotations assigned to this particular data set must be prefaced with the tag `DFTAG_NDG` followed by the reference number `10`.

FIGURE 10b **Object Annotations with Tag/Reference Number Pairs**


Scientific Data Set
tag = DFTAG_NDG, ref = 10
DFTAG_NDG, 10, "This is an object label."
DFTAG_NDG, 10, "This is an object description."
Object Annotations

10.2.4. Terminology

The following pairs of terms are used interchangeably in the following discussions: ***data object annotation*** and ***data annotation***; ***data object label*** and ***data label***; ***data object description*** and ***data description***.

10.3. The AN interface

The AN interface permits concurrent operations on a set of annotations that exist in more than one file rather than requiring the program to deal with the annotations on a file-by-file basis.

10.3.1. AN Library Routines

The C routine names of the AN interface are prefaced by the string "AN" and the FORTRAN-77 routine names are prefaced by "af". These routines are divided into the following categories:

- ***Access routines*** initialize and terminate access to the AN interface and the annotation.
- ***Read/write routines*** read and write file or object annotations.
- ***General inquiry routines*** return information about the annotations.

The AN routines are listed in Table 10A and are described in more detail in subsequent sections of this chapter.

TABLE 10A **AN Library Routines**

Category	Routine Names		Description
	C	FORTRAN-77	
Access	`ANstart`	`afstart`	Initializes the AN interface (Section "*Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate"*)
	`ANcreate`	`afcreate`	Creates a new data annotation (Section "*Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate"*)
	`ANcreatef`	`affcreate`	Creates a new file annotation (Section "*Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate"*)
	`ANselect`	`afselect`	Obtains an existing annotation (Section "*Selecting an Annotation: ANselect"*)
	`ANendaccess`	`afendaccess`	Terminates access to an annotation (Section "*Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate"*)
	`ANend`	`afend`	Terminates access to AN interface (Section "*Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate"*)
Read/write	`ANreadann`	`afreadeann`	Reads an annotation (Section "*Reading an Annotation: ANreadann"*)
	`ANwriteann`	`afwriteann`	Writes an annotation (Section "*Writing an Annotation: ANwriteann"*)
General Inquiry	`ANannlen`	`afannlen`	Returns the length of an annotation (Section "*Getting the Length of an Annotation: ANannlen"*)
	`ANannlist`	`afannlist`	Retrieves the annotation identifiers of an object (Section "*Obtaining the List of Specifically-typed Annotation Identifiers of a Data Object: ANannlist"*)
	`ANatype2tag`	`afatypetag`	Returns the annotation tag corresponding to an annotation type (Section "*Obtaining an Annotation Tag from a Specified Annotation Type: ANatype2tag"*)
	`ANfileinfo`	`affileinfo`	Retrieves the number of annotations of each type in a file (Section "*Obtaining the Number of Annotations: ANfileinfo"*)
	`ANnumann`	`afnumann`	Returns the number of annotations of the given type attached to an object (Section "*Obtaining the Number of Specifically-typed Annotations of a Data Object: ANnumann"*)
	`ANget_tagref`	`afgettagref`	Retrieves the tag/reference number pair of an annotation specified by its index (Section "*Obtaining the Tag/Reference Number Pair of the Specified Annotation Index and Type: ANget_tagref"*)
	`ANid2tagref`	`afidtagref`	Retrieves the tag/reference number pair of an annotation specified by its identifier (Section "*Obtaining the Tag/Reference Number Pair from a Specified Annotation Identifier: ANid2tagref"*)
	`ANtag2atype`	`aftagatype`	Returns the annotation type corresponding to an annotation tag (Section "*Obtaining an Annotation Type from a Specified Object Tag: ANtag2atype"*)
	`ANtagref2id`	`aftagrefid`	Returns the identifier of an annotation given its tag/reference number pair (Section "*Obtaining the Annotation Identifier from a Specified Tag/Reference Number Pair: ANtagref2id"*)

10.3.2. Type and Tag Definitions Used in the AN Interface

The AN interface uses the four general annotation types used in HDF: the data label, the data description, the file label and the file description. These annotation types correspondingly map to the `AN_DATA_LABEL` (or `0`), the `AN_DATA_DESC` (or `1`), the `AN_FILE_LABEL` (or `2`) and the `AN_FILE_DESC` (or `3`) definitions. Several routines in the AN interface require one of these type definitions to be passed in as an argument to designate the kind of annotation to be created or accessed.

10.3.3. Programming Model for the AN Interface

As with the GR and SD interfaces, the programming model for the AN interface allows several files to be open concurrently. The contents of these files can be operated on simultaneously as long as the calling program accurately keeps track of each interface. The file and object identifiers returned by the interface. Each object identifier and file identifier must be explicitly disposed of before the termination of the calling program.

The AN interface writes file labels, file descriptions, data object labels, and data object descriptions according to the following programming model:

1. Open the HDF file.
2. Initialize the AN interface.
3. Create a file annotation or a data annotation.
4. Perform the desired operations on the annotation.
5. Terminate access to the annotation.
6. Terminate access to the AN interface.
7. Close the HDF file.

To create a file or object annotation, the calling program must contain the following AN routine calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            an_id = ANstart(file_id);

            ann_id = ANcreatef(an_id, annot_type);
    OR      ann_id = ANcreate(an_id, obj_tag, obj_ref, annot_type);

            <Optional operations>

            status = ANendaccess(ann_id);
            status = ANend(an_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            an_id = afstart(file_id)

            ann_id = affcreate(an_id, annot_type)
    OR      ann_id = afcreate(an_id, obj_tag, obj_ref, annot_type)

            <Optional operations>

            status = afendaccess(ann_id)
            status = afend(an_id)
            status = hclose(file_id)
```

10.3.4. Accessing Files and Annotations: ANstart, ANcreatef, and ANcreate

An HDF file must be opened by **Hopen** before it can be accessed using the AN interface. **Hopen** is described in Chapter 2, *HDF Fundamentals*.

ANstart initializes the AN interface for subsequent AN interface operations. **ANstart** takes one argument, the file identifier, *file_id*, returned by **Hopen**, and returns an AN interface identifier, *an_id* or `FAIL` (or `-1`) upon unsuccessful completion.

ANcreatef creates a file label or file description. It takes two parameters: the AN interface identifier, *an_id*, returned by **ANstart**, and the type of the file annotation to be created, *annot_type*. The parameter *annot_type* must be set to either `AN_FILE_LABEL` (or `2`) or `AN_FILE_DESC` (or `3`). **ANcreatef** returns the file annotation identifier (*ann_id*) if successful, and `FAIL` (or `-1`) otherwise.

ANcreate creates a data label or data description. It takes four parameters: *an_id*, *obj_tag*, *obj_ref*, and *annot_type*. The parameter *an_id* is the AN interface identifier, returned by **ANstart**. The parameters *obj_tag* and *obj_ref* are the tag/reference number pair of the object the annotation will be assigned to. The parameter *annot_type* specifies the type of the data annotation. It must be set to either `AN_DATA_LABEL` (or `0`) or `AN_DATA_DESC` (or `1`). The annotation type definitions are defined in the header file "hdf.h".

ANcreate returns the data annotation identifier (*ann_id*) if successful and `FAIL` (or `-1`) otherwise. The parameters of **ANcreate**, **ANcreatef**, and **ANstart** are further defined in Table 10B.

10.3.5. Terminating Access to Annotations and Files: ANendaccess and ANend

ANendaccess terminates access to the annotation identified by the parameter *ann_id*, which is returned by **ANcreate** or **ANcreatef**. Any subsequent attempts to access this annotation identifier will result in a value of `FAIL` being returned. One **ANendaccess** must be called for every **ANcreate**, **ANcreatef**, or **ANattach**. Each **ANendaccess** returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`).

ANend terminates access to the AN interface identified by the parameter *an_id*, which is returned by **ANstart**. Any subsequent attempts to access the AN interface identifier or to use AN routines will result in a value of `FAIL` being returned.

ANend returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANendaccess** and **ANend** are defined in Table 10B.

The HDF file must be closed by **Hclose** after all calls to **ANend** have been properly made. **Hclose** is described in Chapter 2, *HDF Fundamentals*.

TABLE 10B **ANstart, ANcreate, ANcreatef, ANendaccess and ANend Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
ANstart [int32] **(afstart)**	file_id	int32	integer	File identifier
ANcreate [int32] **(afcreate)**	an_id	int32	integer	AN interface identifier
	obj_tag	uint16	integer	Tag of the object to be annotated
	obj_ref	uint16	integer	Reference number of the object to be annotated
	annot_type	ann_type	integer	Data annotation type
ANcreatef [int32] **(affcreate)**	an_id	int32	integer	AN interface identifier
	annot_type	ann_type	integer	File annotation type
ANendaccess [intn] **(afendaccess)**	ann_id	int32	integer	Annotation identifier
ANend [int32] **(afend)**	an_id	int32	integer	AN interface identifier

10.4. Writing an Annotation: ANwriteann

The AN programming model for writing an annotation is as follows:

1. Create a file annotation or a data annotation.
2. Write to the annotation.
3. Terminate access to the annotation.

To write a file or data annotation, the calling program must contain the following routine calls:

```
C:          file_id = Hopen(filename, file_access_mode, num_dds_block);
            an_id = ANstart(file_id);

            ann_id = ANcreatef(an_id, annot_type);
    OR      ann_id = ANcreate(an_id, obj_tag, obj_ref, annot_type);

            status = ANwriteann(ann_id, ann_text, ann_length);
            status = ANendaccess(ann_id);
            status = ANend(an_id);
            status = Hclose(file_id);

FORTRAN:    file_id = hopen(filename, file_access_mode, num_dds_block)
            an_id = afstart(file_id)

            ann_id = affcreate(an_id, annot_type)
    OR      ann_id = afcreate(an_id, obj_tag, obj_ref, annot_type)

            status = afwriteann(ann_id, ann_text, ann_length)
            status = afendaccess(ann_id)
            status = afend(an_id)
            status = hclose(file_id)
```

ANwriteann writes the annotation text given in the parameter *ann_text* to the annotation specified by *ann_id*. The parameter *ann_length* specifies the number of characters in the annotation text, not including the `NULL` character. If the annotation has already been written with text, **ANwrite-ann** will overwrite the current text.

ANwriteann returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANwriteann** are further defined in Table 10C.

TABLE 10C

ANwriteann Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
ANwriteann [int32] **(afwriteann)**	ann_id	int32	integer	Annotation identifier
	ann_text	char *	character*(*)	Text of the annotation
	ann_length	int32	integer	Number of characters in the annotation

EXAMPLE 1.

Creating File and Data Annotations

This example illustrates the use of **ANcreatef/affcreate** to create file annotations and **ANcreate/afcreate** to create data annotations.

In this example, the program creates an HDF file named "General_HDFobjects.hdf" then attaches to it two annotations, a file label and a file description. Within the HDF file, the program creates a vgroup named "AN Vgroup" and attaches to it two annotations, a data label and a data description. Refer to Chapter 5, *Vgroups (V API)*, for a discussion of the V interface routines used in this example.

Note that the names `AN_FILE_LABEL`, `AN_FILE_DESC`, `AN_DATA_LABEL`, and `AN_DATA_DESC` are defined by the library to specify the type of the annotation to be accessed.

C:

```
#include "hdf.h"

#define  FILE_NAME      "General_HDFobjects.hdf"
#define  VG_NAME        "AN Vgroup"
#define  FILE_LABEL_TXT "General HDF objects"
#define  FILE_DESC_TXT  "This is an HDF file that contains general HDF objects"
#define  DATA_LABEL_TXT "Common AN Vgroup"
#define  DATA_DESC_TXT  "This is a vgroup that is used to test data annota-
tions"

main( )
{
   /************************* Variable declaration **************************/

   intn   status_n;     /* returned status for functions returning an intn  */
   int32  status_32,    /* returned status for functions returning an int32 */
          file_id,      /* HDF file identifier */
          an_id,        /* AN interface identifier */
          file_label_id,  /* file label identifier */
          file_desc_id,   /* file description identifier */
          data_label_id,  /* data label identifier */
          data_desc_id,   /* data description identifier */
          vgroup_id;
   uint16 vgroup_tag, vgroup_ref;

   /********************** End of variable declaration **********************/
```

```
/*
* Create the HDF file.
*/
file_id = Hopen (FILE_NAME, DFACC_CREATE, 0);

/*
* Initialize the AN interface.
*/
an_id = ANstart(file_id);

/*
* Create the file label.
*/
file_label_id = ANcreatef(an_id, AN_FILE_LABEL);

/*
* Write the annotations to the file label.
*/
status_32 = ANwriteann(file_label_id, FILE_LABEL_TXT,
                       strlen (FILE_LABEL_TXT));

/*
* Create file description.
*/
file_desc_id = ANcreatef(an_id, AN_FILE_DESC);

/*
* Write the annotation to the file description.
*/
status_32 = ANwriteann(file_desc_id, FILE_DESC_TXT,
                       strlen (FILE_DESC_TXT));

/*
* Create a vgroup in the V interface.  Note that the vgroup's ref number
* is set to -1 for creating and the access mode is "w" for writing.
*/
status_n = Vstart(file_id);
vgroup_id = Vattach(file_id, -1, "w");
status_32 = Vsetname (vgroup_id, VG_NAME);

/*
* Obtain the tag and ref number of the vgroup for subsequent
* references.
*/
vgroup_tag = (uint16) VQuerytag (vgroup_id);
vgroup_ref = (uint16) VQueryref (vgroup_id);

/*
* Create the data label for the vgroup identified by its tag
* and ref number.
*/
data_label_id = ANcreate(an_id, vgroup_tag, vgroup_ref, AN_DATA_LABEL);

/*
* Write the annotation text to the data label.
*/
status_32 = ANwriteann(data_label_id, DATA_LABEL_TXT,
                       strlen(DATA_LABEL_TXT));

/*
* Create the data description for the vgroup identified by its tag
* and ref number.
```

```
     */
     data_desc_id = ANcreate(an_id, vgroup_tag, vgroup_ref, AN_DATA_DESC);

     /*
     * Write the annotation text to the data description.
     */
     status_32 = ANwriteann(data_desc_id, DATA_DESC_TXT, strlen(DATA_DESC_TXT));

     /*
     * Teminate access to the vgroup and to the V interface.
     */
     status_32 = Vdetach(vgroup_id);
     status_n = Vend(file_id);

     /*
     * Terminate access to each annotation explicitly.
     */
     status_n = ANendaccess(file_label_id);
     status_n = ANendaccess(file_desc_id);
     status_n = ANendaccess(data_label_id);
     status_n = ANendaccess(data_desc_id);

     /*
     * Terminate access to the AN interface and close the HDF file.
     */
     status_32 = ANend(an_id);
     status_n = Hclose(file_id);
}
```

FORTRAN:

```
      program create_annotation
      implicit none
C
C     Parameter declaration
C
      character*22 FILE_NAME
      character*9  VG_NAME
      character*19 FILE_LABEL_TXT
      character*53 FILE_DESC_TXT
      character*16 DATA_LABEL_TXT
      character*54 DATA_DESC_TXT
C
      parameter (FILE_NAME      = ’General_HDFobjects.hdf’,
     +           VG_NAME        = ’AN Vgroup’,
     +           FILE_LABEL_TXT = ’General HDF objects’,
     +           DATA_LABEL_TXT = ’Common AN Vgroup’,
     +           FILE_DESC_TXT  =
     + ’This is an HDF file that contains general HDF objects’,
     +           DATA_DESC_TXT  =
     + ’This is a vgroup that is used to test data annotations’)
      integer DFACC_CREATE
      parameter (DFACC_CREATE = 4)
      integer AN_FILE_LABEL, AN_FILE_DESC, AN_DATA_LABEL, AN_DATA_DESC
      parameter (AN_FILE_LABEL = 2,
     +           AN_FILE_DESC  = 3,
     +           AN_DATA_LABEL = 0,
     +           AN_DATA_DESC  = 1)
C
C     Function declaration
C
      integer hopen, hclose
      integer afstart, affcreate, afwriteann, afcreate,
     +        afendaccess, afend
      integer vfstart, vfatch, vfsnam, vqref, vqtag, vfdtch, vfend

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, an_id
      integer file_label_id, file_desc_id
      integer data_label_id, data_desc_id
      integer vgroup_id, vgroup_tag, vgroup_ref
C
C**** End of variable declaration ************************************
C
C
C     Create the HDF file.
C
      file_id = hopen(FILE_NAME, DFACC_CREATE, 0)
C
C     Initialize the AN interface.
C
      an_id = afstart(file_id)
C
C     Create the file label.
C
      file_label_id = affcreate(an_id, AN_FILE_LABEL)
C
C     Write the annotation to the file label.
C
      status = afwriteann(file_label_id, FILE_LABEL_TXT,
     +                    len(FILE_LABEL_TXT))
C
C     Create file description.
C
      file_desc_id = affcreate(an_id, AN_FILE_DESC)
C
C     Write the annotation to the file description.
C
      status = afwriteann(file_desc_id, FILE_DESC_TXT,
     +                    len(FILE_DESC_TXT))
C
C     Create a vgroup in the file. Note that the vgroup’s ref number is
C     set to -1 for creating and the access mode is ’w’ for writing.
C
      status    = vfstart(file_id)
      vgroup_id = vfatch(file_id, -1, ’w’)
      status    = vfsnam(vgroup_id, VG_NAME)
C
C     Obtain the tag and reference number of the vgroup for subsequent
C     references.
C
      vgroup_ref = vqref(vgroup_id)
      vgroup_tag = vqtag(vgroup_id)
C
C     Create the data label for the vgroup identified by its tag and ref
C     number.
C
      data_label_id = afcreate(an_id, vgroup_tag, vgroup_ref,
     +                          AN_DATA_LABEL)
C
C     Write the annotation text to the data label.
C
      status = afwriteann(data_label_id, DATA_LABEL_TXT,
     +                   len(DATA_LABEL_TXT))

C
C     Create the data description for the vgroup identified by its tag and ref.
C
      data_desc_id = afcreate(an_id, vgroup_tag, vgroup_ref,
     +                        AN_DATA_DESC)
C
C     Write the annotation text to the data description.
C
      status = afwriteann(data_desc_id, DATA_DESC_TXT,
     +                   len(DATA_DESC_TXT))
C
C     Terminate access to the vgroup and to the V interface.
C
      status = vfdtch(vgroup_id)
      status = vfend(file_id)
C
C     Terminate access to each annotation explicitly.
C
      status = afendaccess(file_label_id)
      status = afendaccess(file_desc_id)
      status = afendaccess(data_label_id)
      status = afendaccess(data_desc_id)
C
C     Terminate access to the AN interface and close the HDF file.
C
      status = afend(an_id)
      status = hclose(file_id)
      end
```

10.5. Reading Annotations Using the AN Interface

Reading an annotation is done by first selecting the desired annotation of the appropriate type using ANselect, then reading the annotation text using ANreadann. These two routines are described in this section.

10.5.1. Selecting an Annotation: ANselect

ANselect obtains the identifier of the annotation specified by its index, *index*, and by its annotation type, *annot_type*. The syntax for **ANselect** is as follows:

```
C:          ann_id = ANselect(an_id, index, annot_type);

FORTRAN:    ann_id = afselect(an_id, index, annot_type)
```

The parameter *index* is a nonnegative integer and is less than the total number of annotations of type *annot_type* in the file. Use **ANfileinfo**, described in Section "*Obtaining the Number of Annotations: ANfileinfo*", to obtain the total number of annotations of type *annot_type* in the file.

Possible valid values of *annot_type* are `AN_DATA_LABEL` (or `0`) for a data label, `AN_DATA_DESC` (or `1`) for a data description, `AN_FILE_LABEL` (or `2`) for a file label, and `AN_FILE_DESC` (or `3`) for a file description.

ANselect returns an annotation identifier or `FAIL` (or `-1`) upon unsuccessful completion. The parameters of **ANselect** are further described in Table 10D.

10.5.2. Reading an Annotation: ANreadann

ANreadann reads the annotation specified by the parameter *ann_id* and stores the annotation text in the parameter *ann_buf*. The syntax for **ANreadann** is as follows

```
C:          status = ANreadann(ann_id, ann_buf, ann_length);

FORTRAN:    status = afreadann(ann_id, ann_buf, ann_length)
```

The parameter *ann_length* specifies the size of the buffer *ann_buf*. If the length of the file or data label to be read is greater than or equal to *ann_length*, the label will be truncated to *ann_length* - 1 characters. If the length of the file or data description is greater than *ann_length*, the description will be truncated to *ann_length* characters. The HDF library adds a `NULL` character to the retrieved label but not to the description. The user must add a `NULL` character to the retrieved description if the C library string functions are to operate on this description.

ANreadann returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANreadann** are further described in Table 10D.

TABLE 10D **ANselect and ANreadann Parameter Lists**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
ANselect [int32] **(afselect)**	an_id	int32	integer	AN interface identifier
	index	int32	integer	Index of the annotation
	annot_type	ann_type	integer	Type of the annotation
ANreadann [int32] **(afreadann)**	ann_id	int32	integer	Annotation identifier
	ann_buf	char *	character*(*)	Buffer for the returned annotation text
	ann_length	int32	integer	Number of characters to be retrieved from the annotation

EXAMPLE 2. **Reading File and Data Annotations**

This example illustrates the use of **ANfileinfo/affileinfo** to get the number of data and file annotations in the file, **ANselect/afselect** to get an annotation, **ANannlen/afannlen** to get the length of the annotation, and **ANreadann/afreadann** to read the contents of the annotation.

In this example, the program reads some of the annotations created in the file "General_HDFobjects.hdf" by Example 1. The program first gets the information on the annotations in the file so that the number of existing annotations of each kind is available prior to reading. The program then gets the length of each annotation and allocates sufficient space for the contents of the annotation to be read. For the simplicity of this example, only the data labels are read. Any other annotations can be read by adding the for loop with appropriate values as noted below.

This example uses the **ANfileinfo/affileinfo** routine to get annotation information. This routine is described in the Section *"Obtaining the Number of Annotations: ANfileinfo"*.

C:

```
#include "hdf.h"

#define  FILE_NAME   "General_HDFobjects.hdf"

main( )
```

```
{
   /************************* Variable declaration *************************/

   intn  status_n;      /* returned status for functions returning an intn  */
   int32 status_32,     /* returned status for functions returning an int32 */
         file_id,       /* HDF file identifier */
         an_id,         /* AN interface identifier */
         ann_id,        /* an annotation identifier */
         index,         /* position of an annotation in all of the same type*/
         ann_length,    /* length of the text in an annotation */
         n_file_labels, n_file_descs, n_data_labels, n_data_descs;
   char *ann_buf;       /* buffer to hold the read annotation */

   /********************* End of variable declaration *********************/

   /*
   * Open the HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the AN interface.
   */
   an_id = ANstart (file_id);

   /*
   * Get the annotation information, e.g., the numbers of file labels, file
   * descriptions, data labels, and data descriptions.
   */
   status_n = ANfileinfo (an_id, &n_file_labels, &n_file_descs,
                          &n_data_labels, &n_data_descs);

   /*
   * Get the data labels.  Note that this for loop can be used to
   * obtain the contents of each kind of annotation with the appropriate
   * number of annotations and the type of annotation, i.e., replace
   * n_data_labels with n_file_labels, n_file_descs, or n_data_descs, and
   * AN_DATA_LABEL with AN_FILE_LABEL, AN_FILE_DESC, or AN_DATA_DESC,
   * respectively.
   */
   for (index = 0; index < n_data_labels; index++)
   {
      /*
      * Get the identifier of the current data label.
      */
      ann_id = ANselect (an_id, index, AN_DATA_LABEL);

      /*
      * Get the length of the data label.
      */
      ann_length = ANannlen (ann_id);

      /*
      * Allocate space for the buffer to hold the data label text.
      */
      ann_buf = malloc ((ann_length+1) * sizeof (char));

      /*
      * Read and display the data label.  Note that the size of the buffer,
      * i.e., the third parameter, is 1 character more than the length of
      * the data label; that is for the null character.  It is not the case
      * when a description is retrieved because the description does not
      * necessarily end with a null character.
```

```
            *
            */
            status_32 = ANreadann (ann_id, ann_buf, ann_length+1);
            printf ("Data label index: %d\n", index);
            printf ("Data label contents: %s\n", ann_buf);

            /*
            * Terminate access to the current data label.
            */
            status_n = ANendaccess (ann_id);

            /*
            * Free the space allocated for the annotation buffer.
            */
            free (ann_buf);
         }

         /*
         * Terminate access to the AN interface and close the HDF file.
         */
         status_32 = ANend (an_id);
         status_n = Hclose (file_id);
      }
```

FORTRAN:

```
      program  read_annotation
      implicit none
C
C     Parameter declaration
C
      character*22 FILE_NAME
C
      parameter (FILE_NAME = 'General_HDFobjects.hdf')
      integer    DFACC_READ
      parameter (DFACC_READ = 1)
      integer    AN_DATA_LABEL
      parameter (AN_DATA_LABEL = 0)
C
C     Function declaration
C
      integer hopen, hclose
      integer afstart, affileinfo, afselect, afannlen, afreadann,
     +        afendaccess, afend
C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, an_id, ann_id
      integer index, ann_length
      integer n_file_labels, n_file_descs, n_data_labels, n_data_descs
      character*256 ann_buf
C
C**** End of variable declaration ************************************
C
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the AN interface.
C
      an_id = afstart(file_id)
```

```
C
C     Get the annotation information, i.e., the number of file labels,
C     file descriptions, data labels, and data descriptions.
C
      status = affileinfo(an_id, n_file_labels, n_file_descs,
     +                    n_data_labels, n_data_descs)
C
C     Get the data labels. Note that this DO loop can be used to obtain
C     the contents of each kind of annotation with the appropriate number
C     of annotations and the type of annotation, i.e., replace
C     n_data_labels with n_file_labels, n_files_descs, or n_data_descs, and
C     AN_DATA_LABEL with AN_FILE_LABEL, AN_FILE_DESC, or AN_DATA_DESC,
C     respectively.
C
      do 10 index = 0, n_data_labels-1
C
C     Get the identifier of the current data label.
C
      ann_id = afselect(an_id, index, AN_DATA_LABEL)
C
C     Get the length of the data label.
C
      ann_length = afannlen(ann_id)
C
C     Read and display the data label. The data label is read into buffer
C     ann_buf. One has to make sure that ann_buf has sufficient size to hold
C     the data label. Also note, that the third argument to afreadann is
C     1 greater that the actual length of the data label (see comment to
C     C example).
C
      status = afreadann(ann_id, ann_buf, ann_length+1)
      write(*,*) 'Data label index: ', index
      write(*,*) 'Data label contents: ', ann_buf(1:ann_length)
10    continue
C
C     Terminate access to the current data label.
C
      status = afendaccess(ann_id)
C
C     Terminate access to the AN interface and close the HDF file.
C
      status = afend(an_id)
      status = hclose(file_id)
      end
```

10.6. Obtaining Annotation Information Using the AN Interface

The HDF library provides various AN routines to obtain annotation information for the purpose of locating either a particular annotation or a set of annotations that correspond to a set of search criteria. The following sections describe these AN routines.

10.6.1. Obtaining the Number of Annotations: ANfileinfo

ANfileinfo retrieves the total number of file labels, file descriptions, data labels, and data descriptions in the file identified by the parameter *an_id*. The syntax for **ANfileinfo** is as follows:

```
C:        status = ANfileinfo(an_id, &n_file_labels, &n_file_descs,
                              &n_data_labels, &n_data_descs);
```

```
FORTRAN:   status = affileinfo(an_id, n_file_labels, n_file_descs, n_data_la-
                          bels, n_data_descs)
```

The retrieved information will be stored in the parameters *n_file_labels*, *n_file_descs*, *n_data_labels*, and *n_data_descs*, respectively. They can also be used as loop boundaries.

ANfileinfo returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANfileinfo** are further described in Table 10E.

10.6.2. Getting the Length of an Annotation: ANannlen

ANannlen returns either the length of the annotation, identified by the parameter *ann_id*, or `FAIL` (or `-1`) upon unsuccessful completion. The syntax for **ANannlen** is as follows:

```
C:         ann_len = ANannlen(ann_id);

FORTRAN:   ann_len = afannlen(ann_id)
```

The parameters of **ANannlen** are further described in Table 10E.

TABLE 10E **ANfileinfo and ANannlen Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
ANfileinfo [intn] **(affileinfo)**	an_id	int32	integer	AN interface identifier
	n_file_labels	int32 *	integer	Number of file labels in the file
	n_file_descs	int32 *	integer	Number of file descriptions in the file
	n_data_labels	int32 *	integer	Number of data labels in the file
	n_data_descs	int32 *	integer	Number of data descriptions in the file
ANannlen [int32] **(afannlen)**	ann_id	int32	integer	Annotation identifier

10.6.3. Obtaining the Number of Specifically-typed Annotations of a Data Object: ANnumann

ANnumann returns the total number of annotations that are of type *annot_type* and that are attached to the object identified by its tag, *obj_tag*, and reference number, *obj_ref*. The syntax for **ANnumann** is as follows:

```
C:         ann_num = ANnumann(an_id, annot_type, obj_tag, obj_ref);

FORTRAN:   ann_num = afnumann(an_id, annot_type, obj_tag, obj_ref)
```

As this routine is implemented only to obtain the total number of data annotations and not file annotations, the valid values of *annot_type* are `AN_DATA_LABEL` (or `0`) and `AN_DATA_DESC` (or `1`). To obtain the total number of file annotations or all data annotations, use **ANfileinfo**.

ANnumann returns the total number of qualified annotations or `FAIL` (or `-1`). The parameters of **ANnumann** are further described in Table 10F.

10.6.4. Obtaining the List of Specifically-typed Annotation Identifiers of a

Data Object: ANannlist

ANannlist retrieves the annotation identifiers for all of the annotations that are of type *annot_type* and belong to the object identified by its tag, *obj_tag*, and its reference number, *obj_ref*. The syntax for **ANannlist** is as follows:

```
C:          status = ANannlist(an_id, annot_type, obj_tag, obj_ref, ann_list);

FORTRAN:    status = afselect(an_id, annot_type, obj_tag, obj_ref, ann_list)
```

The identifiers of the retrieved annotations are stored in the parameter *ann_list*. The routine **ANnumann** can be used to obtain the number of annotations to be retrieved for dynamic memory allocation.

ANannlist returns the number of identifiers found, if succesful, or `FAIL` (or `-1`). The parameters of **ANannlist** are further described in Table 10F.

TABLE 10F **ANnumann and ANannlist Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FOR-TRAN-77	
ANnumann [intn] **(afnumann)**	an_id	int32	integer	AN interface identifier
	annot_type	ann_type	integer	Type of the annotation
	obj_tag	uint16	integer	Tag of the object the annotation is attached to
	obj_ref	uint16	integer	Reference number of the object the annotation is attached to
ANannlist [intn] **(afannlist)**	an_id	int32	integer	AN interface identifier
	annot_type	ann_type	integer	Type of the annotation
	obj_tag	uint16	integer	Tag of the object the annotation is attached to
	obj_ref	uint16	integer	Reference number of the object the annotation is attached to
	ann_list	int32 *	integer (*)	Buffer for returned annotation identifiers that match the search criteria

10.6.5. Obtaining the Tag/Reference Number Pair of the Specified Annotation Index and Type: ANget_tagref

ANget_tagref retrieves the tag and reference number of the annotation identified by its index, specified by the parameter *index*, and by the annotation type, specified by the parameter *annot_-type*. The syntax for **ANget_tagref** is as follows:

```
C:          status = ANget_tagref(an_id, index, annot_type, &ann_tag, &ann_ref);

FORTRAN:    status = afgettagref(an_id, index, annot_type, ann_tag, ann_ref)
```

The tag is stored in the parameter *ann_tag* and the reference number is stored in the parameter *ann_ref*. The parameter *index* is a nonnegative value and is less than the total number of annotations of type *annot_type* in the file. Use **ANfileinfo** to obtain the total number of annotations of type *annot_type* in the file.

The value of *annot_type* can be either `AN_DATA_LABEL` (or `0`), `AN_DATA_DESC` (or `1`), `AN_FILE_LA-BEL` (or `2`), or `AN_FILE_DESC` (or `3`).

ANget_tagref returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANget_tagref** are further described in Table 10G.

10.6.6. Obtaining the Tag/Reference Number Pair from a Specified Annotation Identifier: ANid2tagref

ANid2tagref retrieves the tag/reference number pair of the annotation identified by the parameter *ann_id*. The syntax for **ANid2tagref** is as follows:

```
C:          status = ANid2tagref(ann_id, &ann_tag, &ann_ref);

FORTRAN:    status = afidtagref(ann_id, ann_tag, ann_ref)
```

ANid2tagref stores the retrieved tag and reference number into the parameters *ann_tag* and *ann_ref*. Possible values returned in *ann_tag* are `DFTAG_DIL` (or `104`) for a data label, `DFTAG_DIA` (or `105`) for a data description, `DFTAG_FID` (or `100`) for a file label, and `DFTAG_FD` (or `101`) for a file description.

ANid2tagref returns either `SUCCEED` (or `0`) or `FAIL` (or `-1`). The parameters of **ANid2tagref** are further described in Table 10G.

10.6.7. Obtaining the Annotation Identifier from a Specified Tag/Reference Number Pair: ANtagref2id

ANtagref2id routine returns the identifier of the annotation that is specified by its tag/reference number pair or `FAIL` (or `-1`). The syntax for **ANtagref2id** is as follows:

```
C:          ann_id = ANtagref2id(an_id, ann_tag, ann_ref);

FORTRAN:    ann_id = aftagrefid(an_id, ann_tag, ann_ref)
```

The parameters of **ANtagref2id** are further described in Table 10G.

10.6.8. Obtaining an Annotation Tag from a Specified Annotation Type: ANatype2tag

ANatype2tag returns the tag that corresponds to the annotation type specified by the parameter *annot_type* if successful, or `DFTAG_NULL` (or `0`) otherwise. The syntax for **ANatype2tag** is as follows:

```
C:          ann_tag = ANatype2tag(annot_type);

FORTRAN:    ann_tag = afatypetag(annot_type)
```

The following table lists the valid values of *annot_type* in the left column and the corresponding values for the returned annotation tag on the right.

Annotation Type	Annotation Tag
AN_DATA_LABEL (or 0)	DFTAG_DIL (or 104)
AN_DATA_DESC (or 1)	DFTAG_DIA (or 105)
AN_FILE_LABEL (or 2)	DFTAG_FID (or 100)
AN_FILE_DESC (or 3)	DFTAG_FD (or 101)

The parameters of **ANatype2tag** are further described in Table 10G.

10.6.9. Obtaining an Annotation Type from a Specified Object Tag: ANtag2atype

ANtag2atype returns the annotation type corresponding to the annotation tag *ann_tag* if successful, or `AN_UNDEF` (or `-1`) otherwise. The syntax for **ANtag2atype** is as follows:

```
C:          annot_type = ANtag2atype(ann_tag);

FORTRAN:    annot_type = aftagatype(ann_tag)
```

The following table lists the valid values of *ann_tag* in the left column and the corresponding values of the returned annotation type in the right column.

Annotation Tag	Annotation Type
`DFTAG_DIL` (or `104`)	`AN_DATA_LABEL` (or `0`)
`DFTAG_DIA` (or `105`)	`AN_DATA_DESC` (or `1`)
`DFTAG_FID` (or `100`)	`AN_FILE_LABEL` (or `2`)
`DFTAG_FD` (or `101`)	`AN_FILE_DESC` (or `3`)

The parameters of **ANtag2atype** are further described in Table 10G.

TABLE 10G **ANget_tagref, ANid2tagref, ANtagref2id, ANatype2tag, and ANtag2atype Parameter Lists**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
ANget_tagref [int32] **(afgettagref)**	an_id	int32	integer	AN interface identifier
	ann_index	int32	integer	Index of the annotation
	annot_type	ann_type	integer	Annotation type of the annotation
	ann_tag	uint16 *	integer	Tag of the annotation
	ann_ref	uint16 *	integer	Reference number of the annotation
ANid2tagref [int32] **(afidtagref)**	ann_id	int32	integer	Identifier of the annotation
	ann_tag	uint16 *	integer	Tag of the annotation
	ann_ref	uint16 *	integer	Reference number of the annotation
ANtagref2id [int32] **(aftagrefid)**	an_id	int32	integer	AN interface identifier
	ann_tag	uint16	integer	Tag of the annotation
	ann_ref	uint16	integer	Reference number of the annotation
ANatype2tag [uint16] **(afatypetag)**	annot_type	ann_type	integer	Annotation type
ANtag2atype [ann_type] **(aftagatype)**	ann_tag	uint16	integer	Annotation tag

EXAMPLE 3. **Obtaining Annotation Information**

This example illustrates the use of **ANnumann/afnumann** to obtain the number of annotations of an object, **ANannlist/afannlist** to obtain the list of annotation identifiers, and **ANid2tagref/afid-**

tagref, **ANatype2tag/afatypetag**, and **ANtag2atype/aftagatype** to perform some identifier conversions.

In this example, the program locates the vgroup named "AN Vgroup" that was created in the file "General_HDFobjects.hdf" by Example 1. The program then gets the number of data descriptions that this vgroup has and the list of their identifiers. If there are any identifers in the list, the program displays the corresponding reference numbers. Finally, the program makes two simple conversions, from an annotation type to a tag and from a tag to an annotation type, and displays the results.

C:

```
#include "hdf.h"

#define  FILE_NAME   "General_HDFobjects.hdf"
#define  VG_NAME     "AN Vgroup"

main( )
{
   /************************* Variable declaration *************************/

   intn   status_n;      /* returned status for functions returning an intn */
   int32  status_32,     /* returned status for functions returning an int32*/
          file_id, an_id, ann_id,
          n_annots,      /* number of annotations */
         *ann_list,      /* list of annotation identifiers */
          vgroup_ref,    /* reference number of the vgroup */
          index;         /* index of an annotation in the annotation list */
          ann_type annot_type = AN_DATA_DESC;   /* annotation to be obtained*/
   uint16 ann_tag, ann_ref,              /* tag/ref number of an annotation */
          vgroup_tag = DFTAG_VG;         /* tag of the vgroup */

   /********************* End of variable declaration *********************/

   /*
   * Create the HDF file.
   */
   file_id = Hopen (FILE_NAME, DFACC_READ, 0);

   /*
   * Initialize the V interface.
   */
   status_n = Vstart (file_id);

   /*
   * Get the vgroup named VG_NAME.
   */
   vgroup_ref = Vfind  (file_id, VG_NAME);

   /*
   * Initialize the AN interface and obtain an interface id.
   */
   an_id = ANstart (file_id);

   /*
   * Get the number of object descriptions.  Note that, since ANnumann takes
   * the tag and reference number as being of type unit16, vgroup_ref must be
   * safely cast to uint16 by checking for FAIL value first.
   */
   if (vgroup_ref != FAIL)
   {
      n_annots = ANnumann (an_id, annot_type, vgroup_tag, (uint16)vgroup_ref);
      /*
      * Allocate space to hold the annotation identifiers.
      */
      ann_list = malloc (n_annots * sizeof (int32));

      /*
      * Get the list of identifiers of the annotations attached to the
      * vgroup and of type annot_type.
      */
      n_annots = ANannlist (an_id, annot_type, vgroup_tag, (uint16)vgroup_ref,
                            ann_list);

      /*
      * Get each annotation identifier from the list then display the
      * tag/ref number pair of the corresponding annotation.
      */
      printf ("List of annotations of type AN_DATA_DESC:\n");
      for (index = 0; index < n_annots; index++)
      {
        /*
         * Get and display the ref number of the annotation from
         * its identifier.
         */
         status_32 = ANid2tagref (ann_list[index], &ann_tag, &ann_ref);
         printf ("Annotation index %d: tag = %s\nreference number= %d\n",
           index, ann_tag == DFTAG_DIA ? "DFTAG_DIA (data description)":
           "Incorrect", ann_ref);
      } /* for */
   } /* for */

   /*
   * Get and display an annotation type from an annotation tag.
   */
   annot_type = ANtag2atype (DFTAG_FID);
   printf ("\nAnnotation type of DFTAG_FID (file label) is %s\n",
            annot_type == AN_FILE_LABEL ? "AN_FILE_LABEL":"Incorrect");

   /*
   * Get and display an annotation tag from an annotation type.
   */
   ann_tag = ANatype2tag (AN_DATA_LABEL);
   printf ("\nAnnotation tag of AN_DATA_LABEL is %s\n",
            ann_tag == DFTAG_DIL ? "DFTAG_DIL (data label)":"Incorrect");

   /*
   * Terminate access to the AN interface and close the HDF file.
   */
   status_32 = ANend (an_id);
   status_n = Hclose (file_id);

   /*
   * Free the space allocated for the annotation identifier list.
   */
   free (ann_list);
}
```

FORTRAN:

```
      program annotation_info
      implicit none
C
C     Parameter declaration
C
```

```
      character*22 FILE_NAME
      character*9  VG_NAME
C
      parameter (FILE_NAME      = 'General_HDFobjects.hdf',
     +           VG_NAME        = 'AN Vgroup')
      integer    DFACC_READ
      parameter (DFACC_READ = 1)
      integer AN_FILE_LABEL, AN_DATA_LABEL, AN_DATA_DESC
      parameter (AN_FILE_LABEL = 2,
     +           AN_DATA_LABEL = 0,
     +           AN_DATA_DESC  = 1)
      integer DFTAG_DIA, DFTAG_FID, DFTAG_DIL
      parameter (DFTAG_DIA = 105,
     +           DFTAG_FID = 100,
     +           DFTAG_DIL = 104)
      integer DFTAG_VG
      parameter (DFTAG_VG = 1965)
C
C     Function declaration
C
      integer hopen, hclose
      integer afstart, afnumann, afannlist, afidtagref, aftagatype,
     +        afatypetag, afend
      integer vfstart, vfind

C
C**** Variable declaration *******************************************
C
      integer status
      integer file_id, an_id
      integer n_annots, ann_index, annot_type, ann_tag, ann_ref
      integer ann_list(10)
      integer vgroup_tag, vgroup_ref
C
C**** End of variable declaration ***********************************
C
      annot_type = AN_DATA_DESC
      vgroup_tag = DFTAG_VG
C
C     Open the HDF file for reading.
C
      file_id = hopen(FILE_NAME, DFACC_READ, 0)
C
C     Initialize the V interface.
C
      status = vfstart(file_id)
C
C     Get the group named VG_NAME.
C
      vgroup_ref = vfind(file_id, VG_NAME)
C
C     Initialize the AN interface.
C
      an_id = afstart(file_id)

C
C     Get the number of object descriptions.
C
      if (vgroup_ref .eq. -1) goto 100
      n_annots = afnumann(an_id, annot_type, vgroup_tag, vgroup_ref)
C
C     Get the list of identifiers of the annotations attached to the
C     vgroup and of type annot_type. Identifiers are read into ann_list
```

```
C     buffer. One has to make sure that ann_list has the size big enough
C     to hold the list of identifiers.
C
      n_annots = afannlist(an_id, annot_type, vgroup_tag, vgroup_ref,
     +                     ann_list)
C
C     Get each annotation identifier from the list then display the
C     tag/ref number pair of the corresponding annotation.
C
      write(*,*) 'List of annotations of type AN_DATA_DESC'
      do 10 ann_index = 0, n_annots - 1
C
C     Get and display the ref number of the annotation from its
C     identifier.
C
      status = afidtagref(ann_list(ann_index+1), ann_tag, ann_ref)
      write(*,*) 'Annotation index: ', ann_index
      if (ann_tag .eq. DFTAG_DIA) then
          write(*,*) 'tag = DFTAG_DIA (data description)'
      else
          write(*,*) ' tag = Incorrect'
      endif
      write(*,*) 'reference number = ', ann_ref
10    continue
C
C     Get and display an annotation type from an annotation tag.
C
      annot_type = aftagatype(DFTAG_FID)
      if (annot_type .eq. AN_FILE_LABEL) then
         write(*,*) 'Annotation type of DFTAG_FID (file label) is ',
     +               'AN_FILE_LABEL '
      else
         write(*,*) 'Annotation type of DFTAG_FID (file label) is ',
     +               'Incorrect'
      endif
C
C     Get and display an annotation tag from an annotation type.
C
      ann_tag = afatypetag(AN_DATA_LABEL)
      if (ann_tag .eq. DFTAG_DIL ) then
         write(*,*) 'Annotation tag of AN_DATA_LABEL is ',
     +               'DFTAG_DIL (data label)'
      else
         write(*,*) 'Annotation type of DFTAG_FID (file label) is ',
     +               'Incorrect'
      endif
C
C     Terminate access to the AN interface and close the HDF file.
C
100   continue
      status = afend(an_id)
      status = hclose(file_id)
      end
```

CHAPTER 11 -- Single-file Annotations (DFAN API)

11.1. Chapter Overview

The original HDF annotation tools were the single-file tools that constitute the DFAN interface. These tools, which are used to read and write file and data object annotations, are described in this chapter.

Note that there is a multifile annotations interface, called the AN interface, for dealing with annotations.

Note: The AN interface supersedes the DFAN interface and is described in Chapter 10, *Annotations (AN API)*.

11.2. The Single-file Annotation Interface

The functions and routines that comprise the single-file annotation interface have names that begin with the string "DFAN" in C; the equivalent FORTRAN-77 routine names are prefaced by "da". This interface is the older annotation interface and only supports annotation access within one particular HDF file. It doesn't support the concept of an annotation identifier used in the newer multifile interface. Therefore, annotations created with the multifile interface cannot be accessed or manipulated with DFAN interface functions.

11.2.1. DFAN Library Routines

These functions are divided into the following categories:

- ***Write routines*** assign a file or object annotation.
- ***Read routines*** retrieve a file or object annotation.
- ***General inquiry routines*** return a list of all labels and reference numbers.
- ***Maintenance routine*** performs cleanup services.

The DFAN interface routines are listed in the following table and are discussed in the subsequent sections of this document.

TABLE 11A **DFAN Library Routines**

Purpose	Functions		Description
	C	FORTRAN-77	
Write	`DFANaddfds`	`daafds`	Assigns a file description to a specific file
	`DFANaddfid`	`daafid`	Assigns a file label to a specific file
	`DFANputdesc`	`dapdesc`	Assigns an object description to a specific data object
	`DFANputlabel`	`daplab`	Assigns an object label to a specific data object
Read	`DFANgetdesc`	`dagdesc`	Reads the text of an object description
	`DFANgetdes-clen`	`dagdlen`	Returns the length of an object description
	`DFANgetfds`	`dagfds`	Reads the text of a file description
	`DFANgetfdslen`	`dagfdsl`	Returns the length of a file description
	`DFANgetfid`	`dagfid`	Reads the text of a file label
	`DFANgetfidlen`	`dagfidl`	Returns the length of a file label
	`DFANgetlabel`	`daglab`	Reads the text of an object label
	`DFANgetlablen`	`dagllen`	Returns the length of an object label
General Inquiry	`DFANlablist`	`dallist`	Gets a list of all the labels in a file for a particular tag
	`DFANlastref`	`dalref`	Returns the reference number of the last annotation accessed
Maintenance	`DFANclear`	`None`	Clears the internal tables and structures used by the DFAN interface

11.2.2. Tags in the Annotation Interface

Table 11B lists the annotation tags defined in HDF versions 2.0, 3.0, and 4.0. Newly-defined tag names in each version are bolded. For a more complete list of tags, refer to the *HDF Specifications and Developer's Guide v3.2* from the HDF web site at `http://www.hdfgroup.org/`.

TABLE 11B **List of Annotation Interface Tags in HDF Versions 2.0, 3.0 and 4.0**

Interface	Data Object	Tag Name		
		v2.0	v3.0	v4.0
DFR8	Raster Image: 8-bit (uncompressed)	`DFTAG_RI8`	`DFTAG_RI`	`DFTAG_RI`
	Compressed Image: 8-bit	`DFTAG_CI8`	`DFTAG_CI`	`DFTAG_CI`
	Image Dimension: 8-bit	`DFTAG_ID8`	`DFTAG_ID`	`DFTAG_ID`
	Image Palette: 8-bit	`DFTAG_IP8`	`DFTAG_LUT`	`DFTAG_LUT`
DF24	Raster Image Group	None	`DFTAG_RIG`	`DFTAG_RIG`
	Raster Image (uncompressed)	None	`DFTAG_RI`	`DFTAG_RI`
	Compressed Image	None	`DFTAG_CI`	`DFTAG_CI`
	Image Dimension	None	`DFTAG_ID`	`DFTAG_ID`
DFP	Color Look-up Table	`DFTAG_LUT`	`DFTAG_LUT`	`DFTAG_LUT`
DFSD	Scientific Data Group	`DFTAG_SDG`	`DFTAG_SDG`	`DFTAG_NDG`
	Scientific Data	`DFTAG_SD`	`DFTAG_SD`	`DFTAG_SD`
	Scientific Data Dimension	`DFTAG_SDD`	`DFTAG_SDD`	`DFTAG_SDD`
	Scientific Data Scale Attribute	`DFTAG_SDS`	`DFTAG_SDS`	`DFTAG_SDS`
	Scientific Data Label Attribute	`DFTAG_SDL`	`DFTAG_SDL`	`DFTAG_SDL`
	Scientific Data Unit Attribute	`DFTAG_SDU`	`DFTAG_SDU`	`DFTAG_SDU`
	Scientific Data Format Attribute	`DFTAG_SDF`	`DFTAG_SDF`	`DFTAG_SDF`
	Scientific Data Max/Min Attribute	`DFTAG_SDM`	`DFTAG_SDM`	`DFTAG_SDM`
	Scientific Data Coordinates Attribute	`DFTAG_SDC`	`DFTAG_SDC`	`DFTAG_SDC`
DFAN	File Identifier	`DFTAG_FID`	`DFTAG_FID`	`DFTAG_FID`
	File Descriptor	`DFTAG_FD`	`DFTAG_FD`	`DFTAG_FD`
	Data Identifier Label	`DFTAG_DIL`	`DFTAG_DIL`	`DFTAG_DIL`
	Data Identifier Annotation	`DFTAG_DIA`	`DFTAG_DIA`	`DFTAG_DIA`
Vdata	Vdata Storage	`DFTAG_VS`	`DFTAG_VS`	`DFTAG_VS`
Vgroups	Vgroup Storage	`DFTAG_VG`	`DFTAG_VG`	`DFTAG_VG`

11.3. Programming Model for the DFAN Interface

There are two general programming models for the DFAN interface; the first programming model addresses file annotation while the second addresses object annotation. In the case of file annotations, the DFAN interface relies on the calling program to initiate and terminate access to files. This approach necessitates the following programming model:

1. Open the file.
2. Perform the desired file annotation operation.
3. Close the file.

The object annotation programming model is a simplified version of the file annotation programming model:

1. Perform the desired object annotation operation.

Essentially, the difference between the two models is that file annotations require **Hopen** and **Hclose** to open and close the target files whereas object annotations do not.

11.4. Writing Annotations

The DFAN interface supports writes to file labels, file descriptions, object labels, and object descriptions.

11.4.1. Assigning a File Label: DFANaddfid

To write a file label, the calling program must call **DFANaddfid**:

```
C:          status = DFANaddfid(file_id, label);

FORTRAN:    status = daafid(file_id, label)
```

DFANaddfid has two parameters: *file_id* and *label*. The *file_id* parameter contains the file identifier for the file to be annotated and the *label* parameter contains the annotation string. The label array must be null-terminated. In the FORTRAN-77 version, the length of the label should be the length of the label array as in FORTRAN-77 string lengths are assumed to be the declared length of the array that holds the string.

The parameters of **DFANaddfid** are further defined in Table 11C.

11.4.2. Assigning a File Description: DFANaddfds

To write a file description, the calling program must call **DFANaddfds**:

```
C:          status = DFANaddfds(file_id, description, desc_length);

FORTRAN:    status = daafds(file_id, description, desc_length)
```

DFANaddfds has three parameters: *file_id*, *description*, and *desc_length*. The *file_id* parameter contains the file identifier. The parameter *description* can contain any sequence of ASCII characters and is not limited to a single string (e.g., a carriage return may appear anywhere in the description). The *desc_length* parameter specifies the length of the description.

The parameters of **DFANaddfds** are defined in Table 11C.

TABLE 11C **DFANaddfid and DFANaddfds Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFANaddfid [intn] **(daafid)**	file_id	int32	integer	File identifier
	label	char *	character*(*)	File label string
DFANaddfds [intn] **(daafds)**	file_id	int32	integer	File identifier
	description	char *	character*(*)	File description string
	desc_length	int32	integer	Length of the description in bytes

EXAMPLE 1. **Writing a File Label and a File Description**

The following examples add a file label and description to the file named "Example1.hdf". Notice that after the file is opened, the file_id may be used to add any combination of file annotations before the file is closed.

C:

```
#include "hdf.h"

main( )
{

    int32 file_id;
    intn status;
    static char file_label[] = "This is a file label.";
    static char file_desc[] = "This is a file description.";

    /* Open the HDF file to write the annotations. */
    file_id = Hopen("Example1.hdf", DFACC_CREATE, 0);

    /* Write the label to the file. */
    status = DFANaddfid(file_id, file_label);

    /* Write the description to the file. */
    status = DFANaddfds(file_id, file_desc, strlen(file_desc));

    /* Close the file. */
    status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM CREATE ANNOTATION

       character*50 file_label, file_desc
       integer daafid, daafds, status, file_id, hopen, hclose

       integer*4 DFACC_CREATE
       parameter (DFACC_CREATE = 4)

       file_label = "This is a file label."
       file_desc = "This is a file description."

C      Open the HDF file to write the annotations.
       file_id = hopen('Example1.hdf', DFACC_CREATE, 0)

C      Write the label to the file.
       status = daafid(file_id, file_label)

C      Write the description to the file.
       status = daafds(file_id, file_desc, 26)

C      Close the file.
       status = hclose(file_id)

       end
```

11.4.3. Assigning an Object Label: DFANputlabel

To write a file label, the calling program must contain a call to **DFANputlabel**:

```
C:         status = DFANputlabel(filename, tag, ref, label);

FORTRAN:   status = daplab(filename, tag, ref, label)
```

DFANputlabel has four parameters: *filename*, *tag*, *ref*, and *label*. The *label* parameter contains a single null-terminated string that defines the annotation.

The parameters of **DFANputlabel** are further defined in Table 11D.

11.4.4. Assigning an Object Description: DFANputdesc

To write an object description, the calling program must contain a call to **DFANputdesc**:

```
C:          status = DFANputdesc(filename, tag, ref, description, desc_len);

FORTRAN:    status = dapdesc(filename, tag, ref, description, desc_len)
```

DFANputdesc has five parameters: *filename*, *tag*, *ref*, *description*, and *desc_len*. The *filename* parameter is the name of the HDF file containing the object to be annotated. The *tag* and *ref* parameters are the tag/reference number pair of the object to be annotated. The *description* parameter contains a buffer for the annotation text and the *desc_len* parameter specifies the length of the buffer.

The parameters of **DFANputdesc** are further defined in Table 11D.

TABLE 11D **DFANputlabel and DFANputdesc Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFANputlabel [intn] **(daplab)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag of the object to be annotated
	ref	uint16	integer	Reference number of the object to be annotated
	label	char *	character*(*)	Object label string
DFANputdesc [int] **(dapdesc)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag of the object to be annotated
	ref	uint16	integer	Reference number of the object to be annotated
	description	char *	character*(*)	Object description string
	desc_len	int32	integer	Length of the description in bytes

EXAMPLE 2. **Writing an Object Label and Description to a Scientific Data Set**

These examples illustrate the use of **DFANputlabel** and **DFANputdesc** to assign both an object label and an object description to a scientific data set immediately after it is written to file. The tag for scientific data sets is `DFTAG_NDG`.

C:

```
#include "hdf.h"

#define X_LENGTH 3
#define Y_LENGTH 2
#define Z_LENGTH 5

main( )
{

    /* Create the data array. */
    static float32 sds_data[X_LENGTH][Y_LENGTH][Z_LENGTH] =
    {  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 };

    /*
    * Create the array that will hold the dimensions of
    * the data array.
    */
    int32 dims[3] = {X_LENGTH, Y_LENGTH, Z_LENGTH};
    intn refnum, status;
    static char object_desc[] = "This is an object description.";
    static char object_label[] = "This is an object label.";

    /* Write the data to the HDF file. */
    status = DFSDadddata("Example1.hdf", 3, dims, (VOIDP)sds_data);

    /* Get the reference number for the newly written data set. */
    refnum = DFSDlastref( );

    /* Assign the object label to the scientific data set. */
    status = DFANputlabel("Example1.hdf", DFTAG_NDG, refnum, \
                   object_label);

    /* Assign the object description to the scientific data set. */
    status = DFANputdesc("Example1.hdf", DFTAG_NDG, refnum, \
                   object_desc, strlen(object_desc));

}
```

FORTRAN:

```
      PROGRAM ANNOTATE OBJECT

       integer dsadata, dims(3), status, refnum
       integer daplab, dapdesc, dslref

       integer*4 DFTAG_NDG, X_LENGTH, Y_LENGTH, Z_LENGTH
       parameter(DFTAG_NDG = 720,
      +          X_LENGTH = 5,
      +          Y_LENGTH = 2,
      +          Z_LENGTH = 3)

C      Create the data array.
       real*4 sds_data(X_LENGTH, Y_LENGTH, Z_LENGTH)
       data sds_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      Create the array the will hold the dimensions of the data array.
       data dims /X_LENGTH, Y_LENGTH, Z_LENGTH/

C      Write the data to the HDF file.
       ref = dsadata('Example1.hdf', 3, dims, sds_data)

C      Get the reference number for the newly written data set.
       refnum = dslref( )
```

```
C     Assign the object label to the scientific data set.
      status = daplab('Example1.hdf', DFTAG_NDG, refnum,
     +             'This is an object label.')

C     Assign an object description to the scientific data set.
      status = dapdesc('Example1.hdf', DFTAG_NDG, refnum,
     +              'This is an object description.', 30)

      end
```

11.5. Reading Annotations

The DFAN interface provides several functions for reading file and data object annotations, which are described below.

11.5.1. Reading a File Label: DFANgetfidlen and DFANgetfid

The DFAN programming model for reading a file label is as follows:

1. Get the length of the label.
2. Read the file label.

To read the first file label in a file, the calling program must contain the following function calls:

```
C:         isfirst = 1;
           label_length = DFANgetfidlen(file_id, isfirst);
           label_buffer = HDgetspace(label_length);
           fid_len = DFANgetfid(file_id, label_buffer, label_length, isfirst);

FORTRAN:   isfirst = 1
           label_length = dagfidl(file_id, isfirst)
           fid_len = dagfid(file_id, label_buffer, label_length, isfirst)
```

DFANgetfidlen has two parameters: *file_id* and *isfirst*. The *isfirst* parameter specifies whether the first or subsequent file annotations are to be read. To read the first file label length, *isfirst* should be set to the value `1`; to sequentially step through all the remaining file labels assigned to a file *isfirst* should be set to `0`.

When **DFANgetfidlen** is first called for a given file, it returns the length of the first file label. To get the lengths of subsequent file labels, you must call **DFANgetfid** between calls to **DFANget-fidlen**. Otherwise, additional calls to **DFANgetfidlen** will return the length of the same file label.

DFANgetfid has four parameters: *file_id*, *label_buffer*, *label_length*, and *isfirst*. The *label_buffer* parameter is a pointer to a buffer for the label text. The *label_length* parameter is the length of the buffer in memory, which can be shorter than the full length of the label in the file. If the *label_length* is not large enough, the label is truncated to *label_length* - 1 characters in the buffer *label_buffer*. The *isfirst* parameter is used to determine whether to read the first or subsequent file annotations. To read the first file label, *isfirst* should be set to `1`; to sequentially step through all the remaining file labels assigned to a file, *isfirst* should be set to `0`.

HDgetspace is described in Chapter 2, *HDF Fundamentals*.

The parameters of **DFANgetfidlen** and **DFANgetfid** are described in Table 11E.

11.5.2. Reading a File Description: DFANgetfdslen and DFANgetfds

The DFAN programming model for reading a file description is as follows:

1. Get the length of the description.
2. Read the file description.

To read the first file description in a file, the calling program must contain the following calls:

```
C:          isfirst = 1;
            desc_length = DFANgetfdslen(file_id, isfirst);
            desc_buffer = HDgetspace(desc_length);
            fds_len = DFANgetfds(file_id, desc_buf, desc_length, isfirst);

FORTRAN:    isfirst = 1
            desc_length = dagfdsl(file_id, isfirst)
            fds_len = dagfds(file_id, desc_buf, desc_length, isfirst)
```

DFANgetfdslen has two parameters: *file_id* and *isfirst*. The *isfirst* parameter specifies whether the first or subsequent file annotations are to be read. To read the first file description length, *isfirst* should be set to the value `1`; to sequentially step through all the remaining file descriptions assigned to a file, *isfirst* should be set to `0`.

When **DFANgetfdslen** is first called for a given file, it returns the length of the first file description. As with **DFANgetfidlen**, you must call **DFANgetfds** between calls to **DFANgetfdslen** to get the lengths of successive file descriptions.

DFANgetfds has four parameters: *file_id*, *desc_buf*, *desc_length*, and *isfirst*. The *desc_buffer* parameter is a pointer to a buffer for the description text. The *desc_length* parameter is the length of the buffer in memory, which can be shorter than the full length of the description in the file. If *desc_length* is not large enough, the description is truncated to *desc_length* characters in the buffer *desc_buf*. The *isfirst* parameter specifies whether the first or subsequent file annotations are to be read. To read the first file description, *isfirst* should be set to the value `1`; to sequentially step through all the remaining file descriptions assigned to a file, *isfirst* should be set to `0`.

The parameters of these routines are described further in the following table.

TABLE 11E

DFANgetfidlen, DFANgetfid, DFANgetfdslen, and DFANgetfds Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFANgetfidlen [int32] **(dagfidl)**	file_id	int32	integer	File identifier
	isfirst	intn	integer	Location of the next annotation
DFANgetfid [int32] **(dagfid)**	file_id	int32	integer	File identifier
	desc_buf	char *	character*(*)	File label buffer
	buf_length	int32	integer	Label buffer length
	isfirst	intn	integer	Location of the next annotation
DFANgetfdslen [int32] **(dagfdsl)**	file_id	int32	integer	File identifier
	isfirst	intn	integer	Location of the next annotation
DFANgetfds [int32] **(dagfds)**	file_id	int32	integer	File identifier
	description	char *	character*(*)	File description buffer
	desc_length	int32	integer	Description buffer length
	isfirst	intn	integer	Location of the next annotation

EXAMPLE 3.

Reading a File Label and a File Description

The following examples read a file label from the HDF file named "Example1.hdf". The **DFANgetfidlen** routine is used to verify the length of the label before the read operation is performed. The argument "1" in both routines indicate the first description in the HDF file is the target. **DFANgetfdslen** and **DFANgetfds** can be directly substituted for **DFANgetfidlen** and **DFANgetfid** in order to read a file description instead of a file label.

C:

```
#include "hdf.h"

main( )
{
    int32 file_id, file_label_len;
    char *file_label;
    intn status;

    /* Open the HDF file containing the annotation. */
    file_id = Hopen("Example1.hdf", DFACC_READ, 0);

    /* Determine the length of the file label. */
    file_label_len = DFANgetfidlen(file_id, 1);

    /* Allocated memory for the file label buffer. */
    file_label = HDgetspace(file_label_len);

    /* Read the file label. */
    file_label_len = DFANgetfid(file_id, file_label, file_label_len, 1);

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

}
```

FORTRAN:

```
      PROGRAM GET ANNOTATION

       integer status, file_id, label_length
       integer hopen, hclose, dagfidl, dagfid
       character file_label(50)

       integer*4 DFACC_READ
       parameter(DFACC_READ = 1)

C      Open the HDF file containing the file label.
       file_id = hopen("Example1.hdf", DFACC_READ, 0)

C      Determine the length of the file label.
       label_length = dagfidl(file_id, 1)

C      Read the file label.
       status = dagfid(file_id, file_label, label_length, 1)

C      Close the HDF file.
       status = hclose(file_id)

       end
```

11.5.3. Reading an Object Label: DFANgetlablen and DFANgetlabel

The DFAN programming model for reading a data object label is as follows:

1. Get the length of the label.
2. Read the file label.

To read the first object label in a file, the calling program must contain the following routines:

```
C:         label_length = DFANgetlablen(filename, tag, ref);
           label_buf = HDgetspace(label_length);
           status = DFANgetlabel(filename, tag, ref, label_buf, label_length);

FORTRAN:   label_length = daglabl(filename, tag, ref)
           status = daglab(filename, tag, ref, label_buf, label_length)
```

DFANgetlablen returns the length of the label assigned to the object identified by the given tag/reference number pair. **DFANgetlabel** must be called between calls to **DFANgetlablen**. **DFANgetlabel** is the routine that actually returns the label and prepares the API to read the next label.

DFANgetlabel has five parameters: *filename*, *tag*, *ref*, *label_buf*, and *label_length*. The *label_buf* parameter is a pointer to a buffer that stores the label text. The *label_length* parameter is the length of the buffer in memory. *label_length* can be shorter than the full length of the label in the file, but if so, the label is truncated to *label_length* characters in the buffer *label_buf*. The length of *label_buf* must be at least one greater than the anticipated length of the label to account for the null termination appended to the label text.

The parameters of **DFANgetlablen** and **DFANgetlabel** are defined below.

11.5.4. Reading an Object Description: DFANgetdesclen and DFANgetdesc

The DFAN programming model for reading a data object description is as follows:

1. Get the length of the description.
2. Read the file description.

To read the first object description in a file, the calling program must contain the following routines:

```
C:          desc_length = DFANgetdesclen(filename, tag, ref);
            desc_buf = HDgetspace(desc_length);
            status = DFANgetdesc(filename, tag, ref, desc_buf, desc_length);

FORTRAN:    label_length = dagdlen(filename, tag, ref)
            status = dagdesc(filename, tag, ref, desc_buf, desc_length)
```

DFANgetdesclen returns the length of the description assigned to the object identified by the specified tag/reference number pair. **DFANgetdesc** must be called between calls to **DFANgetdesclen** to reset the current object description to the next in the file.

DFANgetdesc takes five parameters: *filename*, *tag*, *ref*, *desc_buf*, and *desc_length*. The *desc_buf* parameter is a pointer to the buffer that stores the description text. The *desc_length* parameter is the length of the buffer in memory, which can be shorter than the full length of the description in the file. If the *desc_length* is not large enough, the description is truncated to *desc_length* characters in the buffer *desc_buf*.

The parameters of **DFANgetdesclen** and **DFANgetdesc** are defined in the following table.

TABLE 11F

DFANgetlablen, DFANgetlabel, DFANgetdesc and DFANgetdesclen Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFANgetlablen [int32] **(dagllen)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag assigned to the annotated object
	ref	uint16	integer	Reference number for the annotated object
DFANgetlabel [intn] **(daglab)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag assigned to the annotated object
	ref	uint16	integer	Reference number assigned to the annotated object
	label_buf	char *	character*(*)	Buffer for the returned annotation
	label_length	int32	integer	Size of the buffer allocated to hold the annotation
DFANgetdesclen [int32] **(dagdlen)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag assigned to the annotated object
	ref	uint16	integer	Reference number for the annotated object
DFANgetdesc [intn] **(dagdesc)**	filename	char *	character*(*)	Name of the file to be accessed
	tag	uint16	integer	Tag assigned to the annotated object
	ref	uint16	integer	Reference number assigned to the annotated object
	desc_buf	char *	character*(*)	Buffer for the returned annotation
	desc_length	int32	integer	Size of the buffer allocated to hold the annotation

EXAMPLE 4.

Reading an Object Label and Description

The following examples demonstrate the use of **DFANgetdesclen** and **DFANgetdesc** to read an object description assigned to a scientific data set. These examples assume that, in addition to other data objects, the "Example1.hdf" HDF file also contains multiple scientific data sets, some of which may not be annotated. **Hfind** is used to determine the reference number for the first annotated scientific data object in the file.

C:

```
#include "hdf.h"

main( )
{
    intn desc_length = -1, status;
    char desc[50];
    int32 file_id;
    uint16 tag = 0, ref = 0;
    uint32 find_offset, find_length;

    /* Open the file and initialize the searching parameters to 0. */
    file_id = Hopen("Example1.hdf", DFACC_READ, 0);

    /*
    * Start a sequential forward search for the first reference
    * number assigned to a scientific data set.
    */
    while (Hfind(file_id, DFTAG_NDG, DFREF_WILDCARD, &tag, &ref, \
            &find_offset, &find_length, DF_FORWARD) != FAIL) {

    /*
    * After discovering a valid reference number, check for an
```

```
        * object description by returning the length of the description.
        * If the inquiry fails, continue searching for the next valid
        * reference number assigned to a scientific data set.
        */
        if ((desc_length = DFANgetdesclen("Example1.hdf", tag, ref)) \
                == FAIL)
                break;

        /*
        * If a description exists and it will fit in the description buffer,
        * print it.
        */
        if (desc_length != FAIL && desc_length <= 50) {
                status = DFANgetdesc("Example1.hdf", tag, ref, desc, desc_length);
                printf("Description: %s\n", desc);
        }
        }

        /* Close the file. */
        status = Hclose(file_id);

    }
```

FORTRAN:

There is no FORTRAN-77 version of the Example 4 C code for this version of the documentation as there is no FORTRAN-77 equivalent of **Hfind**.

11.6. Maintenance Routines

The DFAN interface provides one function for interface maintenance, **DFANclear**.

11.6.1. Clearing the DFAN Interface Internal Structures and Settings: DFANclear

DFANclear clears all internal library structures and parameters of the DFAN annotation interface.

When a file is regenerated in a single run by a library routine of another interface (such as **DFSDputdata**), **DFANclear** should be called to reset the interface

DFANclear returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise. **DFANclear** takes no parameters, as described in the following table.

TABLE 11G **DFANclear Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFANclear [intn] **(daclear)**	None	None	None	None

11.7. Determining Reference Numbers

It is advisable to check the reference number before attempting to assign an object annotation, as the overwriting of reference numbers is not prevented by the HDF library routines.

There are three ways to check a reference number for an object:

- Access the object with a read or write operation followed by **DF*lastref**.
- Call **DFANlablist** to return a list of all assigned reference numbers for a given tag.
- Call **Hfind** to locate an object with a given tag/reference number pair.

11.7.1. Determining a Reference Number for the Last Object Accessed: DF*lastref and DF*writeref

There are two methods of obtaining a reference number through the use of a **DF*lastref** call. The first approach is to obtain and store the reference number of an object immediately after the object is created:

1. Create the data object.
2. Call **DF*lastref** to determine its reference number.
3. Read or write an object annotation.

The second approach is to determine the reference number at some time after the data object is created. This approach requires repeated **DF*read** calls until the appropriate object is accessed, followed by a call to **DF*lastref**:

1. Read the appropriate data object.
2. Call **DF*lastref** to determine its reference number.
3. Read or write and object annotation.

Most HDF interfaces provide one routine that assigns a specified reference number to a data object and another routine that returns the reference number for the last data object accessed. (See TABLE 11H) However, the SD interface doesn't. Also, the DFAN annotation doesn't include a **DF*lastref** routine.

Although **DF*writeref** calls are designed to assign specific reference numbers, they are not recommended for general use because there is no protection against reassigning an existing reference number and overwriting data. In general, it is better to determine a reference number for a data object by calling **DF*lastref** immediately after reading or writing a data object.

The **DF*lastref** routines have no parameters. The **DF*writeref** routines have two: *filename*, which is the name of the file that contains the data object, and *ref*, which is the reference number for the next data object read operation.

The **DF*lastref** and **DF*writeref** routines are further described in the following table.

TABLE 11H **List and Descriptions of the DF*writeref and DF*lastref Routines**

HDF Data Object	Routine Name (FORTRAN-77)	Description
8-bit Raster Image	`DFR8writeref` (`d8wref`)	Assigns the specified number as the reference number for the next 8-bit raster write operation and updates the write counter to the reflect highest reference number
	`DFR8lastref` (`d8lref`)	Returns the reference number for the last 8-bit raster image set accessed
24-bit Raster Image	`DF24writeref` (`d2wref`)	Assigns the specified number as the reference number for the next 24-bit raster write operation and updates the write counter to reflect the highest reference number
	`DF24lastref` (`d2lref`)	Returns the reference number for the last 24-bit raster image set accessed
Palette	`DFPwriteref` (`dpwref`)	Assigns the specified number as the reference number for the next palette write operation and updates the write counter to reflect the highest reference number
	`DFPlastref` (`dplref`)	Returns the reference number for the last palette accessed
DFSD Scientific Data	`DFSDwriteref` (`dswref`)	Assigns the specified number as the reference number for the next SDS write operation and updates the write counter to reflect the highest reference number
	`DFSDlastref` (`dslref`)	Returns the reference number for the last scientific data set accessed
Annotation	`DFANlastref` (`dalref`)	Returns the reference number for the last annotation accessed

11.7.2. Querying a List of Reference Numbers for a Given Tag: DFANlablist

Given a tag and two buffers, **DFANlablist** will fill one buffer with all reference numbers for the given tag and the other with all labels assigned to the given tag. The programming model for determining a list of reference numbers is as follows:

1. Determine the number of reference numbers that exist for a given tag.
2. Allocate a buffer to store the reference numbers.
3. Specify the maximum label length.
4. Allocate a buffer to store the labels.
5. Store the list of reference numbers and their labels.

To create a list of reference numbers and their labels for a given tag, the following routines should be called:

```
C:          num_refs = Hnumber(file_id, tag);
            ref_buf = HDmalloc(sizeof(uint16*)*num_refs);
            max_lab_len = 16;
            label_buf = HDmalloc(max_lab_len * num_refs);
            start_pos = 0;
            num_of_refs = DFANlablist(filename, tag, ref_buf, label_buf,
            num_refs, max_lab_len,
            start_pos);

FORTRAN:    num_refs = hnumber(file_id, tag)
            max_lab_len = 16
            start_pos = 0
            num_of_refs = dallist(filename, tag, ref_buf, label_buf,
            num_refs, max_lab_len, start_pos)
```

Hnumber determines how many objects with the specified tag are in a file. It is described in Chapter 2, *HDF Fundamentals*.

DFANlablist has seven parameters: *filename*, *tag*, *ref_list*, *label_buf*, *num_refs*, *max_lab_len*, and *start_pos*. The *filename* parameter specifies the name of the file to search and *tag* specifies the search tag to use when creating the reference and label list. The *ref_buf* and *label_buf* parameters are buffers used to store the reference numbers and labels associated with *tag*. The *num_ref* parameter specifies the length of the reference number list and the *max_lab_len* parameter specifies the maximum length of a label. The *start_pos* parameter specifies the first label to read. For instance, if *start_pos* has a value of `1` all labels will be read; if it has a value of `4`, all but the first three labels will be read.

Taken together, the contents of *ref_list* and *label_list* constitute a directory of all objects and their labels for a given tag. The contents of *label_list* can be displayed to show all of the labels for a given tag or it can be searched to find the reference number of a data object with a certain label. Once the reference number for a given label is found, the corresponding data object can be accessed by invoking other HDF routines. Therefore, this routine provides a mechanism for direct access to data objects in HDF files.

TABLE 11I

DFANlablist Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFANlablist [int] **(dallist)**	filename	char *	character*(*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref_list	uint16 []	integer (*)	Reference number for the annotated object.
	label_list	char *	character*(*)	Buffer for the labels.
	list_len	int	integer	Size of the reference number and label lists.
	label_len	intn	integer	Maximum label length.
	start_pos	intn	integer	First entry in the reference number and label lists to be returned.

EXAMPLE 5.

Getting a List of Labels for All Scientific Data Sets

These examples illustrate the method used to get a list of all labels used in scientific data sets in an HDF file using **DFANlablist**. The `DFS_MAXLEN` definition is located in the "hlimits.h" include file.

C:

```
#include "hdf.h"

#define LISTSIZE 20

main( )
{

    int i, num_of_labels, start_position = 1, list_length = 10;
    uint16 ref_list[LISTSIZE];
    char label_list[DFS_MAXLEN*LISTSIZE-1];

    /* Get the total number of labels in the "Example1.hdf" file. */
    num_of_labels = DFANlablist("Example1.hdf", DFTAG_NDG, ref_list, \
                  label_list, list_length, DFS_MAXLEN,   \
                start_position);

    /*
    * Print the reference numbers and label names for each label
    * in the list.
    */
    for (i = 0; i < num_of_labels; i++)
      printf("\n\t%d\tRef number: %d\tLabel: %s", i+1, ref_list[i], \
             label_list - (i * 13));

    printf("\n");

}
```

FORTRAN:

```
      PROGRAM GET LABEL LIST

       integer dallist
       integer*4 DFTAG_NDG, LISTSIZE, DFS_MAXLEN

       parameter (DFTAG_NDG = 720,
      +          LISTSIZE = 20,
      +          DFS_MAXLEN = 255)

       character*60 label_list(DFS_MAXLEN*LISTSIZE)
       integer i, num_of_labels, start_position, ref_list(DFS_MAXLEN)

       start_position = 1

       num_of_labels = dallist('Example1.hdf', DFTAG_NDG, ref_list,
      +                        label_list, 10, DFS_MAXLEN,
      +                        start_position)

       do 10 i = 1, num_of_labels
         print *,'    Ref number:  ',ref_list(i),
      +         '    Label: ',label_list(i)
 10    continue

       end
```

11.7.3. Locate an Object by Its Tag and Reference Number: Hfind

Instead of using **DFANlablist** to create a list of reference numbers to search, HDF provides a general search routine called **Hfind**. **Hfind** is described in Chapter 2, *HDF Fundamentals*.

CHAPTER 12 -- Single-File Scientific Data Sets (DFSD API)

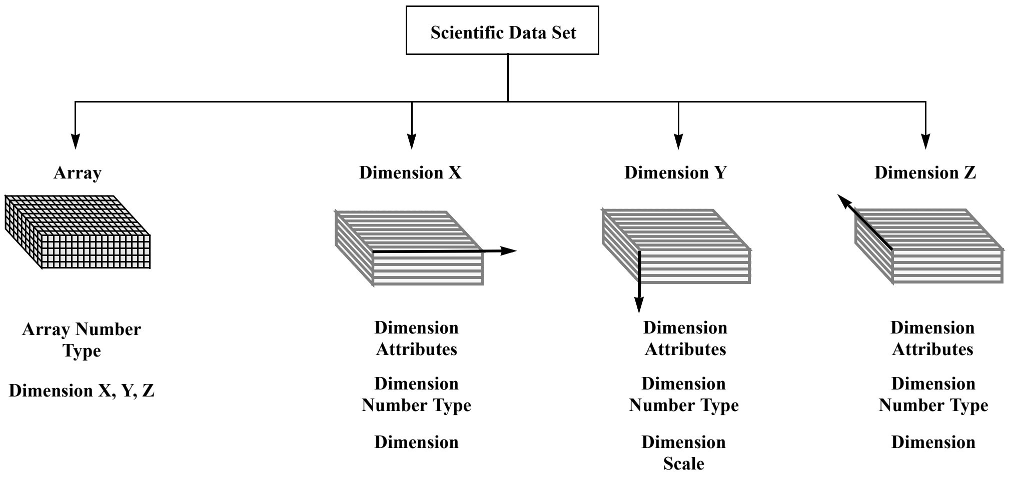
12.1. Chapter Overview

The DFSD interface was one of two interfaces in the HDF library that support the scientific data model. With the release of HDF version 3.3, the multifile SD interface described in Chapter 3, *Scientific Data Sets (SD API)*, was made available. The DFSD interface is now deprecated, only the SD interface should be used.

12.2. The DFSD Scientific Data Set Data Model

The scientific data set, or SDS, data model supports four primary data objects: arrays, dimensions, dimension scales, and dimension attributes. As in the multifile SD SDS model, the fundamental object of the data model is the SDS array. Unlike the SD multifile SDS model the DFSD SDS model has, in addition to dimension attributes, attributes that refer to the SDS array itself.

FIGURE 12a **The Contents of a Three-Dimensional DFSD Scientific Data Set**



12.2.1. Required DFSD SDS Objects

The only required objects in the DFSD SDS model are the ***array*** and the ***data type*** of the array data. Without this information, the data set is inaccessible. Required objects are created by the library using the information supplied at the time the SDS is defined.

Descriptions of these objects are in Chapter 3, *Scientific Data Sets (SD API)*.

12.2.1.1. Dimensions

Unlimited dimensions, supported in the multifile SD SDS model, aren't supported in the single-file DFSD SDS model.

12.2.2. Optional DFSD SDS Objects

There are two types of optional objects available for inclusion in an SDS: dimension scales and attributes. Optional objects are only created when specified by the calling program.

12.2.2.1. Dimension Scales

Conceptually, a dimension ***scale*** is a series of numbers placed along a dimension to demarcate intervals in a data set. They are assigned one per dimension. Structurally, each dimension scale is a one-dimensional array with size and name equal to its assigned dimension name and size.

12.2.2.2. Predefined Attributes

Predefined attributes are attributes that have reserved labels and in some cases predefined number types. They are described in Chapter 3, *Scientific Data Sets (SD API)*.

12.1. The Single-File Scientific Data Set Interface

The HDF library currently contains several routines for storing scientific data sets in the HDF format. **DFSDadddata**, **DFSDputdata**, and **DFSDgetdata** perform data I/O and by default assume that all scientific data is uncompressed 32-bit floating-point data stored in row-major order. DFSD library routines also read and write subsets and slabs of data, set defaults, determine the number of data sets in a file, and inquire about or assigning reference numbers before reading or writing data.

12.1.1. DFSD Library Routines

The names of the C routines in the DFSD library are prefaced by "DFSD" and the names of the equivalent FORTRAN-77 functions are prefaced by "ds". They are categorized as follows:

- ***Write routines*** create new data sets and add slabs to existing data sets.
- ***Read routines*** read whole scientific data sets.
- ***Slab routines*** read and write subsets and slabs of scientific data.
- ***Data set attribute routines*** read and write the predefined string and value attributes assigned to data sets.
- ***Dimension attribute routines*** read and write the predefined string and value attributes assigned to dimensions.

DFSD library routines are more explicitly defined in Table 12A and on their respective pages in the *HDF Reference Manual*.

TABLE 12A **DFSD Library Routines**

Category	Routine Name		Description
	C	FORTRAN-77	
Write	`DFSDadddata`	`dsadata`	Appends a data set to a file.
	`DFSDclear`	`dsclear`	Clears all possible set values.
	`DFSDputdata`	`dspdata`	Overwrites new data to a file.
	`DFSDsetdims`	`dssdims`	Sets the rank and dimension for succeeding data sets.
	`DFSDsetNT`	`dssnt`	Sets the number type for the data set.
	`DFSDwriteref`	`dswref`	Assigns a reference number to the next data set written.
Read	`DFSDgetdata`	`dsgdata`	Retrieves the next data set in the file.
	`DFSDgetdims`	`dsgdims`	Returns the number and dimensions for the next data set.
	`DFSDgetNT`	`dsgnt`	Determines the number type for the data in the data set.
	`DFSDlastref`	`dslref`	Returns the reference number of last data set accessed.
	`DFSDndatasets`	`dsnum`	Returns the number of data sets in a file.
	`DFSDpre32sdg`	`dsp32sd`	Determines if the data set was created before HDF version 3.2.
	`DFSDreadref`	`dsrref`	Locates a data set with the specified reference number.
	`DFSDrestart`	`dsfirst`	Sets the location of the next access operation to be the first data set in the file.
Slabs	`DFSDendslab`	`dssslab`	Terminates a read or write slab operation.
	`DFSDreadslab`	`dsrslab`	Reads a slab of data from a data set.
	`DFSDstartslab`	`dssslab`	Begins a read or write slab operation.
	`DFSDwriteslab`	`dswslab`	Writes a slab of data to a data set.
Data Set Attribute	`DFSDgetcal`	`dsgcal`	Retrieves the calibration information for the data se.t
	`DFSDgetdatalen`	`dsgdaln`	Retrieves the length of the attributes assigned to the data.
	`DFSDget-datastrs`	`dsgdast`	Returns the label, unit, format and coordinate system for data.
	`DFSDgetfill-value`	`dsgfill`	Retrieves the fill value used to complete the data set.
	`DFSDgetrange`	`dsgrang`	Retrieves the range of values for the data set.
	`DFSDsetcal`	`dsscal`	Sets the calibration information for the data set.
	`DFSDset-datastrs`	`dssdast`	Sets label, unit, format and coordinate system for data.
	`DFSDsetfill-value`	`dssfill`	Sets the fill value to use when completing a data set.
	`DFSDsetlengths`	`dsslens`	Sets the length for the data set and dimension attributes.
	`DFSDsetrange`	`dssrang`	Sets the range of values for the data set.
Dimension Attribute	`DFSDgetdimlen`	`dsgdiln`	Retrieves the length of the attributes assigned to the dimension.
	`DFSDgetdims-cale`	`dsgdisc`	Returns the scale for a dimension.
	`DFSDgetdimstrs`	`dsgdist`	Returns the label, unit, and format for a dimension.
	`DFSDsetdims-cale`	`dssdisc`	Sets the scale for a dimension.
	`DFSDsetdimstrs`	`dssdist`	Sets the label, unit and format for the dimension.

12.1.2. File Identifiers in the DFSD Interface

File identifiers are handled internally by each routine and access to a file is granted simply by providing a filename. As the file identifier is handled by the function call, the calling program need not keep track of how to open and close files.

12.2. Writing DFSD Scientific Data Sets

The DFSD programming model for writing an SDS to an HDF file involves the following steps:

1. Define data set options. (optional)
2. Write all or part of the data set.

These steps are performed for every data set written to a file. However, it is not always necessary to define data set options for every write operation as setting an option places information about the data set in a structure in primary memory. This information is retained until explicitly altered by another set call.

12.2.1. Creating a DFSD Scientific Data Set: DFSDadddata and DFSDputdata

To define and write a single SDS, the calling program must contain of of the following routines:

```
C:          status = DFSDadddata(filename, rank, dim_sizes, data);

FORTRAN:    status = dsadata(filename, rank, dim_sizes, data)

    OR

C:          status = DFSDputdata(filename, rank, dim_sizes, data);

FORTRAN:    status = dspdata(filename, rank, dim_sizes, data)
```

DFSDadddata appends data to a file when given an existing file name and creates a new file when given a unique file name. **DFSDputdata** replaces the contents of a file when given an existing file name and creates a new file when given a unique file name. To avoid accidentally overwriting data in a file, the use of **DFSDadddata** is recommended.

DFSDadddata and **DFSDputdata** have four parameters: *filename*, *rank*, *dim_sizes*, and *data*. In both routines, the data set is written to the file specified by the *filename* parameter. The total number of dimensions in the array and the size of each dimension are passed in the *rank* and *dim_sizes* parameters. A pointer to the data or slab of data written to the named file is passed in the *data* parameter.

The parameters of **DFSDadddata** and **DFSDputdata** are further described in the following table.

TABLE 12B **DFSDadddata and DFSDputdata Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDadddata [intn] **(dsadata)**	filename	char *	character*(*)	Name of the file containing the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 *	integer(*)	Size of each dimension in the data array.
	data	VOIDP	<valid numeric data type>	Array containing the data.
DFSDputdata [intn] **(dsadatas)**	filename	char *	character*(*)	Name of the file containing the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 *	integer(*)	Size of each dimension in the data array.
	data	VOIDP	<valid numeric data type>	Array containing the data.

12.2.2. Specifying the Data Type of a DFSD SDS: DFSDsetNT

The default data type for scientific data is `DFNT_FLOAT32`. To change the default setting, the calling program must contain calls to the following routines:

```
C:          status = DFSDsetNT(number_type);
            status = DFSDadddata(filename, rank, dim_sizes, data);

FORTRAN:    status = dssnt(number_type)
            status = dsadata(filename, rank, dim_sizes, data)
```

DFSDsetNT defines the data type for all subsequent **DFSDadddata** and **DFSDputdata** calls until it is changed by a subsequent call to **DFSDsetNT** or reset to the default by **DFSDclear**. **DFSDsetNT**'s only parameter is the data type.

EXAMPLE 1. **Creating and Writing to a DFSD Scientific Data Set**

In the following code examples, **DFSDadddata** is used to write an array of 64-bit floating-point numbers to a file named "Example1.hdf". Although the **DFSDsetNT** function call is optional, it is included here to demonstrate how to override the float32 default.

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. };

    intn status;

    int32 dims[3] = {LENGTH, HEIGHT, WIDTH};

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

    /* Write the data to file */
    status = DFSDadddata("Example1.hdf", 3, dims, scien_data);

}
```

FORTRAN:

```
     PROGRAM WRITE SDS

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

C     Create 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 /3,2,5/

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

C     Write the data to file
      status = dsadata('Example1.hdf', 3, dims, sci_data)

      end
```

12.2.3. Overwriting Data for a Given Reference Number: DFSDwriteref

DFSDwriteref is a highly specialized function call that overwrites data referred to by the specified reference number.

If **DFSDwriteref** is called with a reference number that doesn't exist, an error return value of -1 will be returned.

The following series of function calls should appear in your program:

```
C:          status = DFSDwriteref(filename, ref_number);
            status = DFSDadddata(filename, rank, dim_sizes, data);

FORTRAN:    status = dswref(filename, ref_number)
            status = dsadata(filename, rank, dim_sizes, data)
```

If the filename passed to **DFSDwriteref** is different from the filename in the **DFSDadddata** or **DFSDputdata** routine calls, it will be ignored. The next scientific data set written, regardless of the filename, is assigned the reference number *ref_number*.

Care should be taken when using **DFSDwriteref**, as once the new data has been written the old data cannot be retrieved.

The parameters of **DFSDwriteref** are described in the following table.

TABLE 12C **DFSDsetNT and DFSDwriteref Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDsetNT [intn] **(dssNT)**	number_type	int32	integer	Number type tag.
DFSDwriteref [intn] **(dswref)**	filename	char *	character*(*)	Name of the file containing the data.
	ref_number	int16	integer	Reference number to be assigned to the data set created.

12.2.4. Writing Several Data Sets: DFSDsetdims and DFSDclear

The DFSD programming model for writing multiple data sets to an HDF file is identical to that for writing individual data sets. (Refer to Section "*Writing DFSD Scientific Data Sets*"). To understand how multiple data sets are written to file, it is first necessary to take a closer look at each step of the programming model. First and most importantly, all DFSD routines that set a write option except **DFSDsetNT** and **DFSDsetfillvalue** add information to a special structure in primary memory. This information is used to determine how data is written to file for all subsequent write operations.

Information stored in primary memory is retained by the HDF library until explicitly changed by a call to **DFSDsetdims** or reset to `NULL` by calling **DFSDclear**. **DFSDsetdims** and **DFSDclear** are used to prevent assignments of attributes created for a group of data sets to data sets outside the group. For more information on assigning attributes see Section "*Writing Data Set Attributes*" and Section "*Writing the Dimension Attributes of a DFSD SDS*".

12.2.5. Preventing the Reassignment of DFSD Data Set Attributes: DFSDsetdims

Information stored in primary memory is retained by the HDF library until explicitly changed by a call to **DFSDsetdims** or reset to `NULL` by calling **DFSDclear**. **DFSDsetdims** and **DFSDclear** are used to prevent assignments of attributes created for a group of data sets to data sets outside the group.

The syntax of **DFSDsetdims** is the following:

```
C:          status = DFSDsetdims(rank, dim_sizes);

FORTRAN:    status = dssdims(rank, dim_sizes)
```

DFSDsetdims is not used here to define the rank and dimension sizes to be used in the next operation, but to alert the DFSD interface to stop the automatic assignment of attributes to the data sets to be written to file. **DFSDsetdims** has two parameters: *rank* and *dim_sizes*. The rank of an array is the total number of dimensions in the array and the dimension sizes are the length of each individual dimension.

As a rule of thumb, **DFSDsetdims** should be called if any **DFSDset*** routine (**DFSDsetNT**, for example) has been called. This insures that all attribute values that have been reset will be assigned in future data set operations.

The parameters of **DFSDsetdims** are further defined in the following table.

TABLE 12D **DFSDsetdims Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDsetdims [intn] **(dssdims)**	rank	intn	integer	Number of dimensions in the array.
	dim_sizes	int32*	integer (*)	Size of each dimension in the array.

12.2.6. Resetting the Default DFSD Interface Settings: DFSDclear

The syntax for **DFSDclear** is as follows:

```
C:          status = DFSDclear( );

FORTRAN:    status = dsclear( )
```

The **DFSDclear** routine clears all interface settings defined by any of the **DFSDset*** routines (**DFSDsetNT**, **DFSDsetfillvalue**, **DFSDsetdims**, **DFSDsetdatastrs**, **DFSDsetdatalengths**, **DFSDsetrange**, **DFSDsetcal**, **DFSDsetdimscale** and **DFSDsetdimstrs**). After the **DFSDclear** has been called, calls to any of the **DFSDset*** routines will result in the corresponding value not being written. To write new values, call the appropriate DFSDset routine again.

TABLE 12E **DFSDclear Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDclear [intn] **(dsclear)**	None	None	None	Clears all DFSD interface settings.

12.3. Reading DFSD Scientific Data Sets

The DFSD programming model for reading an SDS is also a two-step operation:

1. Obtain information about the data set if necessary.
2. Read all or part of the data set.

These steps are performed for every data set read. In some cases, calls to determine the data set definition may be reduced or avoided completely. For example, if the data set dimensions are known, the call that returns the data set dimensions may be eliminated.

12.3.1. Reading a DFSD SDS: DFSDgetdata

If the dimensions of the data set are known, **DFSDgetdata** is the only function call required to read an SDS. If the file is being opened for the first time, **DFSDgetdata** returns the first data set in the file. Any subsequent calls will return successive data sets in the file - data sets are read in the same order they were written. Normally, **DFSDgetdims** is called before **DFSDgetdata** so that space allocations for the array can be checked if necessary and the dimensions verified. If this information is already known, **DFSDgetdims** may be omitted.

To read an SDS of known dimension and number type, the calling program should include the following routine:

```
C:          status = DFSDgetdata(filename, rank, dim_sizes, data);

FORTRAN:    status = dsgdata(filename, rank, dim_sizes, data)
```

DFSDgetdata has four parameters: *filename*, *rank*, *dim_sizes*, and *data*. **DFSDgetdata** returns a data set specified by the parameter *filename*. The total number of dimensions is specified in *rank* and the size of each dimension is specified in *dim_sizes.* **DFSDgetdata** returns the array in *data*.

The parameters of **DFSDgetdata** are further defined in the following table.

TABLE 12F **DFSDgetdata Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDgetdata [intn] **(dsgdata)**	filename	char	character*(*)	Name of the file containing the data.
	rank	int32	integer	Number of dimensions.
	dim_sizes	int32 *	integer (*)	Buffer for the dimension sizes.
	data	VOIDP	<valid numeric data type>	Buffer for the stored scientific data.

12.3.2. Specifying the Dimensions and Data Type of an SDS: DFSDgetdims and DFSDgetNT

When **DFSDgetdims** is first called, it returns dimension information of the first data set. Subsequent calls will return this information for successive data sets. If you need to determine the dimensions or the data type of an array before reading it, call **DFSDgetdims** and **DFSDgetNT**. **DFSDgetNT** gets the data type (or, in HDF parlance, number type) of the data retrieved in the next read operation.

To determine the dimensions and data type of an array before attempting to read it, the calling program must include the following:

```
C:          status = DFSDgetdims(filename, rank, dimsizes, max_rank);
            status = DFSDgetNT(number_type);
            status = DFSDgetdata(filename, rank, dimsizes, data);

FORTRAN:    status = dsgnt(filename, rank, dimsizes, max_rank)
            status = dsgdims(number_type)
            status = dsgdata(filename, rank, dimsizes, data)
```

DFSDgetdims has four parameters: *filename*, *rank*, *dim_sizes*, and *maxrank*. The number of dimensions is returned in *rank*, the size of each dimension in the array *dim_sizes*, and the size of the array containing the dimensions sizes in *max_rank*. **DFSDgetNT** has only one parameter: *number_type*. As there is no way to specify the file or data set through the use of **DFSDgetNT,** it is only valid if it is called after **DFSDgetdims**.

The parameters of **DFSDgetdims** and **DFSDgetNT** are further defined in the following table.

TABLE 12G

DFSDgetNT and DFSDgetdims Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDgetdims [intn] **(dsgdims)**	filename	char *	character*(*)	Name of file containing the data.
	rank	intn *	integer	Number of dimensions.
	dim_sizes	int32 *	integer	Buffer for the dimension sizes.
	max_rank	int	integer	Size of the dimension size buffer.
DFSDgetNT [intn] **(dsgnt)**	number_type	int32 *	integer	Data type of the data to be read.

EXAMPLE 2.

Reading from a DFSD Scientific Data Set

The following examples search the file named "Example1.hdf" for the dimensions and data type of a DFSD array. Although use of **DFSDgetdims** and **DFSDgetNT** is optional, they are included here as a demonstration of how to verify the array dimensions and number type before reading any data. If the dimensions and type are known, only a call to **DFSDgetdata** 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, status;
    int32 dims[3];

    /* Get the dimensions and number type of the array */
    status = DFSDgetdims("Example1.hdf", &rank, dims, 3);
    status = DFSDgetNT(&number_type);

    /* Read the array if the dimensions are correct */
    if (dims[0] <= LENGTH && dims[1] <= HEIGHT && dims[2] <= WIDTH)
            status = DFSDgetdata("Example1.hdf", rank, dims, scien_data);

}
```

FORTRAN:

```
     PROGRAM READ SDS

      integer  dsgdata, dsgdims, dsgnt, dims(3), status
      integer rank, num_type
      real*8   sci_data(5, 2, 3)

C     Get the dimensions and number type of the array.
      status = dsgdims('Example1.hdf', rank, dims, 3)
      status = dsgnt(num_type)
```

```
C     Read the array if the dimensions are correct.
      if ((dims(1) .eq. 3) .and. (dims(2) .eq. 2) .and.
     +    (dims(3) .eq. 5)) then
          status = dsgdata('Example1.hdf', rank, dims, sci_data)
      endif

      end
```

12.3.3. Determining the Number of DFSD Data Sets: DFSDndatasets and DFSDrestart

DFSDgetdims and **DFSDgetdata** sequentially access DFSD data sets. By repeatedly calling either function, a program can step through an entire file by reading one data set at a time. However, before attempting to sequentially access all of the data sets in a file the total number of data sets in the file should be determined. To do so, the calling program must call the following routine:

```
C:          num_of_datasets = DFSDndatasets(filename);

FORTRAN:    num_of_datasets = dsnum(filename)
```

Once the total number of data sets is known, a calling program can at any time, reset the current data set to the first data set in the file by calling the following routine:

```
C:          status = DFSDrestart( );

FORTRAN:    status = dsfirst( )
```

Use of **DFSDndatasets** and **DFSDrestart** is optional, it is usually more convenient than cycling through the entire file one SDS at a time.

12.3.4. Obtaining Reference Numbers of DFSD Data Sets: DFSDreadref and DFSDlastref

As the HDF library handles the assignment and tracking of reference numbers, reference numbers must be explicitly returned. Obtaining the reference number is an operation best performed immediately after data set creation.

The DFSD interface uses the function **DFSDreadref** to initiate access to individual scientific data sets. **DFSDreadref** specifies the reference number of the next SDS to be read.

To access a specific SDS, the calling program must contain the following routines:

```
C:          status = DFSDreadref(filename, ref);
            status = DFSDgetdata(filename, rank, dim_sizes, data);

FORTRAN:    status = dsrref(filename, ref)
            status = dsgdata(filename, rank, dim_sizes, data)
```

DFSDreadref has two parameters: *filename* and *ref*. **DFSDreadref** specifies the reference number of the object to be next operated on in the HDF file *filename* as *ref*. Determining the correct reference number is the most difficult part of this operation. As a result, **DFSDreadref** is often used in conjunction with **DFSDlastref**, which determines the reference number of the last data set accessed.

To syntax of **DFSDadddata** and **DFSDlastref** is:

```
C:          status = DFSDadddata(filename, rank, dim_sizes, data);
            ref_num = DFSDlastref( );

FORTRAN:    status = dsadata(filename, rank, dim_sizes, data)
            ref_num = dslref( )
```

DFSDputdata can also be used with **DFSDlastref** to obtain similar results. In any case, **DFSD-lastref** can be used before any operation that requires identifying a scientific data set by reference number, as in the assignment of annotations and inserting data sets into vgroups. For more information about annotations and vgroups refer to, Chapter 10, *Annotations (AN API)* and Chapter 5, *Vgroups (V API)*.

TABLE 12H **DFSDreadref Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDreadref [intn] **(dsrref)**	filename	char *	character*(*)	Name of the file containing the data set.
	ref_number	uint16	integer	Reference number of the next data set to be read.

12.4. Slabs in the DFSD Interface

To review, a slab is an n-dimensional array whose dimensions are smaller than those of the SDS array into which it is written or from which it is read.

12.4.1. Accessing Slabs: DFSDstartslab and DFSDendslab

There are two routines required for every DFSD slab operation - **DFSDstartslab** and **DFSDend-slab**. **DFSDstartslab** is used to initialize the slab interface and to initiate access to new or existing data sets. **DFSDendslab** is used to terminate access to open data sets. **DFSDstartslab** must be called before any read or write slab operation and **DFSDendslab** must be called after the slab operation is completed. Both routines are required when reading and writing slabs.

Given a new filename, **DFSDstartslab** will create a new HDF file with the specified name. Given an existing filename, it will open the named file and append the new data set to the end of the file. Its only parameter is *filename*. **DFSDendslab** has no parameters and need only be called once per file. **DFSDendslab** will write any attributes defined immediately before the data set is created.

For more information on assigning attributes, see Section "*Writing the Dimension Attributes of a DFSD SDS*".

TABLE 12I **DFSDstartslab Parameter List**

Routine Name [Return Type] **(FOR-TRAN-77)**	Parame-ter	Parameter Type		Description
		C	FOR-TRAN-77	
DFSDstartslab [intn] **(dssslab)**	filename	char *	character*(*)	Name of the file containing the data set.

12.4.2. Writing Slabs: DFSDwriteslab

In the DFSD interface, writing an entire data set array and writing slabs follow the same programming model. The difference between the two is that calls to three routines is needed to write slabs, while a call to one routine is needed to write whole data sets.

More specifically, the DFSD programming model for writing slabs to an SDS is as follows:

1. Set the appropriate options to define the new SDS or select an existing SDS.
2. Write the data set using three specialized slab routines.

In addition to writing slabs to both new and existing data sets, **DFSDwriteslab** can also perform the following sequential write operations:

- Write slabs to a single data set when called repeatedly.
- Write slabs to sequential data sets when repeatedly called between calls to **DFSDgetdims**.
- Write slabs to selected data sets when repeatedly called between calls to **DFSDwriteref**.

Although not specifically defined as a slab routine, in practice, the **DFSDsetfillvalue** routine is used to initialize array elements between non-contiguous slab write operations. Setting a fill value places the same value in every array location before the first slab is written. Any hole created by non-contiguous writes can then be recognized by identifying the known fill value. The fill value must have the same number type as the values in the data set. For more information on fill values refer to Section "*Assigning Value Attributes to a DFSD SDS: DFSDsetfillvalue, DFSDsetrange, and DFSDsetcal*".

To write a slab to a new data set, the calling program must include the following routine calls:

```
C:          status = DFSDsetdims(rank, dimsizes);
            status = DFSDsetNT(num_type);
            status = DFSDstartslab(filename);
            status = DFSDwriteslab(start, stride, count, data);
            status = DFSDendslab( );

FORTRAN:    status = dssnt(num_type)
            status = dssdims(rank, dim_sizes)
            status = dssslab(filename)
            status = dswslab(start, stride, edge, data)
            status = dseslab( )
```

When writing slabs to an existing data set, it is impossible to change the number type, array boundaries, fill value, or calibration information. Consequently **DFSDsetNT**, **DFSDsetdims**, **DFSDsetcal**, and **DFSDsetfillvalue** will generate errors if called for an existing data set.

To write a slab to an existing data set, your program should include the following calls:

```
C:          status = DFSDwriteref(filename, ref);
```

```
          status = DFSDstartslab(filename);
          status = DFSDwriteslab(start, stride, count, data);
          status = DFSDendslab( );

FORTRAN:  status = dswref(filename, ref)
          status = dssslab(filename)
          status = dswslab(start, stride, edge, data)
          status = dseslab( )
```

Because **DFSDwriteslab** offers no overwrite protection, the calling program is responsible for eliminating overlap when arranging slabs within the newly defined data set.

DFSDwriteslab has four arguments: *start*, *stride*, *edge*, and *data*. The arguments *start*, *stride*, and *edge* are defined as they are in the corresponding SD routines.

The DFSD SDS model does not support strides. Pass the *start* array as the *stride* parameter as a place holder. Whatever is passed as the *stride* parameter will be ignored by the DFSD interface.

Although **DFSDendslab** need only be called once per file, it is required to write data to the file. It will also write any attributes defined immediately before the data set is created.

TABLE 12J **DFSDwriteslab Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDwriteslab [intn] **(sdwslab)**	start	int32 *	integer (*)	Array containing the starting coordinate the write.
	stride	int32 *	integer (*)	Ignored parameter.
	count	int32 *	integer (*)	Array defining the boundaries of the slab.
	data	VOIDP	<valid numeric data type>	Buffer for the data to be written.

12.4.3. Reading Slabs: DFSDreadslab

The programming model for reading one or more slabs involves the following steps:

1. Select an existing SDS.
2. Read the data set using three specialized slab routines.

In addition to reading single slabs of data, **DFSDreadslab** can perform the following sequential access operations:

- Read multiple slabs from the first data set in a file when called repeatedly.
- Read multiple slabs from a specified data set when repeatedly called after **DFSDreadref**.
- Read multiple slabs from sequential data sets when repeatedly called between calls to **DFSDgetdims**.

To read a slab, the calling program must include the following routine calls:

```
C:        status = DFSDreadref(filename, ref);
          status = DFSDstartslab(filename);
          status = DFSDreadslab(start, stride, edge, data);
          status = DFSDendslab( );

FORTRAN:  status = dsrref(filename, ref)
```

```
status = dssslab(filename)
status = dsrslab(start, stride, edge, data)
status = dseslab( )
```

In addition to **DFSDreadref**, **DFSDgetdims** may also be used to position the read pointer to the appropriate data set. When **DFSDreadslab** is used to read slabs, the coordinates of the *start* array must begin at 0 for each dimension (*start={0,0, ... 0}*) and the size of each dimension must equal the size of the array itself (*edge={dim_size_1, dim_size_2, dim_size_n}*). As with **DFSD-writeslab**, whatever is passed in as the *stride* parameter is ignored. Finally, the *data* buffer must allocate enough space to hold the data: excess data is truncated.

All parameters of the **DFSDreadslab** routine assume FORTRAN-77-style one-based arrays - the starting coordinates of the slab must be given as an offset from the origin of the data set where the origin is defined as (*dim 1 = 1, dim 2 = 1, . . . dim n-1 = 1, dim n = 1*). The first element of the slab will be the coordinates specified by the contents of the *start* array. **DFSDreadslab** will extract elements in increasing order until the until the dimensional offset specified by the contents of the *edge* array are encountered.

TABLE 12K **DFSDreadslab Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDreadslab [intn] **(dswslab)**	filename	char *	character*(*)	Name of the HDF file.
	start	int32 *	integer (*)	Array containing the coordinates for start of the slab.
	slab_size	int32	integer (*)	Array of rank containing the size of each dimension of the slab.
	stride	int32 *	integer (*)	Place holder array.
	buffer	VOIDP	<valid numeric data type>	Array the will used to store the extracted slab.
	buffer_size	int32 *	integer (*)	Array containing the dimensions of the buffer parameter.

12.5. Predefined Attributes and the DFSD Interface

Although they often contain important information, attributes are optional to the data set array and the dimension record. Although both types of attributes use similar names, they are read and written using different sets of routines. All attributes are predefined by the DFSD library.

12.5.1. Writing Data Set Attributes

Data set attributes are described in Chapter 3, *Scientific Data Sets (SD API)*. There is a limit of one string attribute per data set.

12.5.1.1. Assigning String Attributes to a DFSD SDS: DFSDsetlengths and DFSDsetdatastrs

The DFSD interface provides two function calls for creating data set string attributes: **DFSDsetlengths** and **DFSDsetdatastrs**. **DFSDsetlengths** overrides the default string length and **DFSDsetdatastrs** writes the string. **DFSDsetlengths** and **DFSDsetdatastrs** are optional and may be called individually, or in any order as long as they precede calls to **DFSDadddata** or **DFSDputdata**.

Predefined string attributes are defined as follows:

- ***Coordinate system attributes*** specify the coordinate system used to generate the original data.
- ***Format attributes*** specify the format to use when displaying values for the data.
- ***Label attributes*** contains data array names.
- ***Unit attributes*** identifies the units of measurement associated with the data.

To assign a predefined attribute to an HDF file, the program must contain the following routine calls:

```
C:          status = DFSDsetlengths(label_len, unit_len, format_len, coords_len);
            status = DFSDsetdatastrs(label, unit, format, coordsys);
            status = DFSDadddata(filename, rank, dimsizes, data);

FORTRAN:    status = dsslens(label_len, unit_len, format_len, coords_len)
            status = dssdast(label, unit, format, coordsys)
            status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetlengths has four arguments: *label_len*, *unit_len*, *format_len*, and *coords_len*. Each parameter reflects the maximum length for the string that will hold the label, unit, format, and coordinate system. Use of **DFSDsetlengths** is optional and usually not necessary.

DFSDsetdatastrs writes null-terminated strings to an HDF file. It has the same four arguments: *label*, *unit*, *format*, and *coordsys*. To avoid the assignment of a string, pass NULL as the appropriate argument.

TABLE 12L **DFSDsetlengths and DFSDsetdatastrs Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDsetlengths [intn] **(dsslens)**	label_len	intn	integer	Maximum length for any label string.
	unit_len	intn	integer	Maximum length for any unit string.
	format_len	intn	integer	Maximum length for any format string.
	coords_len	intn	integer	Maximum length for any coordinate system string.
DFSDsetdatastrs [intn] **(dssdast)**	label	char *	character*(*)	Label describing the data.
	unit	char *	character*(*)	Unit to be applied to the data.
	format	char *	character*(*)	Format to be applied in displaying the data.
	coordsys	char*	character*(*)	Coordinate system of the data set.

12.5.1.2. Assigning Value Attributes to a DFSD SDS: DFSDsetfillvalue, DFSDsetrange, and DFSDsetcal

The DFSD interface provides the following routines for defining value attributes. All three function calls are optional and may be called in any order provided they precede a call to **DFSDadddata** or **DFSDputdata**.

To assign a value attribute to a data set, the following routines must be called:

```
C:          status = DFSDsetfillvalue(fill_val);
            status = DFSDsetcal(scale, scale_err, offset, offset_err, num_type);
            status = DFSDsetrange(max, min);
            status = DFSDadddata(filename, rank, dimsizes, data);
```

```
FORTRAN:    status = dssfill(fill_val)
            status = dsscal(scale, scale_err, offset, offset_err, num_type)
            status = dssrang(max, min)
            status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetrange sets a new range attribute for the current DFSD SDS. **DFSDsetrange** has two arguments: *max* and *min*. The HDF library will not check or update the range attributes as new data are added to the file, therefore *max* and *min* will always reflect the values supplied by the last **DFSDsetrange** call. The parameters for **DFSDsetrange** is defined in Table 12K below.

DFSDsetfillvalue specifies a new value to the default fill value attribute for an SDS array. It's only argument is *fill_val*, which specifies the new fill value. The fill value must be of the same number type as the array it's written to. To avoid conversion errors, use data-specific fill values instead of special architecture-specific values, such as infinity or Not-a-Number (or ***NaN***). Setting the fill value after data is written to the SDS will not update the fill values already written to the data set - it will only change the attribute.

The parameters for **DFSDsetfillvalue** are further defined in Table 12K below.

The **DFSDsetcal** routine creates a calibration record for a specified array and by doing so adds five attributes to the current data set. As the HDF library does not specifically apply calibration information to the data, **SDsetcal** can be called anytime before or after the data is written. **DFSD-setcal** has five arguments; *scale*, *scale_error*, *offset*, *off_err*, and *num_type*. The arguments *scale* and *offset* are defined as they are for the multifile SD API routines.

In addition to the *scale* and *offset*, **DFSDsetcal** also includes both a scale and offset error. The argument *scale_err* contains the potential error of the calibrated data due to scaling and *offset_err* contains the potential error for the calibrated data due to the offset. The *num_type* parameter specifies the number type of the uncalibrated data.

The parameters of **DFSDsetcal** are defined in the following table.

TABLE 12M **DFSDsetfillvalue, DFSDsetrange and DFSDsetcal Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDsetfillvalue [intn] **(dssfill)**	label	char *	character*(*)	Label describing the data.
	unit	char *	character*(*)	Unit to be applied to the data.
	format	char *	character*(*)	Format to be applied in displaying the data.
	coordsys	char *	character*(*)	Coordinate system of the data set.
DFSDsetrange [intn] (dssrang)	max	VOIDP	<valid numeric data type>	Highest value in the selected range of data.
	min	VOIDP	<valid numeric data type>	Lowest value in the selected range of data.
DFSDsetcal [intn] (dsscal)	cal	float64	real*8	Calibration scale.
	cal_error	float64	real*8	Calibration scale error.
	off	float64	real*8	Uncalibrated offset.
	off_err	float64	real*8	Uncalibrated offset error.
	num_type	int32	integer	Number type of uncalibrated data.

EXAMPLE 3. **Assigning Predefined String Attributes to a File**

The following examples demonstrate the steps necessary to assign predefined string attributes to the data set and stores the data set in the file "Example1.hdf". They create a string attribute using **DFSDsetdatastrs** and a value attribute using **DFSDsetrange**. It also demonstrates the use of **DFSDsetlengths** in altering the maximum string length from 255 characters to 50. It then writes the SDS array by calling **DFSDadddata**.

C:

```
#include "hdf.h"

/*
 *  Write an array of floating point values representing
 *  pressure in a 3x2x5 array.
 */

main( )
{

    float32 data[3][2][5];
    int32 dimsizes[3];
    float32 max, min;
    intn status, rank;
    int i, j, k;

    /* Set the rank and dimension sizes. */
    rank = 3;
    dimsizes[0] = 3;
    dimsizes[1] = 2;
    dimsizes[2] = 5;

    /* Set the dimensions, to define the beginning of a data set. */
    status = DFSDsetdims(rank, dimsizes);
    /* Set the maximum string length to 50. */
    status = DFSDsetlengths(50, 50, 50, 50);

    /* Define the attribute strings and values. */
    status = DFSDsetdatastrs("Pressure Data", "Millibars",
                    "F5.5", "None");
    max = 1500.0;
    min = 0.0;
    status = DFSDsetrange(&max, &min);

    /* Set the rank to 3. */
    rank = 3;

    /* Calculate the data values. */
    for (i = 0; i < 3; i++)
        for (j = 0; j < 2; j++)
            for (k = 0; k < 5; k++)
                data[i][j][k] = i*100.0 + j*10.0 + k;

    /* Write the data set and its attributes to file. */
    status = DFSDadddata("Example3.hdf", rank, dimsizes, data);

}
```

FORTRAN:

```
      PROGRAM SET ATTRIBS

       real*8 data(5, 2, 3), max, min, i, j, k
       integer*4 dimsizes(3)
       integer status, rank

       integer dsslens, dssdast, dssrang, dsadata
       integer dssdims

       character*13 label /"Pressure Data"/
       character*9 unit /"Millibars"/
       character*4 format /"F5.5"/
       character*4 coordsys /"None"/

C      Set the dimensions, to define the beginning of a data set.
       rank = 3
       dimsizes(1) = 5
       dimsizes(2) = 2
       dimsizes(3) = 3
       status = dssdims(rank, dimsizes)

C      Set the maximum string lengths to 50.
       status = dsslens(50, 50, 50, 50)

C      Define the attribute strings and values.
       status = dssdast(label, unit, format, coordsys)
       max = 1500.0
       min = 0.0
       status = dssrang(max, min)

C      Fill the data array with values.
       do 30 k = 1, 3
        do 20 j = 1, 2
         do 10 i = 1, 5
           data(i, j, k) = i*100.0 + j*10.0 + k
10       continue
20      continue
```

```
30    continue

C     Write the data set and its attributes to file.
      status = dsadata("Example3.hdf", rank, dimsizes, data)

      end
```

12.5.2. Reading DFSD Data Set Attributes

The DFSD interface provides two function calls for reading predefined data set attribute strings.

12.5.2.1. Reading Data Set Attributes: DFSDgetdatalen and DFSDgetdatastrs

DFSDgetdatalen returns the length of each string in the attribute. It is useful for determining the length of an attribute before reading it. **DFSDgetdatastrs** reads the label, unit, format, and coordinate system strings.

Attribute data is not read by **DFSDgetdatastrs** until the appropriate routine is called to read the array and its dimension record. If **DFSDgetdatastrs** and **DFSDgetrange** are not called, the array and its dimension record can be read without reading its associated data set attributes. It is also possible to read string and value attributes individually. As attribute data is not actually read by **DFSDgetdatastrs** or **DFSDgetrange**, these calls must be made before calling **DFSDgetdata**.

Reading the attributes of a data set involves the following steps:

1. Determine the length of each attribute string.
2. Read the attribute strings.
3. Read the maximum and minimum values.
4. Read the remainder of the data set.

To assign a predefined attribute to an HDF file, the following routines should be called:

```
C:          status = DFSDgetdatalen(label_len, unit_len, format_len, coords_len);
            status = DFSDgetdatastrs(label, unit, format, coordsys);
            status = DFSDgetrange(max, min);
            status = DFSDgetdata(filename, rank, dimsizes, data);

FORTRAN:    status = dsgdghaln(label_len, unit_len, format_len, coords_len)
            status = dsgdast(label, unit, format, coordsys)
            status = dsgrang(max, min)
            status = dsgdata(filename, rank, dimsizes, data)
```

The parameters of **DFSDgetdatalen** and **DFSDgetdatastrs** are described in the following table.

TABLE 12N

DFSDgetdatalen and DFSDgetdatastrs Parameter List

Routine Name [Return Type] **(FORTRAN-77)**	**Parame-ter**	**Parameter Type**		**Description**
		C	FORTRAN-77	
DFSDgetdatalen [intn] **(dsgdaln)**	label_len	intn *	integer	Length of any label string.
	unit_len	intn *	integer	Length of any unit string.
	format_len	intn *	integer	Length of any format string.
	coords_len	intn *	integer	Length of any coordinate system string.
DFSDgetdatastrs [intn] **(dsgdast)**	label	char *	character*(*)	Label describing the data.
	unit	char *	character*(*)	Unit applied to the data.
	format	char *	character*(*)	Format applied to the data.
	coordsys	char *	character*(*)	Coordinate system of the data set.

EXAMPLE 4.

Reading a Data Set and its Attribute Record

These examples read the pressure data set and the dimension attribute record stored in the "Example1.hdf" file into the arrays pointed to by the *data*, *datalabel*, *dataunit*, *datafmt* and *coordsys* pointer variables. It assumes the dimension sizes and rank are correct and data strings are less than 10 characters long, with one additional character for the null termination.

C:

```
#include "hdf.h"

main( )
{

    intn rank, maxrank, status;
    int32 dimsizes[3];
    char datalabel[50], dataunit[50], datafmt[50], coordsys[50];
    float64 data[3][2][5];

    maxrank = 3;
    status = DFSDgetdims("Example3.hdf", &rank, dimsizes,
                       maxrank);
    status = DFSDgetdatastrs(datalabel, dataunit, datafmt,
                       coordsys);
    status = DFSDgetdata("Example3.hdf", rank, dimsizes, data);

}
```

FORTRAN:

```
      PROGRAM READ SD INFO

       integer dsgdata, dsgdast, dsgdims
       integer*4 dimsizes(3)
       integer status, rank, maxrank
       character*50 datalabel, dataunit, datafmt
       character*10 coordsys
       real*8 data(5, 2, 3)

       maxrank = 3
       status = dsgdims('Example3.hdf', rank, dimsizes, maxrank)
       status = dsgdast(datalabel, dataunit, datafmt, coordsys)
```

```
        status = dsgdata('Example3.hdf', rank, dimsizes, data)

        end
```

12.5.2.2. Reading the Value Attributes of a DFSD Data Set: DFSDgetfillvalue and DFSDgetcal

There are three routines in the DFSD interface that retrieve the fill value, range and calibration information of a data set array: **DFSDgetfillvalue**, **DFSDgetrange**, and **DFSDgetcal**.

The syntax of these routines are as follows:

```
C:          status = DFSDgetfillvalue(sds_id, fill_val);
            status = DFSDgetrange(max, min);
            status = DFSDgetcal(cal, cal_err, offset, offset_err, num_type);

FORTRAN:    status = dsgfill(fill_value)
            status = dsgrang(max, min)
            status = dsadata(cal, cal_err, offset, offset_err, num_type)
```

DFSDgetfillvalue has two arguments; *sds_id* and *fill_val*. The *sds_id* is the data set identifier and *fill_val* is the space allocated to store the fill value.

The maximum range of values in the data set isn't automatically stored with the data set data; it is explicitly stored through a call to **DFSDgetrange**. The defined range of values can be less than the actual range of values stored in the data set. The value of the *max* parameter is the maximum value of the defined range and the value of the *min* parameter is the minimum value. These values must be of the same number type as the values stored in the data array. In C, the *max* and *min* parameters are indirect pointers specifying the range values, while in FORTRAN-77 they are variables set to the range values.

DFSDgetcal reads the calibration record of the current data set, if one exists. Each of the parameters of **DFSDgetcal** correspond to the five elements of the calibration record; - four 64-bit floating-point integers followed by a 32-bit integer. The *cal*, *offset*, *offset_err* and *cal_err* parameters are defined as they are in the multifile SD API. This calibration record exists for information only.

The parameters for **DFSDgetfillvalue**, **DFSDgetcal**, and **DFSDgetrange** are defined in the following table.

TABLE 12O **DFSDgetfillvalue, DFSDgetcal and DFSDgetrange Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDgetfillvalue [intn] **(dsgfill)**	sds_id	int32	integer	Data set identifier.
	fill_val	VOIDP	<valid numeric data type>	Buffer for the fill value.
DFSDgetcal [int32] **(dsgcal)**	cal	float64 *	real*8	Calibration factor.
	cal_err	float64 *	real*8	Calibration error.
	offset	float64 *	real*8	Uncalibrated offset.
	offset_err	float64 *	real*8	Uncalibrated offset error.
	num_type	int32 *	integer	Type of the uncalibrated data.
DFSDgetrange [intn] **(dsgrang)**	max	VOIDP	<valid numeric data type>	Highest value of the selected range.
	min	VOIDP	<valid numeric data type>	Lowest value of the selected range.

12.5.3. Writing the Dimension Attributes of a DFSD SDS

Dimension attributes are described in Chapter 3, *Scientific Data Sets (SD API)*.

12.5.3.1. Writing the String Attributes of a Dimension: DFSDsetlengths and DFSDsetdimstrs

The DFSD interface provides two routines for creating dimension string attributes: **DFSDsetlengths** and **DFSDsetdimstrs**. **DFSDsetlengths** overwrites the default string length and **DFSDsetdimstrs** is defines the string text. **DFSDsetdatalengths** and **DFSDsetdimstrs** are optional and must precede calls to **DFSDadddata** or **DFSDputdata**.

Predefined dimension string attributes are limited to one per dimension and contain the following:

- ***Format attributes*** specify the format to use when displaying values for the dimension.
- ***Label attributes*** contain dimension names.
- ***Unit attributes*** identify the unit of measurement associated with the dimension.

To assign a predefined attribute to a dimension, the following routines should be called:

```
C:          status = DFSDsetlengths(label_len, unit_len, format_len, coords_len);
            status = DFSDsetdimstrs(label, unit, format);
            status = DFSDadddata(filename, rank, dimsizes, data);

FORTRAN:    status = dsslens(label_len, unit_len, format_len, coords_len)
            status = dssdist(label, unit, format)
            status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetlengths has four arguments: *label_len*, *unit_len*, *format_len*, and *coords_len*. Each parameter specifies the maximum length of the string that defines the label, unit, format, and coordinate system. As mentioned earlier in this chapter, attribute lengths seldom need to be reset.

DFSDsetdimstrs also has four arguments; *dim*, *label*, *unit*, and *format*. The parameter *dim = 1* for the first dimension, *dim = 2* for the second dimension, etc. To avoid assigning a string to the coordinate length, pass `NULL` in the appropriate parameter. **DFSDsetdimstrs** writes null-terminated strings to a file.

The parameters for **DFSDsetlengths** and **DFSDsetdimstrs** are further defined in the following table.

TABLE 12P **DFSDsetlengths and DFSDsetdimstrs Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDsetlengths [intn] **(dsslen)**	label_len	intn	integer	Maximum length of any label string.
	unit_len	intn	integer	Maximum length of any unit string.
	format_len	intn	integer	Maximum length of any format string.
	coords_len	intn	integer	Maximum length of any coordinate system string.
DFSDsetdimstrs [intn] **(dssdist)**	dim	intn	integer	Dimension of the attribute strings.specified by the remaining three parameters
	label	char *	character*(*)	Label describing the data.
	unit	char *	character*(*)	Unit to be applied to the data.
	format	char *	character*(*)	Format to be applied in displaying the data.

12.5.3.2. Writing a Dimension Scale of a DFSD SDS: DFSDsetdimscale

The syntax of the two routines needed to write a dimension scale is the following:

```
C:          status = DFSDsetdimscale(dim, dimsize, scale);
            status = DFSDadddata(filename, rank, dimsizes, data);

FORTRAN:    status = dssdisc(dim, dimsize, scale)
            status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetdimscale has three arguments; *dim*, *dimsize*, and *scale*. These arguments identify the dimension, specify its size, and assign a value to each of its grid points. The parameter *dim = 1* for the first dimension, and *dim = 2* for the second dimension. The *dimsize* argument must contain a value equal to the dimension it describes in order for the scale to be applied correctly.

The parameters of **DFSDsetdiscale** are further described in the following table.

TABLE 12Q **DFSDsetdimscale Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
DFSDsetdimscale [intn] **(dssdisc)**	dim	intn	integer	Dimension of the current scale.
	dim_size	int32	integer	Size of the current scale.
	scale	VOIDP	<valid numeric data type>	Values of the current scale.

12.5.4. Reading the Dimension Attributes of a DFSD SDS

The DFSD interface provides three routines for reading dimension attributes: **DFSDgetdimlen**, **DFSDgetdimstrs** and **DFSDgetdimscale**. **DFSDgetdimlen** returns the string length for each string in the attribute record. It is a useful routine to call before reading an attribute. **DFSDgetdimstrs** and **DFSDgetdimscale** are used as instructions for reading the dimension attributes.

DFSDgetdimstrs reads the dimension strings and **DFSDgetdimscale** reads the dimension scale. By avoiding calls to **DFSDgetdimstrs** and **DFSDgetdimscale**, it is possible to read an array and its dimension record without reading the data set attributes associated with it. It is also possible to omit one function call in order to read one attribute without the other. Also, note that **DFSDgetdimstrs** and **DFSDgetdimscale** must be called before **DFSDgetdata**.

Reading data set attributes involves the following steps:

1. Determine the length of each attribute string.
2. Read the attribute strings.
3. Read the scale values.
4. Read the remainder of the data set.

These steps are translated into the following function calls:

```
C:          status = DFSDgetdimlen(label_len, unit_len, format_len, coords_len);
            status = DFSDgetdimstrs(label, unit, format);
            status = DFSDgetdimscale(dim, dim_size, scale);
            status = DFSDgetdata(filename, rank, dimsizes, data);

FORTRAN:    status = dsgdiln(label_len, unit_len, format_len, coords_len)
            status = dsgdist(label, unit, format)
            status = dsgdisc(dim, dim_size, scale)
            status = dsgdata(filename, rank, dimsizes, data)
```

The parameters for **DFSDgetdimlen**, **DFSDgetdimstrs** and **DFSDgetdimscale** are described in the following table.

TABLE 12R **DFSDgetdimlen, DFSDgetdimstrs and DFSDgetdimscale Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
DFSDgetdimlen [intn] **(dsgdiln)**	dim	intn	integer	Dimension of the string attributes describe.
	label_len	intn *	integer	Length of the label attribute string.
	unit_len	intn *	integer	Length of the unit attribute string.
	format_len	intn *	integer	Length of the format attribute string.
DFSDgetdimstrs [intn] **(dsgdist)**	dim	intn	integer	Dimension the string attributes describe.
	label	char *	character*(*)	Label of the dimension.
	unit	char *	character*(*)	Unit to be applied to this dimension.
	format	char *	character*(*)	Format to be applied when displaying the scale.
DFSDgetdimscale [intn] **(dsgdisc)**	dim	intn	integer	Dimension the current scale is attached to
	dim_size	int32	integer	Size of the current scale.
	scale	VOIDP	<valid numeric data type>	Values of the current scale.

CHAPTER 13 -- Error Reporting

13.1. Chapter Overview

This chapter describes the main error reporting routines designed for general HDF use and the types of errors handled by the error reporting API and the general structure of the API.

13.2. The HDF Error Reporting API

The HDF error reporting API consists of routines that query error stack information, the names of which are prefaced by "HE". They are described briefly in Table 13A. Some are primarily for use by HDF developers while others are available to HDF users. In this chapter, three error reporting functions are covered: **HEprint**, **HEvalue** and **HEstring**. Note that only one C error reporting routine has a FORTRAN-77 counterpart: **heprnt/heprntf** (**heprntf** is the newer function, supported on all platforms; **heprnt** is the original function, supported on non-Microsoft Windows platforms).

TABLE 13A **Error Reporting Routine List**

Category	Routine Name		Description
	C	FORTRAN-77	
Error Reporting	`HEprint`	`heprnt` `heprntf`	Prints the errors on the error stack to a specified file.
	`HEstring`	`hestringf`	Returns the error message associated with an error code.
	`HEvalue`	`None`	Returns the nth most recent error reported.

13.3. Error Reporting in HDF

Most HDF error reporting routines return `FAIL` (or `-1`) if the operation is successful and `SUCCEED` (or `0`) otherwise. Each time a `FAIL` code is returned one or more error codes are pushed onto the error code stack. The following pseudo-code will demonstrate the two methods commonly used to access and print the contents of this stack.

```
if (<general HDF function() >= FAIL) {
    <HDF error reporting API routines>
}
```

OR

```
status = <general HDF function( );
if (status == FAIL) {
    <HDF error reporting API routines>
}
```

A list of error codes is included at the end of this chapter.

13.3.1. Writing Errors to a File: HEprint

HEprint writes the errors on the stack to the specified file. There are four sections of an **HEprint** error report:

1. A description of the error.
2. The routine in which the error was detected.
3. The source file in which the error was detected.
4. The line number in which the error was detected.

The syntax for **HEprint** is as follows:

```
C:          HEprint(stream, level);

FORTRAN:    status = heprnt(level)
```

The *stream* parameter is a UNIX file handle indicating the output stream the error information will be written to. The *level* parameter specifies the amount of error information to report. In FORTRAN-77, **heprnt** (supported on non-Microsoft Windows platforms) always writes to the standard error stream, or *stderr*; therefore the only parameter is *level*. To facilitate Microsoft Windows support, a newer function **heprntf** (supported on all platforms) requires two parameters, *filename* to identify the file to which the error information is to be written and *level*.

Errors are written in sequential order starting from the bottom of the stack. Consequently, specifying a *level* parameter value of 1 will write the first error that occurred, or the first error pushed onto the stack. Specifying a *level* parameter of value 0 will write all errors on the stack to the specified file. For example, the following C code will write all errors on the stack to the file named "errors".

```
f = fopen("errors", "w");
HEprint(f, 0);
```

As an example of the output of **HEprint**, suppose an attempt is made to open an nonexistent file with **Hopen**. Calling `HEprint(stdout, 0)` or `heprnt(0)` will produce the following output:

```
HDF error: <error opening file>
Detected in Hopen() [hfile.c line 305]
```

13.3.2. Returning the Code of the Nth Most Recent Error: HEvalue

HEvalue returns the error code for the nth most recent error and is only available as a C routine. The *level* parameter specifies the number of errors to regress from the top of the error stack, i.e., `HEvalue(1)` will return the error code at the top of the stack. Refer to Table 13B for a complete list of HDF4 error codes.

The syntax for **HEvalue** is as follows:

```
C:          status = HEvalue(level);
```

13.3.3. Returning the Description of an Error Code: HEstring/hestringf

HEstring returns the error description associated with the error code specified by the *error_code* parameter as a character string.

The syntax for **HEstring** is as follows:

```
C:          error_message = HEstring(error_code);

FORTRAN:    status = hestringf(error_code, error_message)
```

13.3.4. Clearing the error stack: HEclear

HEclear clears all information on reported errors from the error stack and is only available as a C routine. The syntax for **HEclear** is as follows:

```
C:          status = HEclear();
```

Note that every HDF4 API calls **HEclear** to clear the error stack.

EXAMPLE 1. **Writing Errors to a Console Window**

The following C code fragment will copy errors from the stack to a console window.

C:

```
#include "hdf.h"

    main( )
{

    int32 i, e;
    const char *str;
    ...
    i = 0;
    while ((e = HEvalue(i)) != DFE_NONE) {
                str = HEstring(e);
                <device-specific code to print the string to a console>
                i++
    ...
}
```

TABLE 13B

HDF Error Codes

Error Code	Code Definition
`DFE_NONE`	No error.
`DFE_FNF`	File not found.
`DFE_DENIED`	Access to file denied.
`DFE_ALROPEN`	File already open.
`DFE_TOOMANY`	Too many AID's or files open.
`DFE_BADNAME`	Bad file name on open.
`DFE_BADACC`	Bad file access mode.
`DFE_BADOPEN`	Miscellaneous open error.
`DFE_NOTOPEN`	File can't be closed because it hasn't been opened.
`DFE_CANTCLOSE`	**fclose** error
`DFE_READERROR`	Read error.
`DFE_WRITEERROR`	Write error.
`DFE_SEEKERROR`	Seek error.
`DFE_RDONLY`	File is read only.
`DFE_BADSEEK`	Attempt to seek past end of element.
`DFE_PUTELEM`	**Hputelement** error.
`DFE_GETELEM`	**Hgetelement** error.
`DFE_CANTLINK`	Cannot initialize link information.
`DFE_CANTSYNC`	Cannot synchronize memory with file.
`DFE_BADGROUP`	Error from **DFdiread** in opening a group.
`DFE_GROUPSETUP`	Error from **DFdisetup** in opening a group.
`DFE_PUTGROUP`	Error on putting a tag/reference number pair into a group.
`DFE_GROUPWRITE`	Error when writing group contents.
`DFE_DFNULL`	Data file reference is a null pointer.
`DFE_ILLTYPE`	Data file contains an illegal type: internal error.
`DFE_BADDDLIST`	The DD list is non-existent: internal error.
`DFE_NOTDFFILE`	The current file is not an HDF file and it is not zero length.
`DFE_SEEDTWICE`	The DD list already seeded: internal error.
`DFE_NOSUCHTAG`	No such tag in the file: search failed.
`DFE_NOFREEDD`	There are no free DDs left: internal error.
`DFE_BADTAG`	Illegal WILDCARD tag.
`DFE_BADREF`	Illegal WILDCARD reference number.
`DFE_NOMATCH`	No DDs (or no more DDs) that match the specified tag/reference number pair.
`DFE_NOTINSET`	Warning: Set contained unknown tag. Ignored.
`DFE_BADOFFSET`	Illegal offset specified.
`DFE_CORRUPT`	File is corrupted.
`DFE_NOREF`	No more reference numbers are available.
`DFE_DUPDD`	The new tag/reference number pair has been allocated.
`DFE_CANTMOD`	Old element doesn't exist. Cannot modify.
`DFE_DIFFFILES`	Attempt to merge objects in different files.
`DFE_BADAID`	An invalid AID was received.
`DFE_OPENAID`	Active AIDs still exist.
`DFE_CANTFLUSH`	Cannot flush DD back to file.
`DFE_CANTUPDATE`	Cannot update the DD block.
`DFE_CANTHASH`	Cannot add a DD to the hash table.

Error Code	Code Definition
`DFE_CANTDELDD`	Cannot delete a DD in the file.
`DFE_CANTDELHASH`	Cannot delete a DD from the hash table.
`DFE_CANTACCESS`	Cannot access specified tag/reference number pair.
`DFE_CANTENDACCESS`	Cannot end access to data element.
`DFE_TABLEFULL`	Access table is full.
`DFE_NOTINTABLE`	Cannot find element in table.
`DFE_UNSUPPORTED`	Feature not currently supported.
`DFE_NOSPACE`	**malloc** failed.
`DFE_BADCALL`	Routine calls were in the wrong order.
`DFE_BADPTR`	`NULL` pointer argument was specified.
`DFE_BADLEN`	Invalid length was specified.
`DFE_NOTENOUGH`	Not enough space for the data.
`DFE_NOVALS`	Values were not available.
`DFE_ARGS`	Invalid arguments passed to the routine.
`DFE_INTERNAL`	Serious internal error.
`DFE_NORESET`	Too late to modify this value.
`DFE_GENAPP`	Generic application level error.
`DFE_UNINIT`	Interface was not initialized correctly.
`DFE_CANTINIT`	Cannot initialize the interface the operation requires.
`DFE_CANTSHUTDOWN`	Cannot shut down the interface the operation requires.
`DFE_BADDIM`	Negative number of dimensions, or zero dimensions, was specified.
`DFE_BADFP`	File contained an illegal floating point number.
`DFE_BADDATATYPE`	Unknown or unavailable data type was specified.
`DFE_BADMCTYPE`	Unknown or unavailable machine type was specified.
`DFE_BADNUMTYPE`	Unknown or unavailable number type was specified.
`DFE_BADORDER`	Unknown or illegal array order was specified.
`DFE_RANGE`	Improper range for attempted access.
`DFE_BADCONV`	Invalid data type conversion was specified.
`DFE_BADTYPE`	Incompatible types were specified.
`DFE_BADSCHEME`	Unknown compression scheme was specified.
`DFE_BADMODEL`	Invalid compression model was specified.
`DFE_BADCODER`	Invalid compression encoder was specified.
`DFE_MODEL`	Error in the modeling layer of the compression operation.
`DFE_CODER`	Error in the encoding layer of the compression operation.
`DFE_CINIT`	Error in encoding initialization.
`DFE_CDECODE`	Error in decoding compressed data.
`DFE_CENCODE`	Error in encoding compressed data.
`DFE_CTERM`	Error in encoding termination.
`DFE_CSEEK`	Error seeking in an encoded data set.
`DFE_MINIT`	Error in modeling initialization.
`DFE_COMPINFO`	Invalid compression header.
`DFE_CANTCOMP`	Cannot compress an object.
`DFE_CANTDECOMP`	Cannot decompress an object.
`DFE_NOENCODER`	Encoder not available.
`DFE_NOSZLIB`	SZIP library not available.
`DFE_COMPVERSION`	Version error from zlib Note: when `Z_VERSION_ERROR` (-6) returned from zlib.

Error Code	Code Definition
`DFE_READCOMP`	Error in reading compressed data. Note: when one of the following error codes returned from zlib: `Z_ERRNO` (-1) `Z_STREAM_ERROR` (-2) `Z_DATA_ERROR` (-3) `Z_MEM_ERROR` (-4) `Z_BUF_ERROR` (-5)
`DFE_NODIM`	A dimension record was not associated with the image.
`DFE_BADRIG`	Error processing a RIG.
`DFE_RINOTFOUND`	Cannot find raster image.
`DFE_BADATTR`	Invalid attribute.
`DFE_BADTABLE`	The nsdg table has incorrect information.
`DFE_BADSDG`	Error in processing an SDG.
`DFE_BADNDG`	Error in processing an NDG.
`DFE_VGSIZE`	Too many elements in the vgroup.
`DFE_VTAB`	Element not in `vtab[]`.
`DFE_CANTADDELEM`	Cannot add the tag/reference number pair to the vgroup.
`DFE_BADVGNAME`	Cannot set the vgroup name.
`DFE_BADVGCLASS`	Cannot set the vgroup class.
`DFE_BADFIELDS`	Invalid fields string passed to vset routine.
`DFE_NOVS`	Cannot find the vset in the file.
`DFE_SYMSIZE`	Too many symbols in the users table.
`DFE_BADATTACH`	Cannot write to a previously attached vdata.
`DFE_BADVSNAME`	Cannot set the vdata name.
`DFE_BADVSCLASS`	Cannot set the vdata class.
`DFE_VSWRITE`	Error writing to the vdata.
`DFE_VSREAD`	Error reading from the vdata.
`DFE_BADVH`	Error in the vdata header.
`DFE_VSCANTCREATE`	Cannot create the vdata.
`DFE_VGCANTCREATE`	Cannot create the vgroup.
`DFE_CANTATTACH`	Cannot attach to a vdata or vset.
`DFE_CANTDETACH`	Cannot detach a vdata or vset with write access.
`DFE_BITREAD`	A bit read error occurred.
`DFE_BITWRITE`	A bit write error occurred.
`DFE_BITSEEK`	A bit seek error occurred.
`DFE_TBBTINS`	Failed to insert the element into tree.
`DFE_BVNEW`	Failed to create a bit vector.
`DFE_BVSET`	Failed when setting a bit in a bit vector.
`DFE_BVGET`	Failed when getting a bit in a bit vector.
`DFE_BVFIND`	Failed when finding a bit in a bit vector.

CHAPTER 14 -- HDF Performance Issues

14.1. Chapter Overview and Introduction

This chapter describes many of the concepts the HDF user should understand to gain better performance from their applications that use the HDF library. It also covers many of the ways in which HDF can be used to cause impaired performance and methods for correcting these problems.

As stated earlier in this manual, HDF has been designed to be very general-purpose, and it has been used in many different applications involving scientific data. Each application has its own set of software and hardware resource constraints that will affect performance in a different way, and to a different extent, from the resource constraints in other applications.

Therefore, it is impossible to outline *all* of the performance issues that may relate to a particular application of HDF. However, this chapter should give the reader sufficient knowledge of the most common performance issues encountered by the HDF Group. This knowledge should enable the reader to explore different ways of storing data on the platforms they use for the purpose of increasing library performance.

14.2. Examples of HDF Performance Enhancement

In this section, four pairs of HDF object models along with their C implementations will be presented. Each pair will illustrate a specific aspect of HDF library performance as it relates to scientific data sets. They will be employed here as general pointers on how to model scientific data sets for optimized performance.

In developing and testing these examples, the Sun Solaris OS version supported by HDF version 4.1 release 1 was used. Version 2.0 of the Quantify performance profiler was used to measure the relative differences in library performance between the SDS models in each pair. It should be noted that, while the examples reliably reflect which SDS configurations result in better performance, the specifics of how much performance will be improved depend on many factors such as OS configuration, compiler used and profiler used. Therefore, any specific measurements of performance mentioned in the chapter should be interpreted only as general indicators.

The reader should keep in mind that the following examples have been designed for illustrative purposes only, and should not be considered as real-world examples. It is expected that the reader will apply the library performance concepts covered by these examples to their specific usage of the HDF library.

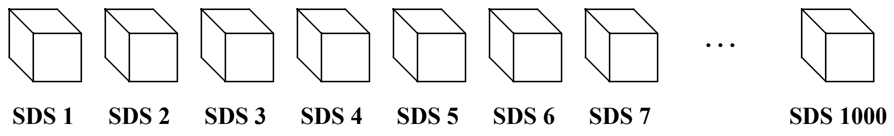
14.2.1. One Large SDS versus Several Smaller SDSs

The scientific data set is an example of what in HDF parlance is referred to as a ***primary object***. The primary objects accessed and manipulated by the HDF library include, beside scientific data sets, raster images, annotations, vdatas and vgroups. Each primary object has ***metadata***, or data describing the data, associated with it. Refer to the *HDF Specifications Manual* for a description of the components of this metadata and how to calculate its size on disk.

An opportunity for performance enhancement can exist when the size of the metadata far exceeds the size of the data described by the metadata. In this situation, more CPU time and disk space will be used to maintain the metadata than the data contained in the SDS. Consolidating the data into fewer, or even one, SDS can increase performance.

To illustrate this, consider 1,000 1 *x* 1 *x* 1 element scientific data sets of 32-bit floating-point numbers. No user-defined dimension, dimension scales or fill values have been defined or created.

FIGURE 14a **1,000 1 x 1 x 1 Element Scientific Data Sets**



In this example, 1,000 32-bit floating-point numbers are first buffered in-core, then written as 1,000 SDSs.

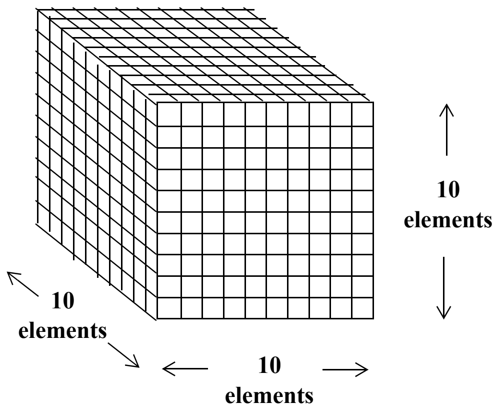
In Table 14A, the results of this operation are reflected in two metrics: the total number of CPU cycles used by the example program, and the size of the HDF file after the write operation.

TABLE 14A **Results of the Write Operation to 1,000 1x1x1 Element Scientific Data Sets**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now the 1,000 32-bit floating point numbers that were split into 1,000 SDSs are combined into one 10 *x* 10 *x* 10 element SDS. This is illustrated in the following figure.

FIGURE 14b **One 10 x 10 x 10 Element Scientific Data Set**



As with the last example, 1,000 32-bit floating-point numbers are first buffered in-core, then written to a single SDS. The following table contains the performance metrics of this operation.

TABLE 14B **Results of the Write Operation to One 10x10x10 Element Scientific Data Set**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
205,201	7,258

It is apparent from these results that merging the data into one scientific data set results in a substantial increase in I/O efficiency - in this case, a 99.9% reduction in total CPU load. In addition, the size of the HDF file is dramatically reduced by a factor of more than 100, even through the amount of SDS data stored is the same.

The extent to which the data consolidation described in this section should be done is dependent on the specific I/O requirements of the HDF user application.

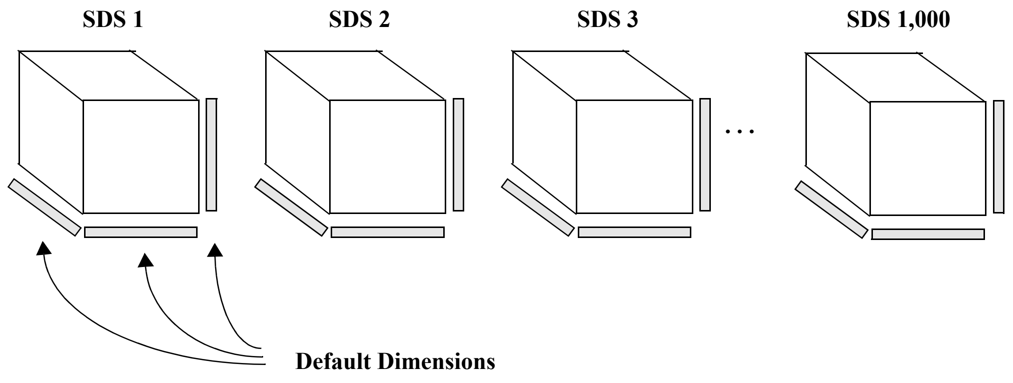
14.2.2. Sharing Dimensions between Scientific Data Sets

When several scientific data sets have dimensions of the same length, name and data type, they can share these dimensions to reduce storage overhead and CPU cycles in writing out data.

To illustrate this, again consider the example of 1,000 1 *x* 1 *x* 1 scientific data sets of 32-bit floating point numbers. Three dimensions are attached by default to each scientific data set by the HDF library. The HDF library assigns each of these dimensions a default name prefaced by the string *fakeDim*. See Chapter 3, *Scientific Data Sets (SD API)*, for a specific explanation of default dimension naming conventions.

FIGURE 14c

1,000 1 x 1 x 1 Element Scientific Data Sets



One 32-bit floating point number is written to each scientific data set. The following table lists the performance metrics of this operation.

TABLE 14C

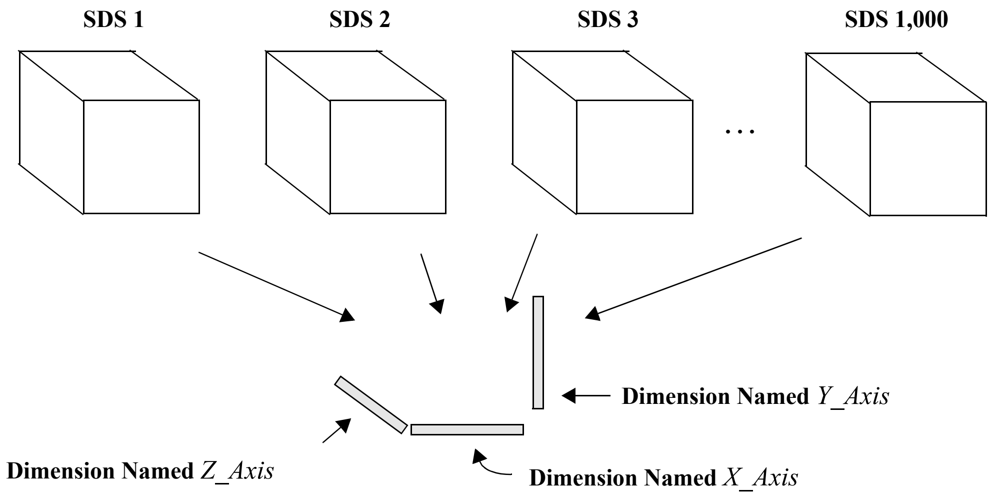
Results of the Write Operation to 1,000 1x1x1 Element Scientific Data Sets

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now consider the 1,000 SDSs described previously in this section. In this case, the 1,000 SDSs share the program-defined *X_Axis*, *Y_Axis* and *Z_Axis* dimensions as illustrated in the following figure.

FIGURE 14d

1,000 1 x 1 x 1 Element Scientific Data Sets Sharing Dimensions



The performance metrics that result from writing one 32-bit floating-point number to each dataset are in the following table.

TABLE 14D **Results of the Write Operation to 1,000 1x1x1 SDSs with Shared Dimensions**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
24,724,384	177,172

An 82% performance improvement in this example program can be seen from the information in this table, due to the fewer write operations involved in writing dimension data to shared dimensions. Also, the HDF file is significantly smaller in this case, due to the smaller amount of dimension data that is written.

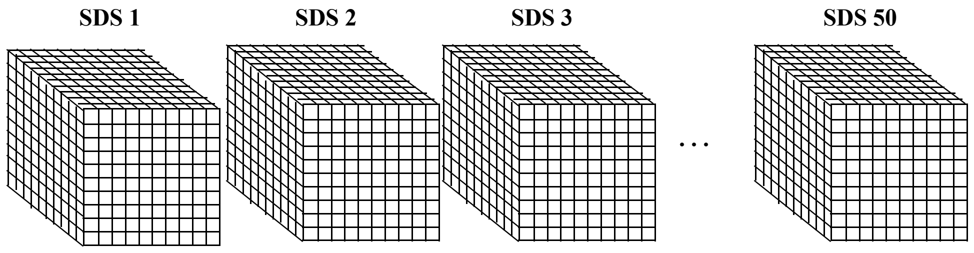
14.2.3. Setting the Fill Mode

When a scientific data set is created, the default action of the HDF library is to fill every element with the default fill value. This action can be disabled, and reenabled once it has been disabled, by a call to the **SDsetfillmode** routine.

The library's default writing of fill values can degrade performance when, after the fill values have been written, every element in the dataset is written to again. This operation involves writing every element in the SDS twice. This section will demonstrate that disabling the initial fill value write operation by calling **SDsetfillmode** can improve library performance.

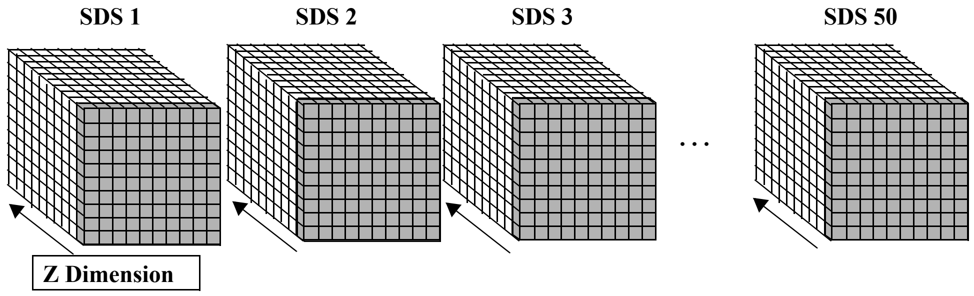
Consider 50 10 *x* 10 *x* 10 scientific data sets of 32-bit floating-point numbers.

FIGURE 14e **50 10 x 10 x 10 Element Scientific Data Sets**



By default, the fill value is written to every element in all 50 SDSs. The contents of a two-dimensional buffer containing 32-bit floating-point numbers is then written to these datasets. The way these two-dimensional slices are written to the three-dimensional SDSs is illustrated in the following figure. Each slice (represented by each shaded area in the figure) is written along the third dimension of each SDS, or if the dimensions are related to a Cartesian grid, the z-dimension, until the entire SDS is filled.

FIGURE 14f **Writing to the 50 10 x 10 x 10 Element Scientific Data Sets**



It should be noted that the reason each SDS is not rewritten to in one write operation is because the HDF library will detect this and automatically disable the initial write of the fill values as a performance-saving measure. Hence, the partial writes in two-dimensional slabs.

The following table shows the number of CPU cycles needed in our tests to perform this write operation with the fill value write enabled. The "Size of the HDF File" metric has been left out of this table, because it will not change substantially regardless of whether the default fill value write operation is enabled.

TABLE 14E **Results of the Write Operation to the 50 10x10x10 SDSs with the Fill Value Write Enabled**

Total Number of CPU Cycles
584,956,078

The following table shows the number of CPU cycles needed to perform the same write operation with the fill value write disabled.

TABLE 14F **Results of the Write Operation to the 50 SDSs with the Fill Value Write Disabled**

Total Number of CPU Cycles
390,015,933

The information in these tables demonstrate that eliminating the I/O overhead of the default fill value write operation when an entire SDS is rewritten to results in a substantial reduction of the CPU cycles needed to perform the operation -- in this case, a reduction of 33%.

14.2.4. Disabling *Fake* Dimension Scale Values in Large One-dimensional Scientific Data Sets

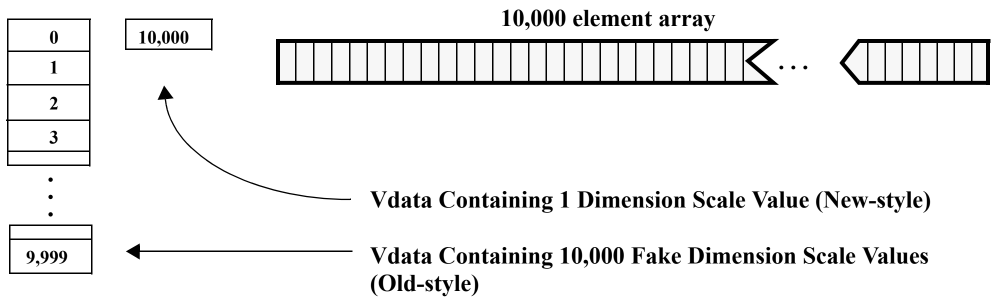
In versions 4.0 and earlier of the HDF library, dimension scales were represented by a vgroup containing a vdata. This vdata consisted of as many records as there are elements along the dimension. Each record contained one number which represented each value along the dimension scale, and these values are referred to as ***fake*** dimension scale values.

In HDF version 4.0 a new representation of the dimension scale was implemented alongside the old one -- a vdata containing only one value representing the total number of values in the dimension scale. In version 4.1 release 2, this representation was made the default. A ***compatible*** mode is also supported where both the older and newer representations of the dimension scale are written to file.

In the earlier representation, a substantial amount of I/O overhead is involved in writing the fake dimension scale values into the vdata. When one of the dimensions of the SDS array is very large, performance can be improved, and the size of the HDF file can be reduced, if the old representation of dimension scales is disabled by a call to the **SDsetdimval_comp** routine. The examples in this section will illustrate this.

First, consider one 10,000 element array of 32-bit floating point numbers, as shown in the following figure. Both the new and old dimension scale representations are enabled by the library.

FIGURE 14g **One 10,000 Element Scientific Data Set with Old- and New-Style Dimension Scales**


0
1
2
3
9,999
10,000
10,000 element array
Vdata Containing 1 Dimension Scale Value (New-style)
Vdata Containing 10,000 Fake Dimension Scale Values (Old-style)

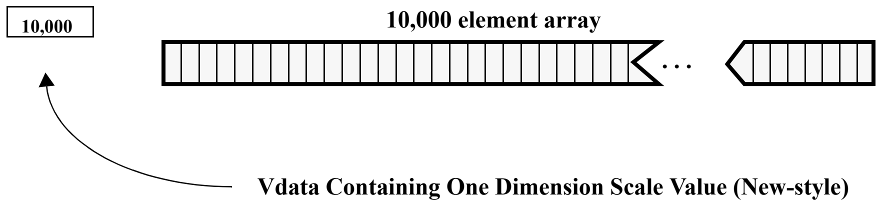
10,000 32-bit floating-point numbers are buffered in-core, then written to the scientific data set. In addition, 10,000 integers are written to the SDS as dimension scale values. The following table contains the results of this operation from our tests.

TABLE 14G **Results of the SDS Write Operation with the New and Old Dimension Scales**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
439,428	82,784

Now consider the same SDS with the fake dimension scale values disabled. The following figure illustrates this.

FIGURE 14h **One 10,000 Element Scientific Data Set with the Old-Style Dimension Scale Disabled**



The following table contains the performance metrics of this write operation.

TABLE 14H **Results of the SDS Write Operation with Only the New Dimension Scale**

Total Number of CPU Cycles	Size of the HDF File
318,696	42,720

The old-style dimension scale is not written to the HDF file, which results in the size of the file being reduced by nearly 50%. There is also a marginal reduction in the total number of CPU cycles.

14.3. Data Chunking

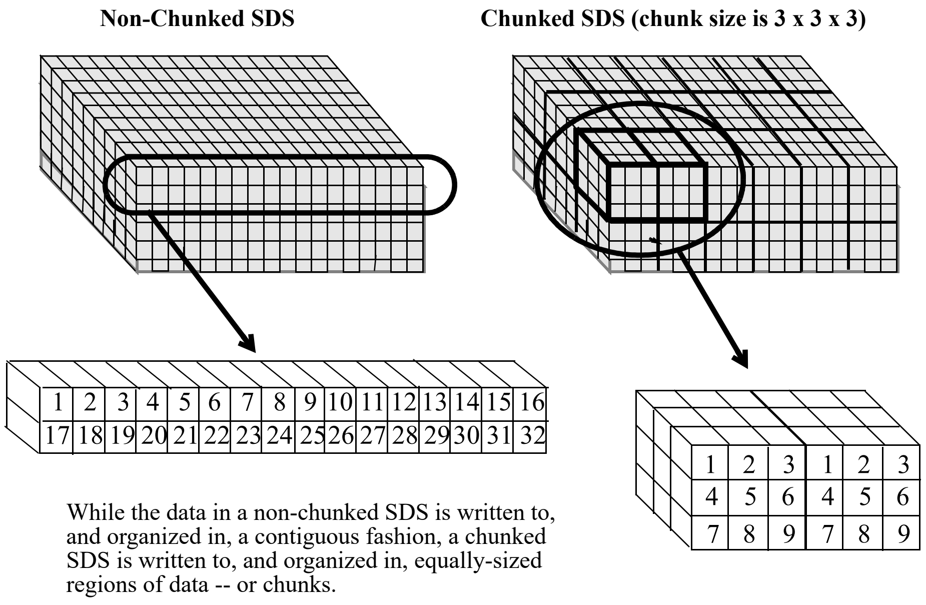
14.3.1. What Is Data Chunking?

Data chunking is a method of organizing data within an SDS where data is stored in ***chunks*** of a predefined size, rather than contiguously by array element. Its two-dimensional instance is sometimes referred to as ***data tiling***. Data chunking is generally beneficial to I/O performance in very large arrays, e.g., arrays with thousands of rows and columns.

If correctly applied, data chunking may reduce the number of seeks through the SDS data array to find the data to be read or written, thereby improving I/O performance. However, it should be remembered that data chunking, if incorrectly applied, can significantly *reduce* the performance of reading and/or writing to an SDS. Knowledge of how chunked SDSs are created and accessed and application-specific knowledge of how data is to be read from the chunked SDSs are necessary in avoiding situations where data chunking works against the goal of I/O performance optimization.

The following figure illustrates the difference between a non-chunked SDS and a chunked SDS.

FIGURE 14i **Comparison between Chunked and Non-chunked Scientific Data Sets**


Non-Chunked SDS
Chunked SDS (chunk size is 3 x 3 x 3)
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 31 32
1 2 3 1 2 3
4 5 6 4 5 6
7 8 9 7 8 9
While the data in a non-chunked SDS is written to, and organized in, a contiguous fashion, a chunked SDS is written to, and organized in, equally-sized regions of data -- or chunks.

14.3.2. Writing Concerns and Reading Concerns in Chunking

There are issues in working with chunks that are related to the reading process and others that are related to the writing process.

Specifically, the issues that affect the process of reading from chunked SDSs are

- Compression
- Subsetting
- Chunk sizing
- Chunk cache sizing

The issues that affect the process of writing to chunked SDSs are

- Compression
- Chunk cache sizing

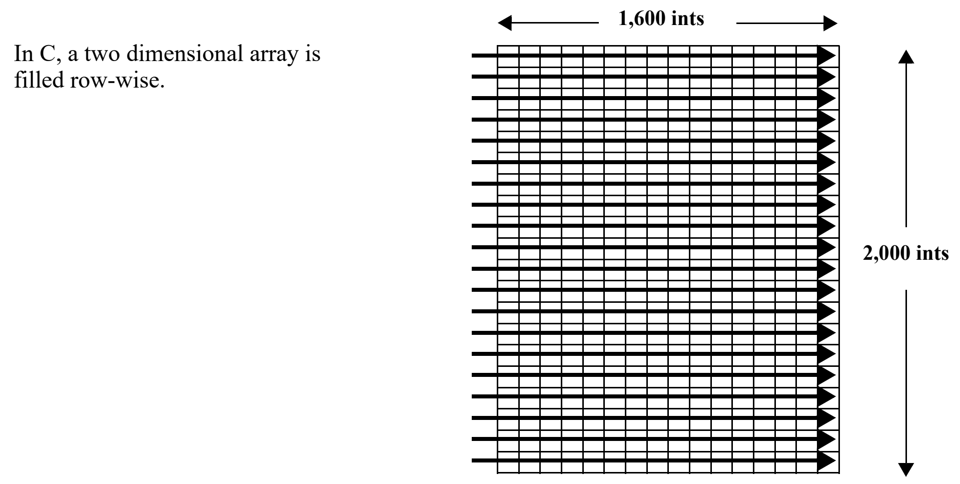
14.3.3. Chunking without Compression

Accessing Subsets According to Storage Order

The main consideration to keep in mind when subsetting from chunked and non-chunked SDSs is that if the subset can be accessed in the same order as it was stored, subsetting will be efficient. If not, subsetting may result in less-than-optimal performance considering the number of elements to be accessed.

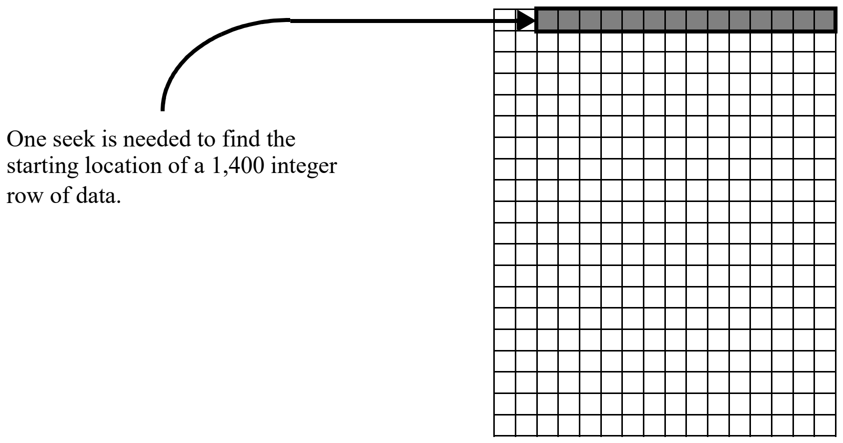
To illustrate this, the instance of subsetting in non-chunked SDSs will first be described. Consider the example of a non-chunked, two-dimensional, 2,000 *x* 1,600 SDS array of integer data. The following figure shows how this array is filled with data in a row-wise fashion. (Each square in the array shown represents 100 *x* 100 integers.)

FIGURE 14j **Filling a Two-dimensional Array with Data Using Row-major Ordering**



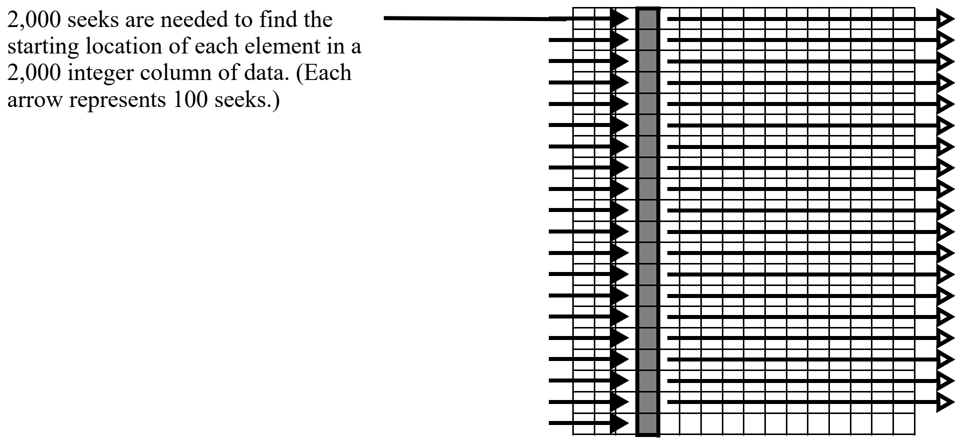
The most efficient way an application can read a row of data, or a portion of a row, from this array, is a contiguous, row-wise read of array elements. This is because this is the way the data was originally written to the array. Only one seek is needed to perform this. (See Figure 14k)

FIGURE 14k **Number of Seeks Needed to Access a Row of Data in a Non-chunked SDS**


One seek is needed to find the starting location of a 1,400 integer row of data.

If the subset of data to be read from this array is one 2,000 integer *column*, then 2,000 seeks will be required to complete the operation. This is the most inefficient method of reading this subset as nearly all of the array locations will be accessed in the process of seeking to a relatively small number of target locations.

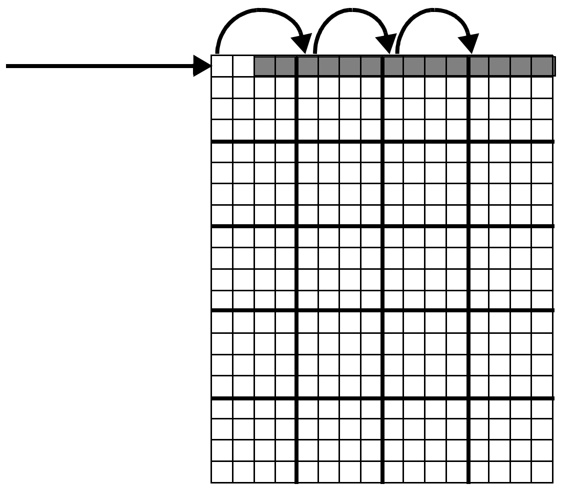
FIGURE 14l **Number of Seeks Needed to Access a Column of Data in a Non-chunked SDS**


2,000 seeks are needed to find the starting location of each element in a 2,000 integer column of data. (Each arrow represents 100 seeks.)

Now suppose this SDS is chunked, and the chunk size is 400 *x* 400 integers. A read of the aforementioned row is performed. In this case, four seeks are needed to read all of the chunks that contain the target locations. This is less efficient than the one seek needed in the non-chunked SDS.

FIGURE 14m **Number of Seeks Needed to Access a Row of Data in a Chunked SDS**

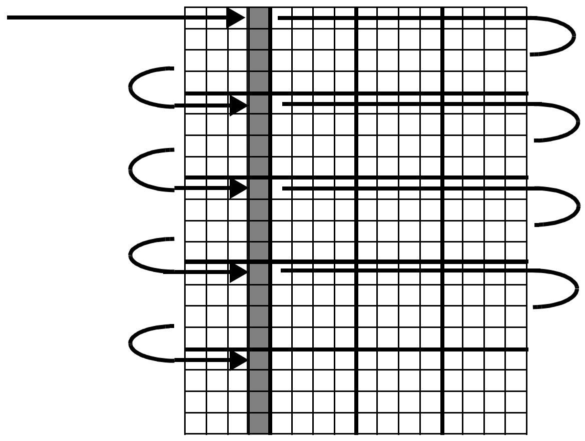
4 seeks are needed to find the starting location of a 1,400 integer row of data in a chunked data array with 400 x 400 integer chunks.



To read the aforementioned column of data, five chunks must be read into memory in order to access the 2,000 locations of the subset. Therefore, five seeks to the starting location of each of these chunks are necessary to complete the read operation, far fewer than the 2,000 needed in the non-chunked SDS.

FIGURE 14n **Number of Seeks Needed to Access a Column of Data in a Chunked SDS**

5 seeks are needed to find the starting location of a 2,000 integer column of data in a chunked data array with 400 x 400 integer chunks. (Each arrow represents one seek.)



These examples show that, in many cases, chunking can be used to reduce the I/O overhead of subsetting, but in certain cases, chunking can impair I/O performance.

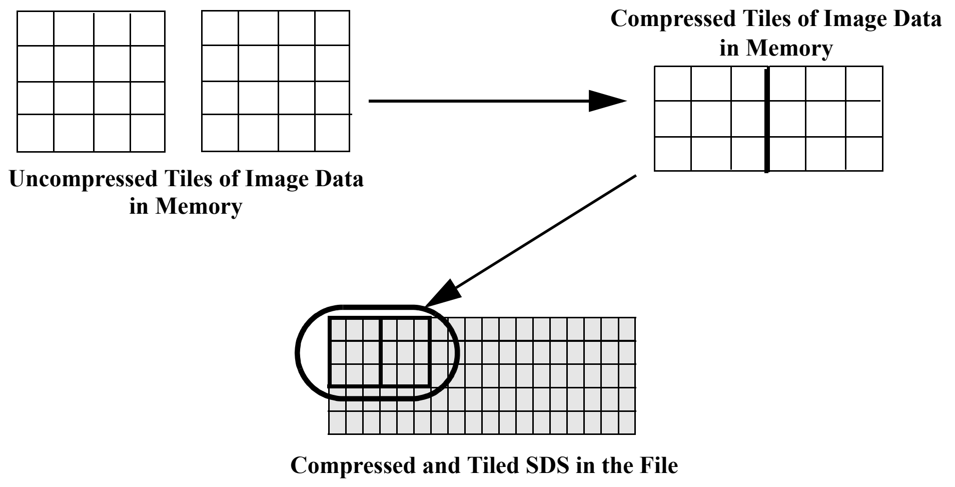
The efficiency of subsetting from chunked SDSs is partly determined by the size of the chunk: the smaller the chunk size, the more seeks will be necessary. Chunking can substantially improve I/O performance when data is read along the slowest-varying dimension. It can substantially degrade performance when data is read along the fastest-varying dimension.

14.3.4. Chunking with Compression

Chunking can be particularly effective when used in conjunction with compression. It allows subsets to be read (or written) without having to uncompress (or compress) the entire array.

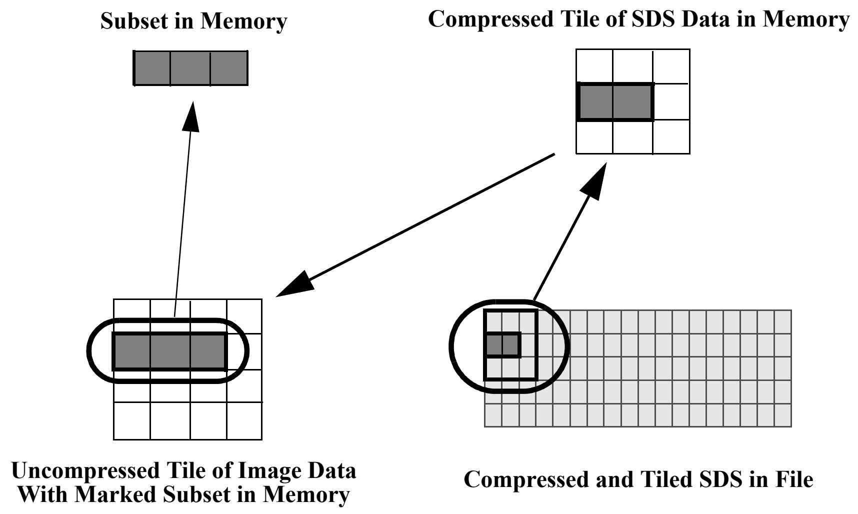
Consider the example of a tiled, two-dimensional SDS containing one million bytes of image data. Each tile of image data has been compressed as illustrated in the following figure.

FIGURE 14o **Compressing and Writing Chunks of Data to a Compressed and Tiled SDS**


Compressed Tiles of Image Data in Memory
Uncompressed Tiles of Image Data in Memory
Compressed and Tiled SDS in the File

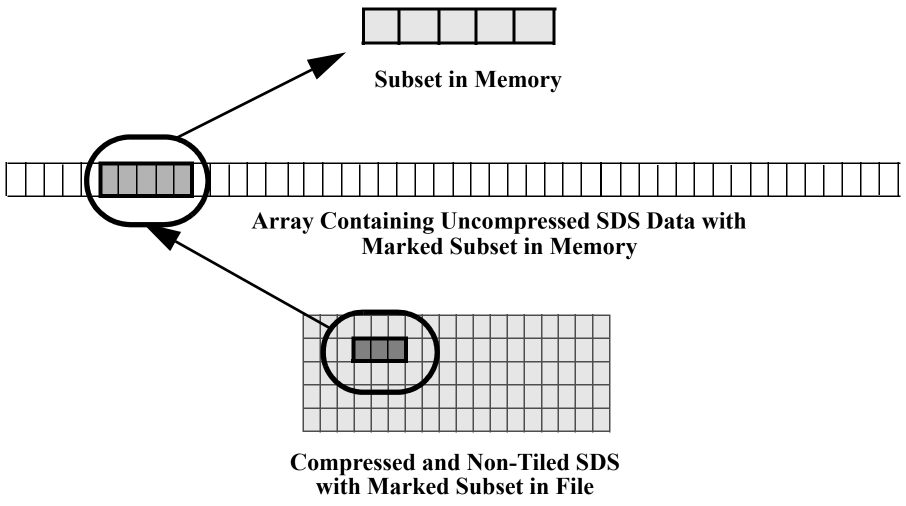
When it becomes necessary to read a subset of the image data, the application passes in the location of a tile, reads the entire tile into a buffer, and extracts the data-of-interest from that buffer.

FIGURE 14p **Extracting a Subset from a Compressed and Tiled SDS**


Subset in Memory
Compressed Tile of SDS Data in Memory
Uncompressed Tile of Image Data With Marked Subset in Memory
Compressed and Tiled SDS in File

In a compressed and *non-tiled* SDS, retrieving a subset of the compressed image data necessitates reading the entire contents of the SDS array into a memory buffer and uncompressing it in-core. (See Figure 14q) The subset is then extracted from this buffer. (Keep in mind that, even though the illustrations show two-dimensional data tiles for clarity, this process can be extended to data chunks of any number of dimensions.)

FIGURE 14q **Extracting a Subset from a Compressed Non-tiled SDS**


Subset in Memory
Array Containing Uncompressed SDS Data with Marked Subset in Memory
Compressed and Non-Tiled SDS with Marked Subset in File

As compressed image files can be as large as hundreds of megabytes in size, and a gigabyte or more uncompressed, it is clear that the I/O requirements of reading to and writing from non-tiled, compressed SDSs can be immense, if not prohibitive. Add to this the additional I/O burden inherent in situations where portions of several image files must be read at the same time for comparison, and the benefits of tiling become even more apparent.

NOTE: It is recommended that the **SDwritechunk** routine be used to write to a compressed and chunked SDS. **SDwritechunk** can perform this operation more efficiently than the combination of **SDsetcompress** and **SDwritedata**. This is because the chunk information provided by the user to the **SDwritechunk** routine must be retrieved from the file by **SDwritedata**, and therefore involves more computational overhead.

14.3.5. Effect of Chunk Size on Performance

The main concern in modelling data for chunking is that the chunk size be approximately equal to the average expected size of the data block needed by the application.

If the chunk size is substantially larger than this, increased I/O overhead will be involved in reading the chunk and increased performance overhead will be involved in the decompression of the data if it is compressed. If the chunk size is substantially smaller than this, increased performance and memory/disk storage overhead will be involved in the HDF library's operations of accessing and keeping track of more chunks, as well as the danger of exceeding the maximum number of chunks per file. (64K)

It is recommended that the chunk size be at least 8K bytes.

14.3.6. Insufficient Chunk Cache Space Can Impair Chunking Performance

The HDF library provides caching chunks. This can substantially improve I/O performance when a particular chunk must be accessed more than once.

There is a potential performance problem when subsets are read from chunked datasets and insufficient chunk cache space has been allocated. The cause of this problem is the fact that two separate levels of the library are working to read the subset into memory and these two levels have a different perspective on how the data in the dataset is organized.

Specifically, higher-level routines like **SDreaddata** access the data in a strictly row-wise fashion, not according to the chunked layout. However, the lower-level code that directly performs the read operation accesses the data according to the chunked layout.

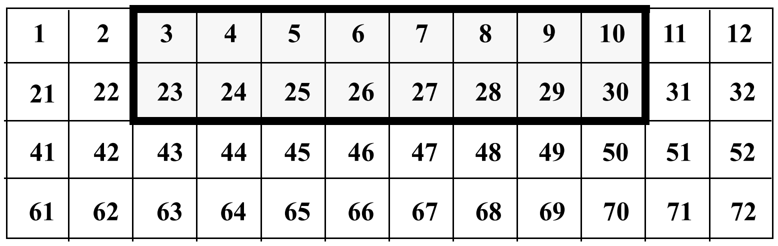
As an illustration of this, consider the 4 *x* 12 dataset depicted in the following figure.

FIGURE 14r **Example 4 x 12 Element Scientific Data Set**

1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

Suppose this dataset is untiled, and the subset shown in the following figure must be read.

FIGURE 14s **2 x 8 Element Subset of the 4 x 12 Scientific Data Set**



1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

As this dataset is untiled, the numbers are stored in linear order. **SDreaddata** finds the longest contiguous stream of numbers, and requests the lower level of the library code to read it into memory. First, the first row of numbers will be read:

```
3  4  5  6  7  8  9 10
```

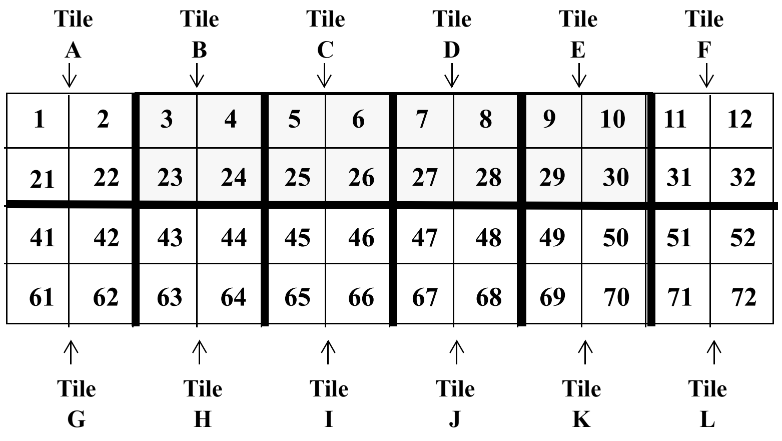
Then the second row:

```
23 24 25 26 27 28 29 30
```

This involves two reads, two disk accesses and sixteen numbers.

Now suppose that this dataset is tiled with 2 *x* 2 element tiles. On the disk, the data in this dataset is stored as twelve separate tiles, which for the purposes of this example will be labelled A through L.

FIGURE 14t **4 x 12 Element Data Set with 2 x 2 Element Tiles**



Also, the chunk cache size is set to 2.

A request is made to read the aforementioned subset of numbers into memory. As before, **SDreaddata** will determine the order the numbers will be read in. The routine has no information about the tiled layout. The higher-level code will again request the values in the first rows of tiles B through E from the lower level code on the first read operation.

In order to access those numbers the lower level code must read in four tiles: B, C, D, E. It reads in tiles B and C, retrieving the values `3`, `4`, `5`, and `6`. However, as the cache space is now completely filled, it must overwrite tile B in the cache to access the values `7 and 8`, which are in tile D. It then has to overwrite tile C to access the values `9 and 10`, which are in tile E. Note that, in each case, half of the values from the tiles that are read in are unused, even though those values will be needed later.

Next, the higher-level code requests the second row of the subset. The lower-level code must *reread* tile B to access the values `23` and `24`. But tile B is no longer in the chunk cache. In order to access tile B, the lower-level code must overwrite tile D, and so on. By the time the subset read operation is complete, it has had to read in each of the tiles twice. Also, it has had to perform 8 disk accesses and has read 32 values.

Now consider a more practical example with the following parameters:

- A scientific data set has 3,000 rows and 8,400 columns.
- The target subset is 300 rows by 1,000 columns, and contains 300,000 numbers.

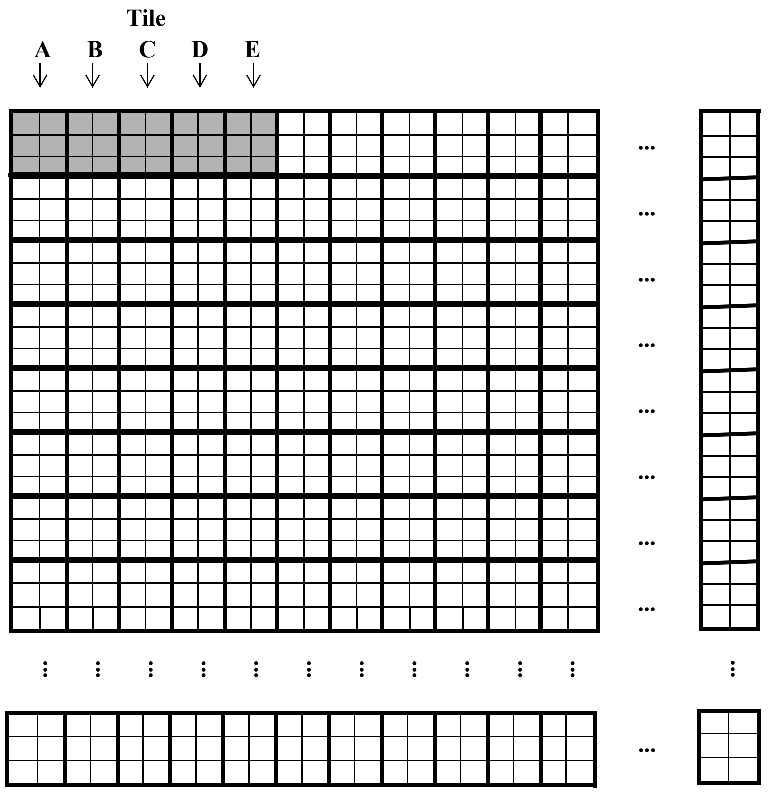
If the dataset is untiled the numbers are read into memory row-by-row. This involves 300 disk accesses for 300 rows, with each disk access reading in 1,000 numbers. The total number of numbers that will be read is 300,000.

Suppose the dataset is tiled as follows:

- The tile size is 300 rows by 200 columns, or 60,000 numbers.
- The size of the chunk cache is 2.

Each square in the following figure represents one 100 *x* 100 element region of the dataset. Five tiles span the 300 *x* 1,000 target subset. For the purposes of this example, they will be labelled A, B, C, D and E.

FIGURE 14u **5 200 x 300 Element Tiles Labelled A, B, C, D and E**



First, the higher-level code instructs the lower-level code to read in the first row of subset numbers. The lower-level code must read all five tiles (A through E) into memory, as they all contain numbers in the first row. Tiles A and B are read into the cache without problem, then the following set of cache overwrites occurs.

1. Tile A is overwritten when tile C is read.
2. Tile B is overwritten when tile D is read.
3. Tile C is overwritten when tile E is read.

When the first row has been read, the cache contains tiles D and E.

The second row is then read. The higher-level code first requests tile A, however the cache is full, so it must overwrite tile D to read tile A. Then the following set of cache overwrites occur.

1. Tile E is overwritten when tile B is read.
2. Tile A is overwritten when tile C is read.
3. Tile B is overwritten when tile D is read.
4. Tile C is overwritten when tile E is read.

For each row, five tiles must be read in. No actual caching results from this overwriting. When the subset read operation is complete, 300 * 5 = 1,500 tiles have been read, or 60,000 * 1,500 = 90,000,000 numbers.

Essentially, five times more disk accesses are being performed and 900 times more data is being read than with the untiled 3,000 *x* 8,400 dataset. The severity of the performance degradation increases in a non-linear fashion as the size of the dataset increases.

From this example it should be apparent that, to prevent this kind of chunk cache "thrashing" from occurring, the size of the chunk cache should be made equal to, or greater than, the number of chunks along the fastest-varying dimension of the dataset. In this case, the chunk cache size should be set to 4.

When a chunked SDS is opened for reading or writing, the default cache size is set to the number of chunks along the fastest-varying dimension of the SDS. This will prevent cache thrashing from occurring in situations where the user does not set the size of the the chunk cache. Caution should be exercised by the user when altering this default chunk cache size.

14.4. Block Size Tuning Issues

A key to I/O performance in HDF is the number of disk accesses that must be made during any I/O operation. If you can decrease significantly the number of disk accesses required, you may be able to improve performance correspondingly. In this section we examine two such strategies for improving HDF I/O performance.

14.4.1. Tuning Data Descriptor Block Size to Enhance Performance

HDF objects are identified in HDF files by 12-byte headers called data descriptors (DDs). Most composite HDF objects, such as SDSs, are made up of many small HDF objects, so it is not unusual to have a large number of DDs in an HDF file. DDs are stored in blocks called data descriptor blocks (DD blocks).

When an HDF file is created, the file's DD block size is specified. The default size is 16 DDs per DD block. When you start putting objects into an HDF file, their DDs are inserted into the first DD block. When the DD block gets filled up, a new DD block is created, stored at some other location in the file, and linked with the previous DD block. If a large number of objects are stored in an HDF file whose DD block size is small, a large number of DD blocks will be needed, and each DD block is likely to be stored on a different disk page.

Consider, for example, an HDF file with 1,000 SDSs and a DD block size of 16. Each SDS could easily require 10 DDs to describe all the objects comprising the SDS, so the entire file might contain 10,000 DDs. This would require 625 (10,000/16) DD blocks, each stored on a different disk page.

Whenever an HDF file is opened, all of the DDs are read into memory. Hence, in our example, 625 disk accesses might be required just to open the file.

Fortunately, there is a way we can use this kind of information to improve performance. When we create an HDF file, we can specify the DD block size. If we know that the file will have many objects stored in it, we should choose a large DD block size so that each disk access will read in a large number of DDs, and hence there will be fewer disk accesses. In our example, we might have chosen the DD block size to be 10,000, resulting in only one disk access. (Of course, this example goes deliberately to a logical extreme. For a variety of reasons, a more common approach would be to set the DD block size to something between 1,000 and 5,000 DDs.)

From this discussion we can derive the following rules of thumb for achieving good performance by altering the DD block size.

- Increasing the size of the data descriptor block may improve performance when opening a file, especially when working with large HDF files with lots of objects. It will reduce the number of times that HDF has to go out and read another DD block. This will be particularly valuable in code that does large numbers of HDF file opens.

- The same principle applies when closing an HDF file that has been written to. Since all DDs are flushed to an HDF file when it is written to and then closed, the DD block size can similarly impact performance.
- Notice that these actions only affect the opening and closing of a file. Once a file is opened, DDs are accessed in memory; no further disk accesses are required.
- Large DD blocks can negatively affect storage efficiency, particularly if very large DD blocks are used. Since the last DD block may only be partially filled up, you probably should not use large DD blocks for very small HDF files.

14.4.2. Tuning Linked Block Size to Enhance Performance

Linked blocks get created whenever compression, chunking, external files, or appendable datasets are used. They provide a means of linking new data blocks to a pre-existing data element. If you have ever looked at an HDF file and seen `Special Scientific Data` or `Linked Block Indicator` tags with strange tag values, these are used in specifying linked blocks. As with DD blocks, linked block size can affect both storage efficiency and I/O performance.

You can change the linked block size for SDSs by use of the function **SDsetblocksize**. To change the linked block size for Vdatas, prior to version 4.1r5, you must edit the `hlimits.h` file, change the value of `HDF_APPENDABLE_BLOCK_LEN`, and re-build the HDF library. However, starting in version 4.1r5, applications can use the public function **VSsetblocksize** for the same purpose. Changing the linked block size only affects the size of the linked blocks used *after* the change is made; it does not affect the size of blocks that have already been written.

There is a certain amount of overhead when creating linked blocks. For every linked block that is added there will be a specified number of block accesses, disk space used, and reference numbers added to the file. If you increase the size of the linked block, it will decrease the number of block accesses, disk space used, and reference numbers added to the file. Making the linked block size larger will decrease the number of reference numbers required; this is sometimes necessary because there are a limited number of available reference numbers.

Linked block size can also affect I/O performance, depending on how the data is accessed. If the data will typically be accessed in large chunks, then making the linked block size large could improve performance. If the data is accessed in small chunks, then making the linked block size small could improve performance.

If data will be randomly accessed in small amounts, then it is better to have small linked blocks.

Ideally one might say that making the linked block size equal to the size of the dataset that will typically be accessed, is the best solution. However, there are other things that will affect performance, such as the operating system being used, the sector size on the disk being accessed, the amount of memory available, and access patterns.

Here are some rules of thumb for specifying linked block size:

- Linked block size should be at least as large as the smallest number of bytes accessed in a single disk access. This amount varies from one system to another, but 4K bytes is probably a safe minimum.
- Linked block size should be a power of 2.
- Linked blocks should be approximately equal to the number of bytes accessed in a typical access. This rule should be mitigated by the amount of locality from one disk access to another, however, as the next rule indicates.
- If memory is large, it may be possible to take advantage of caching that your operating system does by using a large block size. If successive accesses are close to one another, blocks

may be cached by the OS, so that actual physical disk accesses are not always required. If successive accesses are not close to one another, this strategy could backfire, however.

- Although very large blocks can result in efficient access, they can also result in inefficient storage. For instance if the block size is 100K bytes, and 101K bytes of data are stored per SDS in an HDF file, the file will be twice as large as necessary.

Unfortunately, there are so many factors affected by block size that there is no simple formula that you can follow for deciding what the linked block size should be. A little experimentation on the target platform can help a great deal in determining the ideal block size for your situation.

14.4.3. Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance

In some circumstances, repeatedly appending to unlimited dimension data sets can lead to significant performance problems.

Each time data is appended to a Vdata or an unlimited dimension SDS, a new linked block may be created. Eventually, the linked block list may become so large that data seeking performance deteriorates substantially. In the worst case, one can exceed the allowable number of reference numbers, corrupting the HDF file.

In many such instances, increasing the linked block size (see Section 14.4.2. in this *User's Guide* or, for SDSs only, **SDsetblocksize/sfsblsz** in the *HDF Reference Manual* or DD block size (see Section 14.4.1.) will alleviate the reference number problems and improve performance.

CHAPTER 15 -- HDF Command-line Utilities

15.1. Chapter Overview

This chapter describes a number of command-line utilities that are available for working with HDF files.

The HDF command-line utilities are application programs that are executed from the UNIX shell prompt. These utilities serve the following needs of the HDF developer.

- They make it possible to perform, at the command line level, common operations on HDF files without having to resort to custom-programmed utilities to do these operations.
- They provide the capability for performing operations on HDF files that would be very difficult to do with custom-programmed utilities.

Table 15A lists the names and descriptions of the utilities described in this chapter.

TABLE 15A

The HDF Command-line Utilities

Utility Type	Name	Description
File content display tools	`hdp`	Also known as **HDF dumper**. Displays general information about the contents of an HDF file (Section "*Displaying the Contents of an HDF File: hdp (or HDF Dumper)"*)
	`hdiff`	Displays the differences between the contents of two HDF files (Section "*Comparing two HDF Files: hdiff"*)
	`vshow`	Displays vset information (Section "*Displaying Vdata Information: vshow"*)
Raw data to HDF conversions	`hdfimport`	Converts floating-point and/or integer data to HDF scientific data sets (SDS) and/or HDF 8-bit raster image sets (RIS8) format, storing the results in an HDF file (Section "*Converting Floating-point or Integer Data to SDS or RIS8: hdfimport"*) [This utility replaces `fp2hdf`.]
	`r8tohdf`	Converts one or more 8-bit raster images in raw format to the HDF RIS8 format and writes them to a file, optionally with palettes (Section "*Converting 8-Bit Raster Images to the HDF Format: r8tohdf"*)
	`r24hdf8`	Converts raw RGB 24-bit images to an RIS8 with a palette (Section "*Converting 24-Bit Raw Raster Images to RIS8 Images: r24hdf8"*)
	`paltohdf`	Converts a raw palette to the HDF format (Section "*Converting Raw Palette Data to the HDF Palette Format: paltohdf"*)
HDF to raw data conversions	`hdftor8`	Converts raster images and/or palettes from the HDF format to the raw format and stores them in two sets of files - one for images and the other for palettes (Section "*Extracting 8-Bit Raster Images and Palettes from HDF Files: hdftor8"*)
	`hdftopal`	Converts a palette in an HDF file to a raw palette format (Section "*Extracting Palette Data from an HDF File: hdftopal"*)
Raster 8 and 24 image operations	`ristosds`	Converts a set of RIS8 HDF files into a single three-dimensional SDS HDF file (Section "*Converting Several RIS8 Images to One 3D SDS: ristosds"*)
	`hdf24hdf8`	Converting an RIS24 HDF image to an RIS8 HDF image with a 256-color palette (Section "*Converting an HDF RIS24 Image to an HDF RIS8 Image: hdf24hdf8"*)
	`hdfcomp`	Compresses 8-bit raster images from an HDF file, storing them in a new HDF file (Section "*Compressing RIS8 Images in an HDF File: hdfcomp"*)
HDF file maintenance operations	`hdfpack`	Compresses an HDF file, reading all of the objects in the file and writing them to a new HDF file (Section "*Compressing an HDF File: hdfpack"*)
	`hrepack`	Performs a logical copy of an input HDF4 file to an output HDF4 file, copying all high level objects while optionally rewriting the objects with or without compression and/or with or without chunking (Section "*Reformatting an HDF File: hrepack"*)
	`vmake`	Creates vsets (Section "*Creating Vgroups and Vdatas: vmake"*)
Miscellaneous utilities	`hdfls`	Displays information about HDF data objects (Section "*Listing Basic Information about Data Objects in an HDF File: hdfls"*)
	`hdfed`	Displays the contents of an HDF file and allows limited manipulation of the data (Section "*Editing the Contents of an HDF File: hdfed"*)
HDF5 / HDF4 file conversion	`h4toh5, h5toh4, etc`	Tools to assist HDF5 users working with HDF4 files and HDF4 users working with HDF5 files (Section "*Working with Both HDF4 and HDF5 File Formats"*) (These tools are not included in this HDF4 distribution)
HDF-to-GIF and GIF-to-HDF conversion	`hdf2gif`	Converts an HDF file to a GIF file (Section "*Converting an HDF File to a GIF File: hdf2gif"*)
	`gif2hdf`	Converts a GIF file to an HDF file (Section "*Converting a GIF File to an HDF File: gif2hdf"*)
HDF4 Library configuration and management	`h4cc`	Simplifies the compilation of HDF4 programs written in C (Section "*Compiling C applications that Use HDF4: h4cc"*)
	`h4fc`	Simplifies the compilation of HDF5 programs written in Fortran90 (Section "*Compiling Fortran applications that Use HDF4: h4fc"*)
	`h4redeploy`	Updates HDF4 compiler tools after an HDF4 software installation in a new location (Section "*Updating HDF4 Compiler Tools after an Installation in a New Location: h4redeploy"*)

15.2. Displaying the Contents of an HDF File: hdp (or HDF Dumper)

15.2.1. General Description

The **hdp** utility, also known as the HDF dumper, provides quick and general information about all objects in the specified HDF file. It can list the contents of HDF files at various levels with different details. It can also dump the data of one or more specific objects in the file.

15.2.2. Command-line Syntax

```
hdp [[-H command] | [command]] filelist
```

The **hdp** option flags are described in Table 15B.

TABLE 15B **hdp Option Flags**

`-H`	Help:	Displays usage information about the specified command. If no command is listed, information about all commands are displayed.

Like **hdfed**, **hdp** provides a set of commands that allow the user to determine what kind of information is to be displayed.

TABLE 15C **The hdp Command Set**

Name	Description
`list`	Displays the contents of the HDF files in the specified format.
`dumpsds`	Displays the contents of the SDSs in the listed files.
`dumpgr`	Displays the contents of the raster images in the listed files.
`dumpvd`	Displays the contents of the vdata objects in the listed files.
`dumpvg`	Displays the contents of the vgroup objects in the listed files.
`dumprig`	Displays the contents of the RIGs in the listed files.

The **list** command

Syntax: `list [-acensldg] [-o<f|g|t|n>] [-t tag] filelist`

Flags:

`-a`	Print annotations of selected items. (Sets long output)
`-c`	Print classes of selected items. (Sets long output)
`-n`	Print names or labels of selected items. (Sets long output)
`-e`	Print special element information for selected items. (Sets long output)
`-s`	Set output to short format. (default)
`-l`	Set output to long format.
`-d`	Set output to debugging format.
`-g`	Display information for groups only.
`-t number`	Display information for objects with the given tag number.

`-t` *`name`*	Display information for objects with the given name.
`-of`	Print items in the order found in the file.
`-og`	Print items in group order.
`-ot`	Print items in tag order. (default)
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays the contents of the HDF files in the specified format. As with the **hdfed info** command, the listing for special elements will contain a special tag value (for `DFTAG_VS`, it is 18347) and the text `Unknown Tag`.

The **dumpsds** command

Syntax:

```
hdp dumpsds [-a | -i indices | -r refs | -n names ] [-v | -h | -d]
            [-o filename ] [-bx] filelist
```

Flags:

`-a`	Dump all SDSs in the file(s). (default)
`-k`	Dump chosen SDSs in the same order they were specified.
`-i` *`indices`*	Dump the SDSs at the positions listed in *`indices`*.
`-r` *`refs`*	Dump the SDSs with reference numbers listed in *`refs`*.
`-n` *`names`*	Dump the SDSs with names listed in *`names`*.
`-v`	Dump all SDS contents, including annotations. (default)
`-h`	Dump SDS header information only, no data or element annotations.
`-d`	Dump SDS data only, no tag/ref or header information. Output is formatted for input to **fp2hdf**.
`-c`	Print space characters as they are, not \<digit>.
`-g`	Do not print data of file (global) attributes.
`-l`	Do not print data of local attributes.
`-s`	Do not add carriage return to a long line, i.e. dump it as a stream.
`-o` *`filename`*	Print output to the file *`filename`*.
`-b`	Output in binary format.
`-x`	Output in ASCII format. (default)
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays SDS information and/or data in the specified format. The `-r`, `-i`, and `-n` flags can be selected together. When `-k` is specified, it must be in front of `-r`, `-i`, and `-n` to keep the order in which the SDSs are specified by those flags.

The **dumpgr** command

Syntax:

```
hdp dumpgr [-a | -i indices | -r refs | -n names] [-m] [-v | -h | -d
   | -p] [-c] [-g] [-l] [-s] [-o filename] [-bx] filelist
```

Flags:

`-a`	Dump all raster images (RIs) in the file(s). (default)
`-i` *`indices`*	Dump the RIs indicated in *`indices`*.
`-r` *`refs`*	Dump the RIs with reference numbers listed in *`refs`*.

`-n` *`names`*	Dump the RIs with names listed in *`names.`* Note: currently, a name that contains a ',' (comma) will be treated as two different names.
`-m` *`interlace`*	Dump data in interlace mode *`interlace=0, 1, or 2.`*
`-v`	Dump all RI contents, including all annotations. (default)
`-h`	Dump RI header information only, no data or element annotations.
`-d`	Dump RI data only, no tag/ref or header information. Output is formatted for input to **fp2hdf**.
`-p`	Dump palette information for the requested images or for all images if no specific image is requested. With *`-d`*, dump palette data only. With or without *`-v`* and without *`-d`*, dump palette data and header information.
`-c`	Print space characters as they are, not `\<digit>`.
`-g`	Do not print data of file (global) attributes.
`-l`	Do not print data of local attributes.
`-s`	Do not add carriage return to a long line, i.e. dump it as a stream.
`-o` *`filename`*	Print output to file *`filename`*.
`-b`	Output in binary format.
`-x`	Output in ASCII format. (default)
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays GR raster image information in the specified format. The `-r`, `-i`, and `-n` flags can be selected together. GR images are always stored in pixel interlace mode (see Section "*Accessing Images and Files: GRstart, GRselect, and GRcreate"*).

The **dumpvd** command

Syntax: `hdp dumpvd [-a | -i` *`indices`* `| -r` *`refs`* `| -n` *`names`* `| -c` *`classes`* `| -f` *`f1, f2,...`*`] [-v | -h | -d] [-o` *`filename`*`] [-bx]` *`filelist`*

Flags:

`-a`	Dump all vdatas in the file(s). (default)
`-i` *`indices`*	Dump the vdatas at positions listed in *`indices`*.
`-r` *`refs`*	Dump the vdatas with the reference numbers listed in *`refs`*.
`-n` *`names`*	Dump all the vdatas with names listed in *`names`*. Note: currently, a name that contains a ',' (comma) will be treated as two different names.
`-c` *`classes`*	Dump all the vdatas with the classes listed in *`classes`*. Note: same issue as with names regarding commas.
`-f` *`f1,f2,...`*	Dump data based on the indicated fields in the vdata header.
`-v`	Dump everything, including annotations. (default)
`-h`	Dump vdata header information only, no data or element annotations.
`-d`	Dump vdata data only, no tag/ref or header information. Output is formatted for input to **fp2hdf**.

`-o` *`filename`*	Print output to file *`filename`*.
`-b`	Output in binary format.
`-x`	Output in ASCII format. (default)
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays vdata information in the specified format. The `-r`, `-i`, `-n`, and `-c` flags can be selected together.

The **dumpvg** command

Syntax:

```
dumpvg [-a | -i indices | -r refs | -n names | -c classes] [-v | -h]
    [-o filename] filelist
```

Flags:

`-a`	Dump all vgroups in the file(s). (default)
`-i` *`indices`*	Dump the vgroups at positions listed in *`indices`*.
`-r` *`refs`*	Dump the vgroups with the reference numbers listed in *`refs`*.
`-n` *`names`*	Dump all the vgroups with names listed in *`names`*. Note: currently, a name that contains a ',' (comma) will be treated as two different names.
`-c` *`classes`*	Dump all the vgroups with classes listed in *`classes`*. Note: same issue as with names regarding commas.
`-v`	Dump everything, including annotations. (default)
`-h`	Dump vgroup header information only, no data or element annotations.
`-o` *`filename`*	Print output to file *`filename`*.
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays vgroup information in the specified format. The `-r`, `-i`, `-n`, and `-c` flags can be selected together. This command has no binary output option; it produces only ASCII text output.

The **dumprig** command

Syntax:

```
dumprig [-a | -i indices | -m n | -r refs] [-dhv]
        [-o filename [-b | -x]] filelist
```

Flags:

`-a`	Dump all RIGs in the specified file(s). (default)
`-i` *`indices`*	Dump theRIGs with the positions listed in *`indices`*.
`-m` *`n`*	Dump only RIGs with the specified data length. *`n`* can have a value of 8 or 24, for 8- or 24-bit raster images, respectively.
`-r` *`refs`*	Dump the RIGs with the reference numbers listed in *`refs`*.
`-d`	Dump RIG data only, no tag/ref or header information. Output is formatted for input to **fp2hdf**.
`-h`	Dump RIG header information only, no data or element annotations.
`-v`	Dump everything, including annotations. (default)
`-c`	Do not add carriage return to a long line, i.e. dump it as a stream.

`-o` *`filename`*	Print output to file *`filename`*.
`-b`	Output in binary format.
`-x`	Output in ASCII format. (default)
`filelist`	Names of HDF input files, separated by spaces.

Description: Displays RIG information in the specified format. The `-r`, `-i`, and `-m` flags can be selected together.

15.3. Comparing two HDF Files: hdiff

15.3.1. General Description

The **hdiff** utility compares two HDF files and reports differences between them. Only datasets, attributes, and vdata objects are compared. **Hdiff** returns 0 when no differences are found and 1, otherwise.

15.3.2. Command-line Syntax

The **hdiff** command line syntax is as follows:

```
hdiff [-V][-b][-g][-s][-d][-S][-D] [-v var1[, var2...]] [-u var1[, var2...]]
                                [-e count] [-t limit] [-p relative] file1 file2
```

The **hdiff** command line options and usage are described in Table 15D.

TABLE 15D **hdiff Option Flags**

`-b`		Verbose mode
`-g`	Attributes:	Compare global attributes only.
`-s`		Compare SD local attributes only.
`-d`	Data:	Compare SD data only.
`-D`		Compare Vdata data only.
`-v` *`var1 [, var2...]`*	Variables:	Compare SD data on the variable(s) *`var1`*, *`var2`*, etc.
`-u` *`var1 [, var2...]`*		Compare Vdata data on the variable(s) *`var1`*, *`var2`*, etc.
`-S`	Output:	Print statistics.
`-e` *`count`*		Print difference up to *`count`* instances for each variable. *`count`* is a positive integer.
`-t` *`limit`*		Print difference when it is greater than*`limit`*. *`limit`* is a positive floating point value.
`-p` *`relative`*		Print difference when it is greater than a relative limit.
`file1` *`file2`*	Filenames:	First and second HDF input files to be compared.

15.3.3. Examples

This section provides some examples of **hdiff** usage.

Example 1: to compare global attributes only

```
hdiff -g hdifftst1.hdf hdifftst2.hdf
```

Example 2: to compare SD local attributes only

```
hdiff -s hdifftst1.hdf hdifftst2.hdf
```

Example 3: to compare SDS data only

```
hdiff -d hdifftst1.hdf hdifftst2.hdf
```

Example 4: to compare Vdata data only

```
hdiff -D hdifftst1.hdf hdifftst2.hdf
```

Example 5: to print statistics

```
hdiff -d -S hdifftst1.hdf hdifftst2.hdf
```

Example 6: to compare SDS data on specific variable

```
hdiff -d -v dset1 hdifftst1.hdf hdifftst2.hdf
```

Example 7: to compare vdata data on specific variable

```
hdiff -D -u vdata1 hdifftst1.hdf hdifftst2.hdf
```

Example 8: to print up to 2 differences for each variable

```
hdiff -d -e 2 hdifftst1.hdf hdifftst2.hdf
```

Example 9: to print differences when there are more than 2

```
hdiff -d -t 2 hdifftst1.hdf hdifftst2.hdf
```

15.4. Displaying Vdata Information: vshow

15.4.1. General Description

Displays information about either one Vdata object or all Vdata objects in an HDF file.

15.4.2. Command-line Syntax

```
vshow input HDF filename [+|+vdata id]
```

The **vshow** option flags are described in Table 15E.

TABLE 15E **vshow Option Flags**

+	All Vdatas:	The utility will display information about all Vdata objects in the HDF file.
+*vdata_id*	One Vdata:	The utility will display information about the Vdata object corresponding to the specified vdata id.

15.4.3. Examples

The following command will display information about all of the Vdata objects in the HDF file named `image012.hdf`.

```
vshow image012.hdf +
```

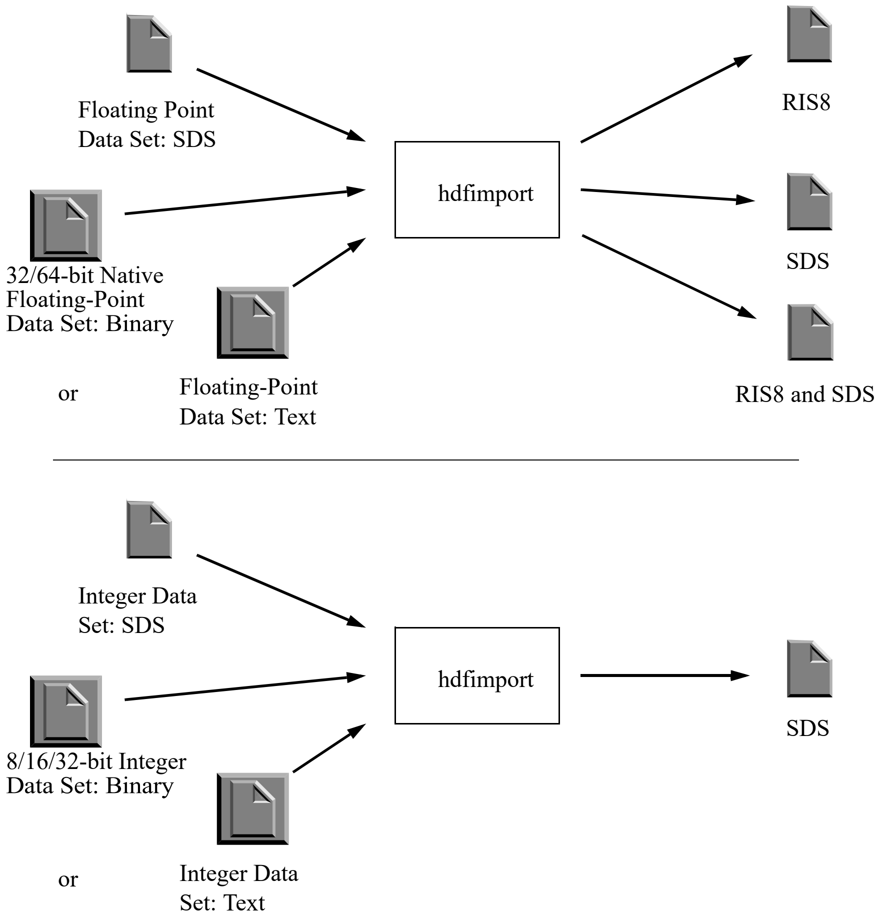
15.5. Converting Floating-point or Integer Data to SDS or RIS8: hdfimport

Note that **hdfimport** replaces the **fp2hdf** utility that was distributed with earlier HDF releases.

15.5.1. General Description

The **hdfimport** utility converts data from ASCII text files, 32-bit or 64-bit native floating point data files, 8-bit, 16-bit or 32-bit integer files, or HDF floating-point scientific data sets to either HDF floating-point scientific data sets or 8-bit HDF raster image datasets, or both, and stores the results in an HDF file. (See Figure 15a) The images can be scaled on a user-specified mean value.

FIGURE 15a **The hdfimport Utility**



15.5.2. Command-line Syntax

The syntax of **hdfimport** is as follows.

```
hdfimport -h[elp]
hdfimport input-file [[-t[ype] output-type |-n]
                          [input-file[-t[ype] output-type | -n]]]
                          -o[utfile] output-file
```

```
                                    [-r[aster] [raster-options . . .]
                                    [-f[loat]]
```

The *input-file* parameter specifies the name of the file containing the unconverted data set. The file may contain a single two-dimensional or three-dimensional array in ASCII text, native floating point, native integer, or HDF SDS format. If an HDF file is used for input, it must contain an SDS. The SDS need only contain a dimension record and the data, but if it also contains maximum and minimum values and/or scales for each axis, these will be used. If the format is ASCII text, native floating point, or native integer, see Table 15G and the accompanying discussion regarding the required structure of the data.

Data from one or more input files will be stored as datasets and/or images in a single output file, the HDF file specified in the parameter *output-file*. The output file will contain one SDS and/or one image for each input file.

The **hdfimport** options and parameters are described in Table 15F.

TABLE 15F

hdfimport Options and Parameters

`-h`	Help:	Prints a usage summary, then exits.

`-t` *`output-type`*
`-type` *`output-type`*
	Output datatype:	Optionally used with each ASCII input file to specify the data type of the data set to be written. Can be any of the following values: `FP32` (default), `FP64`, `INT8`, `INT16`, or `INT32`. If not specified, the default value of `FP32` is assumed.

`-n`	64-bit output:	Used only if a binary input file contains 64-bit foating point data and the default behavior of writing the output as a 32-bit dataset should be overridden to write it as a 64-bit dataset.

`-r` *`raster-options`*
`-raster` *`raster-options`*
	Raster:	Stores the data as a raster image set in the output file. The available *`raster-options`* are described below.

`-f` `-float`	Float:	Stores the data as a scientific data set, an SDS, in the output file. (Default if the `-r` option is not specified.) 32-bit binary input data will be stored to a 32-bit SDS. 64-bit binary input data will be stored to a 64-bit SDS.
`raster-options`	Raster options:	Additional options that accompany the `-r` (or `-raster`) option are as follows:
`-e` *`horiz vert`* [*`depth`*] `-expand` *`horiz vert`* [*`depth`*]	Expand:	Expands the floating point data via pixel replication to produce the output image(s). *`horiz`* and *`vert`* specify the horizontal and vertical resolutions of the image(s) to be produced. The optional parameter *`depth`* is used only with 3-dimensional input data and specifies the number of images or depth planes. If *`max`*, the maximum value, and *`min`*, the minimum value, are supplied in the input file, this option clips values that are greater than *`max`* or less then *`min`*, setting them to the *`max`* and *`min`*, respectively. Cannot be used with the `-i` option.

`-i` *`horiz vert`* `[`*`depth`*`]`
`-interp` *`horiz vert`* `[`*`depth`*`]`
	Interpolation:	Applies bilinear or trilinear interpolation when expanding floating-point data.
		The values of the *`horiz`*, *`vert`*, and *`depth`* parameters specify the horizontal, vertical, and depth resolutions of the dataset(s) to be produced and must be greater than or equal to the dimensions of the original dataset.
		If a maximum value, *`max`*, and/or a minimum value, *`min`*, are supplied in the input file, this option clips values that are greater than *`max`* or less then *`min`*, setting them to the *`max`* and *`min`*, respectively.
		Cannot be used with the `-e` option.

`-p` *`palette`*
`-palfile` *`palette`*
	Palette:	Stores the palette with the image. The *`palette`* parameter names the file containing the palette data. This may be an HDF file containing a palette or a file containing a raw palette.

`-m` *`mean`*
`-mean` *`mean`*
	Mean:	Causes the data to be scaled around a user-specified *`mean`* when generating the image.
		The new maximum and minimum values, *`newmax`* and *`newmin`*, will be equidistant from *`mean`* and determined by the following formulae:
		newmax = *mean* + max(abs(*max* - *mean*), abs(*mean* - *min*))
		newmin = *mean* - max(abs(*max* - *mean*), abs(*mean* - *min*))
		If no mean value is specified, then the mean will be `0.5*(`*`max + min`*`)`.

The *-e* and the *-i* flags cannot be used simultaneously. Either pixel interpolation or bilinear interpolation can be chosen for image expansion, but not both.

Data from several input files (with one data set per input file) are stored as several data sets and/or images in one output HDF file. Alternatively, a shell script can be used to call **hdfimport** repeatedly to convert data from multiple input files to corresponding output HDF files.

15.5.3. Structure of Data in non-HDF Input Files

If the format of*`input-file`* is ASCII text, native floating point, or native integer (i.e., *`input-file`* is not an HDF file), the data must be structured in fields as described below.

TABLE 15G **hdfimport ASCII Text, Native Floating Point, or Native Integer Input Fields**

format	Must contain exactly one format designator: `TEXT`, `FP32`, `FP64`, `IN32`, `IN16`, or `IN08`
rank	Dimension ranks, the next three fields, are specified in the order of slowest-changing dimension first. For two-dimensional input, please see note in the *number of planes* cell below.
number_of_columns	Rank of the fastest-changing dimension, the horizontal axis, or X-axis, in a 3-dimensional scale
number_of_rows	Rank of the vertical axis, or Y-axis, in a 3-dimensional scale
number_of_planes	Rank of the slowest-changing dimension, the depth axis, or Z-axis, in a 3-dimensional scale. *Note*: it must contain the value `1` for two-dimensional input.
max	Maximum data value
min	Minimum data value
plane1 plane2 plane3 ...	Scales for the depth axis
row1 row2 row3 ...	Scales for the vertical axis
col1 col2 col3 ...	Scales for the horizontal axis
data1 data2 data3 ...	Raw data ordered by rows, left to right and top to bottom; then optionally by planes, front to back
...	Data continues...

format, *number_of_columns*, *number_of_rows*, and *number_of_planes* are native integers. *format* is the integer representation of the appropriate 4-character string (`0x46503332` for `FP32`, `0x46503634` for `FP64`).

If the data input format is `FP32` or `FP64`, the remaining input fields are composed of native 32-bit floating point values for `FP32` input format, or native 64-bit floating point values for `FP64` input format data.

If the data input format is `IN08`, `IN16`, or `IN32`, the remaining input fields are composed of native 8-bit integer values for `IN08` input format, native 16-bit integer values for `IN16` input format, or native 32-bit integer values for `IN32` input format data.

The term ***scale*** refers to the spacing between points on the axes. If the spacing is uniform, i.e., the gaps are of equal size, a uniform scale is specified -- for example, `1.0, 2.0, 3.0, ...`... Scales may be omitted in an HDF file; they must be included in a text file.

The arrays containing the plane, row, and column scales must have a size equal to the values specified in the *number_of_rows* , *number_of_columns,* and *number_of_planes* positions, respectively.

15.6. Converting 8-Bit Raster Images to the HDF Format: r8tohdf

15.6.1. General Description

The **r8tohdf** utility converts a set of raw raster images to the HDF RIS8 format and writes them to a file.

15.6.2. Command-line Syntax

```
r8tohdf [number-of-rows number-of-columns] output-filename [-p palette-filename]
                [-c|-r|-i] raw-raster-image-filename-1, raw-raster-image-file-
                name-2, ... raw-raster-image-filename-n
```

The option flags are described in Table 15H.

TABLE 15H **r8tohdf Option Flags**

`-p`	Palette File	Inserts a palette stored in the file *`palette-filename`* in the RIS8. If the -p flag is not specified, a palette is not stored with the RIS8.
`-c`	Run-length Encoding	Compresses the output data using run-length encoding.
`-i`	IMCOMP Compression	Compresses the output data using the IMCOMP method.
`-r`	No Compression	No compression is applied to the output data. (the default)

15.6.3. Examples

A file named `rawras` contains a 256 x 512-byte raw raster image, and its palette is stored in a file name `mypal`. To convert the information in these files to an RIS8 without compression and store the RIS8 in a file named `ras.hdf`, enter the following **r8tohdf** command:

```
r8tohdf 256 512 ras.hdf -p mypal rawras
```

A 800 x 1000-byte raw raster image is stored in a file named `bigpic`. This data must be converted to a RIS8 without a palette, compressing it using run-length encoding, then stored in a file named `bigpic.hdf`. The following command will do this:

```
r8tohdf 800 1000 bigpic.hdf -c bigpic
```

A 300 x 400 raw raster image is contained in each of the files named `pic1`, `pic2`, and `pic3`. To convert all three files to RIS8s, compress them using the IMCOMP method, and store them in a file named `pic.hdf`, enter

```
r8tohdf 300 400 pic.hdf -i pic1 pic2 pic3
```

Different types of raster image data are to be stored in a file named `ras.hdf`. The image data in the file `rawras1` will be stored without a palette. The image data sets from the file named `rawras2` are to be stored with a palette extracted from a file named `mypal`. The images from the `rawras1` and `rawras2` files are to be compressed using run-length encoding, and the image in the `rawras3` file is not to be compressed. The size of all images are 256 x 512 bytes. The following command is used to do this:

```
r8tohdf 256 512 ras.hdf -c rawras1 -p mypal rawras2 -r rawras3
```

15.7. Converting 24-Bit Raw Raster Images to RIS8 Images: r24hdf8

15.7.1. General Description

The **r24hdf8** utility quantizes a raw RGB 24-bit raster image, creating an 8-bit image with a 256-color palette, then it stores the palette and raster image data in an HDF file.

15.7.2. Command-line Syntax

```
r24hdf8 [x-dimension-length y-dimension-length] raw-24-bit-image-filename-hdf
                ris8-image-filename
```

The pixel order in the raw 24-bit image file is left-to-right and top-to-bottom. Each pixel data element consists of three contiguous bytes, the first representing the red intensity value, the second the green intensity value, and the third the blue intensity value. Use the **ptox** filter to convert the raster image data from a pixel-interlaced format to scan-plane interlaced.

15.7.3. Examples

A file named `rawraster` containing 24-bit raw raster images with x and y-dimensions of 480 x 640, respectively, must be converted to the HDF RIS8 format and stored in a file named `hdfraster`. The following command is used to do this:

```
r24hdf8 480 640 rawraster hdfraster
```

15.8. Converting Raw Palette Data to the HDF Palette Format: paltohdf

15.8.1. General Description

The **paltohdf** utility converts raw palette data to the HDF palette format. The raw palette data must have 768 bytes organized in the following order: first, 256 contiguous red intensity values, then 256 contiguous green intensity values, then 256 contiguous blue intensity values. The palette in the HDF file will have the RGB values pixel-interlaced, as follows.

```
red-value green-value blue-value red-value green-value blue-value ...
```

This is the standard HDF format for 8-bit palettes.

15.8.2. Command-line Syntax

```
paltohdf raw-format-palette-filename HDF-format-palette-filename
```

If an HDF palette format file is specified that does not exist, it is created before the converted data is stored. If an HDF palette format file is specified that already exists, the converted data is appended to the file.

15.9. Extracting 8-Bit Raster Images and Palettes from HDF Files: hdftor8

15.9.1. General Description

The **hdftor8** utility extracts the raster images and/or palettes from an HDF file and stores them in one file that contains the raster image data and another that contains the palette data.

15.9.2. Command-line Syntax

```
hdftor8 input-HDF-filename [-i] [-v] [-r raster-image-filename] [-p palette-
                filename]
```

The option flags are described in Table 15I.

TABLE 15I **hdftor8 Option Flags**

`-i`	Interactive Mode:	Program is executed in interactive mode.
`-v`	Verbose Mode:	Program is executed in verbose mode. Diagnostic messages are displayed during the session.
`-r`	Raster Image File Name:	The raster image file name immediately follows this flag.
`-p`	Palette File Name:	The palette file name immediately follows this flag.

The names given as the HDF format file, raster image file, and palette file are interpreted by **hdftor8** as follows: For each raster image file, the file name is given the extension

```
.#.@.%
```

where `#` represents the raster image number from the HDF file, `@` represents the x-dimension of the raster image and `%` represents the y-dimension of the raster image. For each palette file, the file name is given the extensions `.#`, where `#` represents the palette number from the HDF format file.

If no name is given for the raster image file, the default name `img.#.@.%` is assigned, where `#`, `@`, and `%` are defined as in the preceding paragraph. The default name for a palette file, if no name is specifically given in the command, is `pal.#`.

15.9.3. Examples

A file named `denm.hdf` contains three 512 x 256 raster images and three palettes. To store these images and palettes in separate raster image and palette files, use the following **hdftor8** command:

```
hdftor8 denm.hdf
```

Six files are created, named `img1.512.256`, `img2.512.256`, `img3.512.256`, `pal.1`, `pal.2`, and `pal.3`.

15.10. Extracting Palette Data from an HDF File: hdftopal

15.10.1. General Description

The **hdftopal** utility converts a palette in an HDF file to a raw palette in an non-HDF file. The raw palette will have 768 bytes with the first 256 bytes representing red intensity values, the second 256 bytes representing green intensity values, and the third 256 bytes representing blue intensity values. The utility performs the converse operation of the **paltohdf** utility.

15.10.2. Command-line Syntax

```
hdftopal HDF-format-palette-filename raw-format-palette-filename
```

15.11. Converting Several RIS8 Images to One 3D SDS: ristosds

15.11.1. General Description

The **ristosds** utility creates a single HDF file consisting of a three-dimensional SDS from a set of HDF files containing one or more raster images. All images in the input HDF files must have the same dimensions. If a palette is to be included with the images, it should be in the first HDF input file. Only one palette can be associated with the images; any additional palette data encountered by the utility after the first palette has been processed will be ignored.

15.11.2. Command-line Syntax

```
ristosds input-filename-1, input-filename-2, ... input-filename-n [-o output-
                filename]
```

15.11.3. Examples

The contents of a directory consists of 20 files named `storm001.hdf`, `storm002.hdf`, ... `storm020.hdf`. Each file contains a single RIS8 with a 100 x 200 raster image. A file that combines these 20 raster images into a 32-bit floating-point SDS with the dimensions 100 x 200 x 20 can be created with the following **ristosds** command:

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

15.12. Converting an HDF RIS24 Image to an HDF RIS8 Image: hdf24hdf8

15.12.1. General Description

The **hdf24hdf8** utility quantizes an HDF RGB RIS24 pixel-interlaced image, producing an HDF RIS8 image with a 256-color palette and stores the palette and raster image data in an HDF file.

15.12.2. Command-line Syntax

```
hdf24hdf8 ris24-image-filename ris8-image-filename
```

15.13. Compressing RIS8 Images in an HDF File: hdfcomp

15.13.1. General Description

The **hdfcomp** utility reads RIS8 images from a set of HDF files, compresses them and stores the compressed data in a second HDF file. If the output HDF file exists, the compressed images will be appended to it.

15.13.2. Command-line Syntax

```
hdfcomp output-filename [-c|-r|-i] input-filename-1, [-c|-r|-i]input-filename-2,
                ... [-c|-r|-i] input-filename-n
```

The option flags are described in Table 15J

TABLE 15J **hdfcomp Option Flags**

`-r`	No compression:	The raster image data is not compressed. (the default)
`-c`	Run-length Encoding:	The raster image data is compressed using run-length encoding.
`-i`	IMCOMP Compression:	The raster image data is compressed using the IMCOMP algorithm.

15.13.3. Examples

A directory contains twenty files named `storm001`, `storm002`, ... `storm020`. Each of these files contains a single RIS8 image. To compress these images using run-length encoding and store them in a file named `altcomp.hdf`, use the following **hdfcomp** command:

```
hdfcomp allcomp.hdf -c storm*.hdf
```

15.14. Compressing an HDF File: hdfpack

15.14.1. General Description

The **hdfpack** utility compacts an HDF file by reading in all the data elements in the file and writing them out to a new HDF file, eliminating waste spaces. It does not compress the file as in using a compression algorithm to compress the data.

15.14.2. Command-line Syntax

```
hdfpack [-i|-b] [-d number-of-data-descriptors-per-block] [-t number-of-linked-
                blocks-per-table-entry] input-HDF-filename output-HDF-filename
```

The **hdfpack** option flags are described in Table 15K.

TABLE 15K **hdfpack Option Flags**

`-b`	Non-coalesced blocks:	The utility will not coalesce linked-block elements.
`-i`	Interactive mode:	The utility will prompt for each linked-block element.
`-d`	Data descriptors per block:	The output file will be created with the specified number of data descriptors per block of data descriptors.
`-t`	Linked-blocks per table entry:	The output file will be created with the specified number of linked blocks per table entry.

15.14.3. Examples

To compress the data in the file named `aa.hdf` and store the compressed data in the file named `aa.cmp`, use the following **hdfpack** command:

```
hdfpack aa.hdf aa.cmp
```

Suppose a file named `bb.hdf` contains data elements stored as sequences of linked blocks. The following **hdfpack** command compresses the file while leaving the linked-block elements intact, and writes the compressed data to a file named `bb.blk`.

```
hdfpack -b bb.hdf bb.blk
```

15.15. Reformatting an HDF File: hrepack

15.15.1. General Description

hrepack is a command line utility that performs a logical copy of an input HDF4 file to an output HDF4 file, copying all the high level objects while optionally rewriting the objects with or without compression and/or with or without chunking.

Note that compression is supported only for data sets and images in HDF4. In addition, **hrepack** only compresses objects of size 1024 bytes or greater, by default. Option -m can be used to specify a different minimum object size.

Prior to HDF version 4.2.11, an output file from **hrepack** would contain a vgroup of class `RIG0.0` eventhough the original file did not have any raster image elements. This is no longer true starting in 4.2.11.

15.15.2. Command-line Syntax

The **hrepack** syntax is as follows:

```
hrepack -i input -o output [-h] [-v] [-t "comp_info"]
               [-c "chunk_info"] [-f cfile] [-m number]
```

The **hrepack** options and usage are as follows:

`-i` *input*	The input HDF file.
`-o` *output*	The output HDF file.
`-h`	Print usage, or help, message.
`-v`	Print verbose.

`-t "`*`comp_info`*`"`	Specifies the compression type. "*`comp_info`*" is a string with the format "*`list_of_objects`* `:` *`type_of_compression compression_parameters`*" *`list_of_objects`* is a comma-separated list of object names, indicating to apply the specified type of compression only to those objects. `"*"` means to apply the specified type of compression to all objects. *`type_of_compression`* should be one of the following values: • `RLE` for RLE compression • `HUFF` for Huffman compression • `GZIP` for gzip compression • `JPEG` for JPEG compression • `SZIP` for Szip compression • `NONE` to uncompress the object *`compression_parameters`* contains optional compression information as follows: • for `RLE`, no additional information • for `HUFF`, the skip-size • for `GZIP`, the deflation level • for `JPEG`, the quality factor • for `SZIP`, no additional information
`-c "`*`chunk_info`*`"`	Specifies the objects to which to apply chunking. "*`chunk_info`*" is a string with the format "*`list_of_objects`* `:` *`chunk_information`*" *`list_of_objects`* is a comma-separated list of object names, indicating to apply chunking only to those objects. `"*"` means to apply chunking to all objects. *`chunk_information`* specifies the chunk size of each dimension and is of the format *`dim_1`* `x` *`dim_2`* `x ...` *`dim_n`*. The value `NONE` indicates that the object is not to be chunked, i.e., stored as a contiguous data set, even it was stored as a chunked data set in the orignal file.
`-f` *`comp_file`*	Specifies a file, *`comp_file`*, containing the compression information. This option is used in lieu of the `-c` and `-t` options.
`-m` *`number`*	Do not compress objects of a size less than *`number`* bytes. If -m is not specified, a minimum size of 1024 bytes is assumed.

15.15.3. Examples

This section provides some examples of **hrepack** usage.

Example 1: to compress all objects in the file file1.hdf, using RLE compression

```
hrepack -v -i file1.hdf -o file2.hdf -t '*:RLE'
```

Example 2: to apply Skipping Huffman compression with skip factor of 1, for objects /group1/A, /group2/B and C

```
hrepack -v -i file1.hdf -o file2.hdf -t '/group1/A,/group2/B,C:HUFF 1'
```

Example 3: to apply RLE compression for object /group1/D and chunking to objects D and E using a chunk size of 10 for the 2 dimensions

```
hrepack -v -i file1.hdf -o file2.hdf -t '/group1/D:RLE' -c 'D,E:10x10'
```

15.16. Creating Vgroups and Vdatas: vmake

15.16.1. General Description

The **vmake** utility creates Vgroup and Vdata objects in the specified HDF file.

15.16.2. Command-line Syntax

To create a new Vgroup:

```
vmake <output_HDF_filename> "Vgroup_name"
```

To create a new Vdata object:

```
vmake <output_HDF_filename> <Vdata_object_name> <Vdata_field_data_type>
```

The *Vdata_field_data_type* argument consists of a Vdata field name followed by an equal sign and one of the following characters:

- `c` for character data (`char` in the HDF file)
- `b` for byte data (`int8` in the HDF file)
- `s` for short integer data (`int16` in the HDF file)
- `l` for long integer data (`int32` in the HDF file)
- `f` for floating point data (`float32` in the HDF file)

Any of these characters may be preceded by a decimal number specifying an element size other than one.

To create links from one or more Vdatas to a specified Vgroup:

```
vmake <output_HDF_filename> [-l] "Vgroup_ref_number" "Vdata1_ref_number"
                                 "Vdata2_ref_number" ... "Vdatan_ref_number"
```

Vgroup_ref_number is the reference number of the Vgroup to which the Vdatas are to be linked. *Vdata1_ref_number* through *Vdatan_ref_number* are the reference numbers of the Vdatas being linked to the Vgroup.

Note that all **vmake** arguments, except the output HDF filename, are surrounded by double quotes.

15.16.3. Examples

Assume the following. A file containing storm data is named `storm.dat`. A Vdata object named `Storm Data B` must be created in an HDF file named `sdata.hdf` using **vmake**. The new Vdata object is to contain a field named `PLIST` with an element size of three long integers. And finally, the data in `storm.dat` is to be loaded into the Vdata object `Storm Data B`.

This can be accomplished with the following command:

```
vmake sdata.hdf "Storm Data B" "PLIST=3l" < storm.dat
```

15.17. Listing Basic Information about Data Objects in an HDF File: hdfls

15.17.1. General Description

The **hdfls** utility provides general information about the tags, reference numbers, and if requested, lengths of the data elements.

The **hdfls** utility provides general information about the HDF data objects in a file. This information includes the tags and reference numbers of the data objects, the lengths and offsets of the HDF object's data elements, the contents of DD blocks, and information regarding special elements. In situations where the DD block information is not needed, we recommend the **hdp** utility with the `list` command and its options.

15.17.2. Command-line Syntax

```
hdfls [-o][-l][-d][-v][-g][-s][-h][-t #] filename
```

When no flags are used, **hdfls** displays data objects ordered by the tags and reference numbers. Contents of the DD blocks and lengths and offsets of the data elements are not displayed.

The option flags are described in Table 15L.

TABLE 15L **hdfls Option Flags**

`-o`	Order off:	Turns off ordering. Displays data objects in the order in which they are listed in the DD block. Sequential data objects in the DD block with the same tag are grouped together.
`-l`	Long format:	Displays data objects in ascending tag and reference number order along with the length of each data element.
`-d`	Offset/length:	Displays two lists. Displays tags and reference numbers of the data objects and offsets and lengths of the corresponding data elements in the order in which the objects appear in the DD blocks. Then lists data objects in ascending tag and reference number order
`-v`	Verbose:	Displays annotation and label text, along with the information triggered by the `-l` flag
`-g`	Group contents:	Displays the contents of each group, along with the information triggered by the `-l` flag.
`-s`	Special elements:	Displays information about each special element, along with the information triggered by the `-l` flag
`-h`	DD block:	Displays DD block header information and DD block contents followed by the list of data objects in tag and reference number ascending order.
`-t`	Tag:	Lists information about the data objects with the specified tag. Must be followed by a tag number.

15.17.3. Examples

The file `SDSchunked.hdf`, created by the Example 17, in Chapter 3, contains one chunked data set.

`hdfls -s` can be used to display information about the data objects and special elements in this file. Note that data objects are listed in tag and reference number ascending order.

For example, the command

```
hdfls -s SDSchunked.hdf
```

would display the following output:

```
SDSchunked.hdf:
File library version: Major= 4, Minor=1, Release=2
String=NCSA HDF post Version 4.1 Release 2, March 1998

Linked Blocks Indicator       : (tag 20)
        Ref no      1         12 bytes
        Ref no      2         34 bytes
        Ref no      3       4096 bytes

Version Descriptor            : (tag 30)
        Ref no      1         92 bytes
```

```
Data Chunk                    : (tag 61)
        Ref no      1         12 bytes
        Ref no      2         12 bytes
        Ref no      3         12 bytes
        Ref no      4         12 bytes
        Ref no      5         12 bytes
        Ref no      6         12 bytes

Number type                   : (tag 106)
        Ref no     12          4 bytes

SciData dimension record      : (tag 701)
        Ref no     12         22 bytes

Numeric Data Group            : (tag 720)
        Ref no      2         16 bytes

Vdata                         : (tag 1962)
        Ref no      4        116 bytes
        Ref no      7         60 bytes
        Ref no      9         60 bytes
        Ref no     11         60 bytes

Vdata Storage                 : (tag 1963)
        Ref no      7          4 bytes
        Ref no      9          4 bytes
        Ref no     11          2 bytes

Vgroup                        : (tag 1965)
        Ref no      8         33 bytes
        Ref no     10         33 bytes
        Ref no     13         60 bytes
        Ref no     14         47 bytes

Special Scientific Data       : (tag 17086)
        Ref no      3         72 bytes
        Chunked Element:
        logical size: 12
        number of dimensions: 2
        array of chunk lengths for each dimension:       3       2

Special Vdata Storage         : (tag 18347)
        Ref no      4         72 bytes
        Linked Block: first 12 standard 4096 per unit 16
```

15.18. Editing the Contents of an HDF File: hdfed

15.18.1. General Description

The **hdfed** utility allows experienced HDF users to manipulate the elements of an HDF file. These manipulations include

- Selecting groups and showing information about them.
- Dumping group information to output files.
- Writing group data to output files.
- Deleting groups from HDF files.
- Inserting groups in HDF files.
- Replacing elements of HDF files.

- Editing the labels and descriptions of any element in an HDF file.

It is designed primarily for users who need to know about HDF files at the level of individual data elements. It is not designed to provide a comprehensive high-level view of the contents of an HDF file -- other tools and utilities should be used for that purpose. To use **hdfed** one should be familiar with the components of an HDF file covered in the *HDF Specifications manual*.

The **hdfed** utility is loosely modeled on `ed`, the UNIX line editor. When `hdfed` is invoked, it prompts the user for commands, as does `ed`. Also, basic command syntax and description information is available to the user through **hdfed**. The most common **hdfed** commands are used to control the position in the HDF file and the format of the information provided.

The initial view of the file under **hdfed** consists of a set of tag/reference number pairs. Although **hdfed** allows modification of tags and reference numbers *within strict constraints*, it will not allow the user to arbitrarily modify binary data in the file.

The following terms and concepts must be understood in order to use **hdfed** correctly and will be used in the following discussion about **hdfed**.

- The ***data object*** or ***object*** refers to an HDF data object and the data descriptor of that object. (i.e., tags, reference numbers, offsets, or lengths.)
- The ***data*** or ***data element*** refers to the record that the data descriptor points to. For a precise definition of the data that is associated with a given tag consult the *HDF Specifications and Developer's Guide v3.2* from the HDF web site at `http://www.hdfgroup.org/`.
- The ***group*** refers to a predefined collection of data objects that correspond to a particular application. For example, a raster image group refers to the collection of objects that are used to store all of the information in a raster image set.

Once an HDF file has been opened by **hdfed**, the following operations can be performed on the data file, among others:

- Select an HDF object to examine more closely.
- Move forward or backward within the HDF file.
- Get information about an object. (tag, reference number, size, label)
- Display a raster image using the ICR protocol.
- Display the contents of any object.
- Delete an object.
- Annotate an object with a label or description.
- Write an object to a second HDF file.
- Write data elements in binary form to a non-HDF file.
- Close the file and exit, or open a new file.

15.18.2. Command-line Syntax

The syntax of **hdfed** is

```
hdfed [-nobackup][-batch] filename
```

If a file named *`filename`* exists, it is opened and a backup is made of the file. Files may also be opened from within the editor.

The option flags are described in Table 15M.

TABLE 15M **hdfed Option Flags**

-nobackup	Specifies that no backup file is to be made. If this option is omitted, a backup file is automatically created.
-batch	Specifies that input to **hdfed** is to be input via a stream of **hdfed** commands, rather than interactively.

The *-batch* flag is useful when a group of commonly-used commands are included in a UNIX shell script. The following is an example of such a script, using the C-shell, that lists information about the groups in a specified HDF file.

```
#!/bin/csh -f
set file=$1
shift
hdfed -batch $file -nobackup << EOF
info -all group $*
close
quit
EOF
echo ""
```

To receive usage information, as well as a quick list of the **hdfed** commands, type the command

```
hdfed -help
```

While in **hdfed**, the standard command prompt is displayed.

```
hdfed>
```

Many **hdfed** commands have qualifiers, or flags. For example, the command **info** may be followed by the *-all*, *-long*, *-group*, or *-label* flags.

All of the commands and flags can be abbreviated to the extent that their abbreviations are unique. For example, *-he* is ambiguous as it could stand for either the *-hexadecimal* or the *-help* flags, but *-hel* is not ambiguous.

TABLE 15N **The hdfed Command Set**

Name	Description
`help`	Displays general **hdfed** help information.
`open`	Opens an HDF file.
`close`	Closes an HDF file.
`revert`	Reverts to the original HDF file.
`next`	Goes to the next object or group that satisfies the predicate.
`prev`	Goes to the previous object or group that satisfies the predicate.
`info`	Displays information about the current data object.
`dump`	Displays information about the current data object in non-default formats. (i.e., binary, ASCII, etc.). The default is octal.
`display`	Displays a raster image using ICR.
`put`	Writes the current data element in a non-HDF file with the specified filename in binary format.
`putr8`	Writes the current RIS8 group into a non-HDF file with the specified file-name.
`getr8`	Reads a RIS8 group from a non-HDF file with the specified filename.
`delete`	Deletes an object or group.
`write`	Writes an object or group to an HDF file.
`annotate`	Annotates an object.
`if`	Conditional statement.
`select`	Loop for each object.
`alias`	Defines an alias or display the alias list.
`unalias`	Deletes an alias.
`wait`	Prints a message and wait for a carriage return.

To obtain information about the usage of any **hdfed** command, type the following at the **hdfed** prompt.

```
any-hdfed-command -help
```

Note that usage information cannot be obtained by typing only the command, with no flags. There are other **hdfed** commands, such as `delete`, that do not require an argument, so watch out for this kind of error.

There is a subset of **hdfed** commands where *predicates*, *items*, and *comparators* are used. ***Items*** are used to denote an HDF object type and can be any of the following identifiers; *tag*, *ref*, *image_size*, or *label*. A ***comparator*** is an expression used to compare an item with a user-defined value, and can be any of the following:

=	equal	!=	not equal
<	less than	<=	less than or equal
>	greater than	>=	greater than or equal

User-defined values can be either a number (with or without a decimal point) or a string of characters delimited by double-quotes. ***Predicates*** consist of items, comparators and user-defined values and are of the syntax:

```
item comparator-value
```

Or they may consist of the identifier **group**, as in the **next group** command. Some examples of predicates are:

```
next group
next (same as "next group" as "group" is the default identifier)
next tag = 720
next ref = 2
next image_size < 1000
next label = "abc"
```

The following is a more inclusive description of the **hdfed** commands.

The **help** command

Syntax: help

Flags: None

Description: Prints a help screen describing the basic purpose and functionality of the hdfed utility.

Usage Example:

```
hdfed> help
hdfed allows sophisticated HDF users the ability to manipulate the
elements in an HDF file. These manipulations include selecting groups
...
```

The **open** command

Syntax: open [-nobackup] *filename*

Flags: -nobackup The specified file name is not backed up.

Description: Opens the specified HDF file.

Usage Example:

```
    hdfed> open -help
    open <file> [-nobackup]
    -nobackup Don't make a backup for this file.
    hdfed>
    hdfed> open h1
    hdfed>
```

The **info** command

Syntax: info [-all] [-long] [-group] [-label]

Flags:

-all	Displays information for all of the objects in the current file.
-long	Displays the long form of the information.
-group	Organizes the information into groups.
-label	Shows any labels.

Description: Displays information for a data object. The listing for special elements will contain a special tag value (in Item 13 below it's 18347, which corresponds to DFTAG_VS) and the text "Unknown Tag".

Usage Example:

```
hdfed> info -all -label -long
(1) Version Descriptor: (Tag 30)
    Ref: 1, Offset: 202, Length:92 (bytes)
(2) Scientific Data: (Tag 702)
    Ref: 2, Offset: 294, Length : 200 (bytes)
```

```
(3) Number type: (Tag 106)
    Ref: 2, Offset: 494, Length: 4 (bytes)
(4) SciData description: (Tag 701)
    Ref: 2, Offset: 498, Length: 2 (bytes)
(5) SciData max/min: (Tag 707)
    Ref: 2, Offset: 520, Length: 4 (bytes)
*(6)Numeric Data Group: (Tag 720)
    Ref: 2, Offset: 524, Length: 12 (bytes)
    Label: Experiment #1
(7) Data Id Label: (Tag 104)
    Ref: 3, Offset: 536, Length: 17 (bytes)
(8) Scientific Data: (Tag 702)
    Ref: 4, Offset: 553, Length: 400 (bytes)
(9) Number type: (Tag 106)
    Ref: 4, Offset: 953, Length: 4 (bytes)
(10)SciData description: (Tag 701)
    Ref: 4, Offset:957, Length: 22 (bytes)
(11)Numeric Data Group: (Tag 720)
    Ref: 4, Offset: 979, Length: 8 (bytes)
    Label: Experiment #2
(12)Data Id Label: (Tag 104)
    Ref: 5, Offset: 987, Length: 17 (bytes)
(13)Unknown Tag: (Tag 18347)
    Ref: 8, Offset: 0, Length: 40(bytes
hdfed>
hdfed> info -group -all
**Group 1:
    Numeric Data Group: (Tag 720) Ref 2
    Scientific Data: (Tag 702) Ref 2
    SciData description : (Tag 701) Ref 2
    SciData max/min : (Tag 707) Ref 2
**Group 2:
    Numeric Data Group: (Tag 720) Ref 4
    Scientific Data : (Tag 702) Ref 4
    SciData description : (Tag 701) Ref 4
**These do not belong to any group:
    Version Descriptor: (Tag 30) Ref 1
    Number Type : (Tag 106) Ref 2
    Data Id Label: (Tag 104) Ref 3
    Number Type : (Tag 106) Ref 4
    Data Id Label: (Tag 104) Ref 5
hdfed>
```

The **prev** command

Syntax: prev *predicate-list*

Flags: None.

Description: Moves to the next object that satisfies the predicate list.

Usage Example:

```
hdfed> info -all
    (1)     Version Descriptor: (Tag 30) Ref 1
    (2)     Scientific Data: (Tag 702) Ref 2
    (3)     Number type : (Tag 106) Ref 2
    (4)     SciData description: (Tag 701) Ref 2
    (5)     SciData max/min : (Tag 707) Ref 2
    *(6)    Numeric Data Group : (Tag 720) Ref 2
    (7)     Data Id Label: (Tag 104) Ref 3
    (8)     Scientific Data: (Tag 702) Ref 4
    (9)     Number type : (Tag 106) Ref 4
    (10)    SciData description: (Tag 701) Ref 4
    (11)    Numeric Data Group: (Tag 720) Ref 4
    (12)    Data Id Label: (Tag 104) Ref 5
hdfed>
hdfed> ! The ’*’ in the first column marks the current
hdfed> ! position.
hdfed> ! The ’next’ and ’prev’ commands work with predicates.
hdfed> ! If I want to move to the max/min element,
hdfed> ! I can use the ’tag=’ predicate.
hdfed>
hdfed> prev tag=707
hdfed> info
    (5)     SciData max/min (SciData) : (Tag 707) Ref:2
hdfed>
```

The **next** command

Syntax: next *predicate-list*

Flags: None.

Description: Moves to the next object that satisfies the predicate.

Usage Example:

```
hdfed> ! Move in the file using next and prev
hdfed> ! The move direction depends on the relative positions.
hdfed> ! so it is often necessary to do an ’info -all’ first.
hdfed> info -all
    (1)     Version Descriptor: (Tag 30) Ref 1
    (2)     Scientific Data : (Tag 702) Ref 2
    (3)     Number type : (Tag 106) Ref 2
    (4)     SciData description : (Tag 701) Ref 2
   *(5)     SciData max/min : (Tag 707) Ref 2
    (6)     Numeric Data Group : (Tag 720) Ref 2
    (7)     Data Id Label: (Tag 104) Ref 3
    (8)     Scientific Data : (Tag 702) Ref 4
    (9)     Number type : (Tag 106) Ref 4
    (10)    SciData description: (Tag 701) Ref 4
    (11)    Numeric Data Group: (Tag 720) Ref 4
    (12)    Data Id Label : (Tag 104) Ref 5
hdfed>
hdfed> ! This predicate persists for the next and prev
hdfed> ! commands. That means if I now type another ’next’
hdfed> ! command, it will look for a tag that equals 707.
hdfed>
hdfed> next
Reached end of file. Not moved.
hdfed> info
    (5)      SciData max.min (SciData): (Tag 707) Ref: 2
hdfed>
hdfed> next group
hdfed> next group
hdfed> info
    (11)    Numeric Data Group : (Tag 720) Ref 4
hdfed>
```

The **dump** command

Syntax: dump [-offset *offset*] [-length *length*] [-decimal|-short|-byte|-octal|-hexadecimal|-float|-double|-ascii]

Flags:

-offset	Starting offset
-length	Length of the object to dump.
-decimal	Decimal format (32-bit integers)
-short	Decimal format (16-bit integers)

-byte	Decimal format (8-bit integers)
-octal	Octal format (the default)
-hexadecimal	Hexadecimal format
-float	Single-precision floating-point format (32-bit floats)
-double	Double-precision floating-point format (16-bit floats)
-ascii	ASCII format

Description: Displays the contents of the current object in the specified format.

Usage Example:

```
hdfed> ! to see the binary representation of this element
hdfed>
hdfed> dump
0: 257400004 257200004
hdfed>
hdfed> dump -short
hdfed>
0:  702     4   701 4
hdfed>
```

The **delete** command

Syntax: delete

Flags: None.

Description: Deletes the current object or group.

Usage Example:

```
hdfed> ! deleting groups
hdfed>
hdfed> ! If an element is required by other group it is alone.
hdfed> ! However, this is not perfect as the method by which group
hdfed> ! membership is determined can be pretty ad hoc.
hdfed>
hdfed> delete
hdfed> ! This deletes the Scientific Data Group
hdfed> info -all
    (1)     Version Descriptor: (Tag 30) Ref 1
    (2)     Scientific Data: (Tag 702) Ref 2
    (3)     Number type: (Tag 106) Ref 2
    (4)     SciData description : (Tag 701) Ref 2
    (5)     SciData max/min: (Tag 707) Ref 2
    (6)     Numeric Data Group : (Tag 720) Ref 2
    (7)     Data Id Label : (Tag 104) Ref 3
    (8)     Number type : (Tag 106) Ref 4
    (9)     Data Id Label : (Tag 104) Ref 5
hdfed>
hdfed> ! Notice that the Numeric Data Group with reference
hdfed> ! number 4 is missing, and now there are only 9
hdfed> ! objects in the file.
hdfed>
```

The **annotate** command

Syntax: annotate [-label] [-descriptor] [-editor editor]

Flags:

-label	Edit a label (the default)
-descriptor	Edit a descriptor.
-editor	Use an editor. (Default is the editor referred to by the `EDITOR` environment variable.

Description: Edits an annotation.

Usage Example:

```
hdfed>
hdfed> ! Annotations are labels and descriptors
hdfed>
hdfed> prev -group
hdfed> info -label
    (6)     Numeric Data Group: (Tag 720) Ref 2
            Label: Experiment #1
hdfed> annotate -editor /usr/ucb/ex
"/tmp/he5091.1" 1 line, 14 characters
:p
Experiment #1
:s/$/<more stuff>/
Experiment #1<more stuff>
:wq
"/tmp/he5091.1" 1 line 27 characters
hdfed> info -label
    (6)     Numeric Data Group: (Tag 720) Ref 2
            Label: Experiment #1 <more stuff>
hdfed>
```

The **write** command

Syntax: write [-attachto *tag-reference-number*] *filename*

Flags: -attachto Which element the annotation will be attached to. (only for writing annotations)

Description: Writes an element or group into another HDF file.

Usage Example:

```
hdfed>
hdfed> ! Write object or group to another HDF file.
hdfed>
hdfed> write test
hdfed>
hdfed> ! Let’s take a look at the file ’test’
hdfed> close; open test; info -all
    (1)     Version Descriptor(Tag 30) Ref 1
    (2)     Scientific Data(Tag 702) Ref 2
    (3)     Number type (Tag 106) Ref 2
    (4)     SciData description(Tag 701) Ref 2
    (5)     SciData max/min(Tag 707) Ref 2
   *(6)     Numeric Data Group (Tag 720) Ref 2
hdfed>
hdfed> close;
hdfed>
```

The **display** command

Syntax: display [-position *x-position y-position*] [-expansion *expansion*] [-large]

Flags:

-position	Image position on console screen
-expansion	Image expansion factor
-large	Make image as large as possible

Description: Displays image on screen.

Usage Example:

```
hdfed> ! We will open a file with some RIS8 images.
hdfed>
hdfed> open denm,HDF
hdfed> display
hdfed>
hdfed> ! The 'display' command displays the current RIS8
hdfed> ! group image via ICR. I.e. if you are using NCSA Telnet
hdfed> ! on a Mac II, this would display the images from denm.HDF
hdfed> ! on your screen.
hdfed> ! NOTE: not guaranteed to work otherwise.
hdfed>
```

The **putr8** command

Syntax: putr8 [-image *image_filename palette _ilename* -verbose]

Flags: -image Image file name template (Default is "img#.@.%")

-palette Palette file name template (Default is "pal#")

-verbose To give output of steps taken.

Description: Writes a RIS8 group into raw image and palette files.

Usage Example:

```
hdfed> ! putr8 puts an RIS8 group into raw files
hdfed>
hdfed> putr8 -image my_image.#.@.% -palette testPalettes# -verbose
Writing to file: my_image8.10.10
Writing to file: my_palette
hdfed>
```

The **close** command

Syntax: close [-keep]

Flags: -keep The backup file is not deleted.

Description: Closes the HDF file opened by the last open command.

Usage Example:

```
hdfed> close
hdfed>
```

The **select** command

Syntax: select *predicate_list command_list*

Flags: None.

Description: Step through all the elements in the HDF file that satisfies the predicates, and execute the command list.

Usage Example:

```
hdfed> ! To step through a file and, for example, putr8 on all
hdfed> ! RIS8 groups we can use the select command.
hdfed>
hdfed> select tag=306
>> putr8 -image testImages# -palette testPalettes# -verbose
>> end
Writing to file: testImages8
Writing to file: testPalettes8
Writing to file: test Images14
Writing to file: testPalettes14
Writing to file: testImages21
Writing to file: testPalettes21
hdfed>
hdfed> ! The 'select' and 'if' commands take the same
hdfed> ! predicates as 'next' and 'pref'. There are also
hdfed> ! the predicates 'succeed" and "fail" that test the
hdfed> ! return status of the 'last' command.
hdfed>
```

The **put** command

Syntax: put [-file *filename*] [-verbose]

Flags:

-file	Output file name (Default is "elt#.@")
-verbose	Output diagnostic information.

Description: Writes the raw binary image of the current object to a file.

Usage Example:

```
hdfed> ! The 'put' command writes an element into a binary file.
hdfed> ! This is a dumb routine and does not know about the
hdfed> ! formats of an element.
hdfed>
hdfed> put -file binary#
hdfed> put -file myBinary -verbose
Writing to file: myBinary
hdfed>
```

The **revert** command

Syntax: revert

Flags: None.

Description: Discards all changes made in the current hdfed session.

Usage Example:

```
hdfed> revert
hdfed>
```

The **getr8** command

Syntax: getr8 *image-file-name* [*x-dimension y-dimension*] [-palette *palette-file-name*] [-raster|-rle|-imcomp]

Flags:

-palette	Palette will be read from a binary file.
-raster	No compression will be performed during the write. (the default)
-rle	Run-length compression will be performed during the write.
-imcomp	IMCOMP compression will be performed during the write.

Description: Reads a RIS8 group from binary files.

The **if** conditional

Syntax: if *predicate-list command-list* end

Flags: None.

Description: Executes commands in a loop if predicates are satisfied for each element processed.

The **select** loop command

Syntax: select *predicate-list command-list* end

Flags: None.

Description: Executes the list of commands for each element that satisfies the predicates.

The **wait** command

Syntax: wait *message*

Flags: None.

Description: Prints a message, then waits for a carriage return to be typed.

15.19. Working with Both HDF4 and HDF5 File Formats

The document *Mapping HDF4 Objects to HDF5 Objects* defines a complete mapping between HDF4 and HDF5 objects. This document is available at `http://www.hdfgroup.org/HDF5/doc/ADGuide/H4toH5Mapping.pdf`.

This mapping is implemented by the H4toH5 Conversion Library and the **h4toh5** and **h5toh4** conversion utilities. These tools and further information regarding download, installation, and usage are available at `http://www.hdfgroup.org/h4toh5/`.

The H4toH5 Conversion Library is a C library providing APIs for customized conversion of individual objects from an HDF4 file to equivalent objects in an HDF5 file. The conversion follows the default mapping defined in the specification document, *Mapping HDF4 Objects to HDF5 Objects*. The library uses both the HDF4 and HDF5 libraries. Further information is available at `http://www.hdfgroup.org/h4toh5/libh4toh5.html`.

The **h4toh5** and **h5toh4** utilities are special-purpose tools developed for users who must convert files created with either an HDF4 or an HDF5 library to files that can be opened and manipulated by applications built on the other library. These utilities convert all supported objects in entire files and do not require the user to write any additional software. These utilities are documented in the *Tools* section of the *HDF5 Reference Manual*, which is available at http://www.hdfgroup.org/products/hdf5_tools/.

15.20. Converting an HDF File to a GIF File: hdf2gif

15.20.1. General Description

hdf2gif is a command line utility that converts files from the Hierarchical Data Format (HDF) (`http://www.hdfgroup.org`) to the Compuserve Graphics Interchange Format (GIF) (`http://www.w3.org/Graphics/GIF/spec-gif89a.txt`)

15.20.2. Command-line Syntax and Requirements:

hdf2gif takes two arguments: the name of the HDF file to read and the name of the GIF file to write.

```
hdf2gif <HDF file> <GIF file>
```

Inputs:	*HDF file*	Name of the HDF file
Outputs:	*GIF file*	Name of the GIF file

Requirements:

This utility requires the HDF library.

The HDF file is expected to contain 8-bit raster images which are consecutively converted to GIF images. At this time, this utility cannot be used to convert higher resolution images (16-bit, 24-bit, or 32-bit) to GIF images, which have a maximum resolution of 8-bit.

15.20.3. Structure of the GIF File

The GIF file may be of either GIF 87a or 89a formats. The choice between the two formats depends on the number of images stored in the HDF file. If there is only one image in the HDF file, then a GIF 87a file is written. If there are multiple images, a GIF89a file is written and it is animated with a time delay of 15ms between two consecutive images. The animation is set to loop indefinitely. The only exception occurs in case the HDF file was generated from a previous GIF file using the **gif2hdf** utility. In this case the original GIF file's values for animation and time out are taken into account instead of the preset defaults.

Depending on the version of the GIF file generated, the structure of the output file is as follows:

GIF87a:	The GIF file consists of a header, logical screen descriptor, image descriptor, local color table, image data, and the trailer. There is no global color table.
GIF89a:	The GIF file consists of a header, logical screen descriptor, and the `Netscape 2.0` application extension. This is followed by graphic control extension, image descriptor, local color table, and raw image data, in that order and repeated for every image present in the HDF file. The trailer follows and signifies the end of the GIF file. As in the GIF87a format, there is no global color table. The `Netscape 2.0` application extension is present to inform the GIF renderer the number of times the GIF animation should loop.

15.20.4. Building the Utility

hdf2gif is made when the utilites in the HDF 4 libraries are made.

Please refer to the instructions on how to make the HDF 4 libraries in order to make these utilities.

15.21. Converting an HDF File to a JPEG File: hdf2jpeg

15.21.1. General Description

hdf2jpeg is a command line utility that extracts JPEG images from an HDF file and writes them to a JPEG file.

15.21.2. Command-line Syntax and Requirements

hdf2jpeg takes two arguments: the name of the HDF file to read and the name of the JPEG file to write.

```
hdf2jpeg <HDF file> <JPEG file>
```

Inputs:	*HDF file*	Name of the HDF file
Outputs:	*JPEG file*	Name of the JPEG file

Requirements:

This utility requires the HDF library.

The HDF file is expected to contain JPEG images

Note that the utility only extracts JPEG images. If the HDF file also contains any non-JPEG images, **hdf2jpeg** will not extract them. If the HDF file does not contain any JPEG image, hdf2-jpeg will display: *"Error, no JPEG images found in HDF file"*.

15.21.3. Building the Utility

hdf2jpeg is created when the utilites in the HDF libraries are built.

Please refer to the instructions on how to build the HDF libraries in order to make these utilites.

15.22. Converting a GIF File to an HDF File: gif2hdf

15.22.1. General Description

gif2hdf is a command line utility to convert files from the Compuserve Graphics Interchange Format (GIF) (`http://www.w3.org/Graphics/GIF/spec-gif89a.txt`) to the Hierarchical Data Format (HDF) (`http://www.hdfgroup.org`).

15.22.2. Command-line Syntax and Requirements

gif2hdf takes two arguments: the name of the GIF file to read and the name of the HDF file to write.

```
gif2hdf <GIF file> <HDF file>
```

Inputs:	*GIF file*	Name of the GIF file
Outputs:	*HDF file*	Name of the HDF file

Requirements:

This utility requires the HDF library.

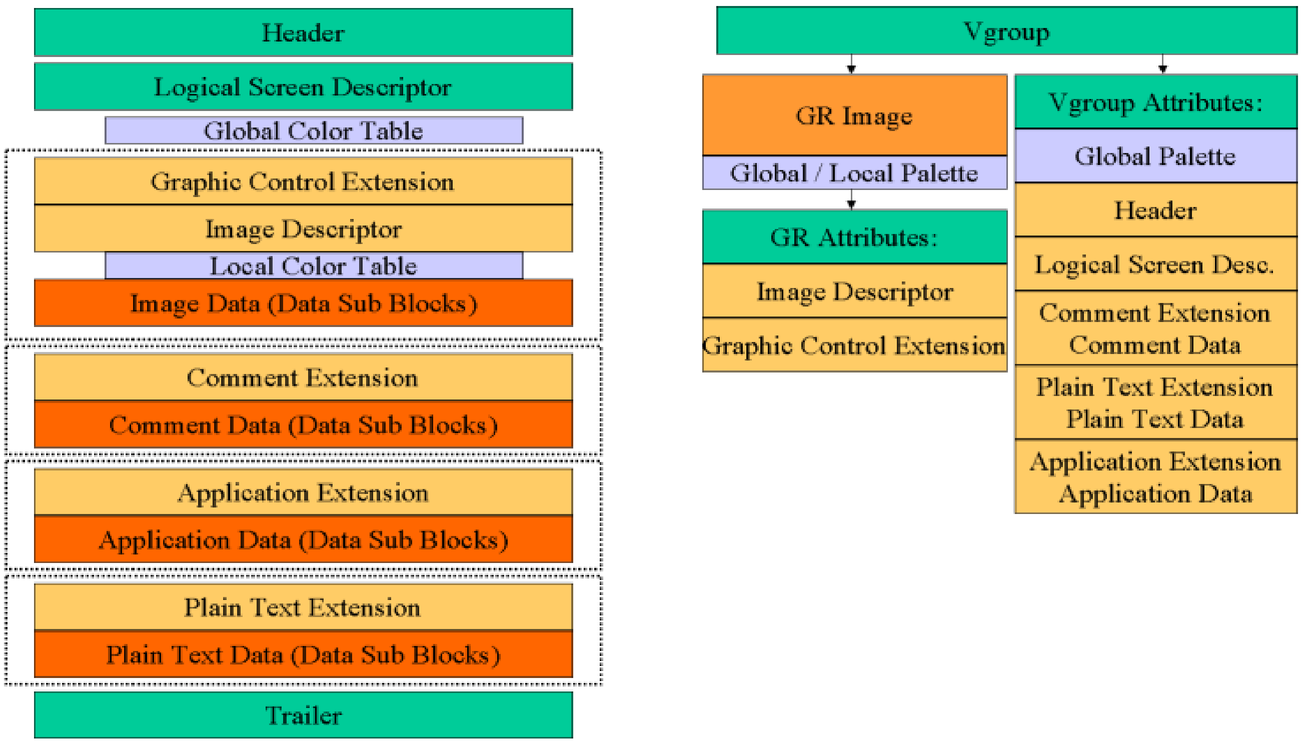
The amount of memory used by the program depends on the size of the GIF file and to some extent the type and quality of the image stored.

The GIF file being used as input must be a valid GIF87a or GIF89a file. If the file has multiple images (e.g., animated GIF), then the corresponding HDF file will contain all the images in a single Vgroup. Since HDF was not intended to be a format for animation, some information, such as the time between two consecutive images of an animation which is present in the GIF file, cannot be used by HDF tools. That information is, however, stored in the HDF file as an attribute.

15.22.3. Structure of the GIF and HDF Files and the Mapping between

Them

FIGURE 15b **Structure of the GIF and HDF files**



GIF File **HDF File**

The GIF file structure consists of a compulsory header followed by a logical screen descriptor. If the GIF file has a global color table, it follows the logical screen descriptor. The image descriptor precedes the raw image data. If the file is a GIF89a file, a graphic control extension may precede the image descriptor.

The comment extension, application extension, and plain text extension blocks are not compulsory and may appear any number of times within the GIF file. There is no preset order in which they must appear. These blocks are restricted to GIF89a files.

The final block is the trailer that consists only of one byte and signifies the end of the file. This block is compulsory.

For further information on the structure of a GIF file, refer to the GIF format specification at `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`.

The GIF images are stored in the HDF file in a Vgroup with `Class="GIF"` and *Name* being the name of the original GIF file. Prior to HDF version 4.2.4, the name and class are restricted to 64 characters, as set by `VSNAMELENMAX`. Starting from release 4.2.4, the maximum length of vgroup's name is no longer limited to `VGNAMELENMAX` (or `64`) and release 4.2.5 for vgroup's class name.

The GIF file contains a number of extensions that are all stored as attributes to the Vgroup, with the exception of the graphic control extension which is stored as an attribute to the individual image. In the case of comment extension, application extension, and plain text extension, there are two Vgroup attributes for every extension block: the extension dump attribute and the extension data attribute.

Each image in the GIF file is stored as a compressed GR image, using gzip compression, under the Vgroup in the HDF file. A palette is stored with each image in the HDF file. If the original GIF image contained a local color table, this table is stored as a palette. If the images contained only a global color table, each GR image in the HDF file has the global color table associated with it. This association of color tables enables an HDF viewer (such as HDFview, available from `http://www.hdfgroup.org/`) to correctly render the corresponding image. The image descriptor and the graphic control extension, if present, of the GIF file are attached to the GR image as attributes. If this HDF file is reconverted to the GIF format, the graphic control extension contains important information regarding the animation of those images.

15.22.4. Building the Utility

gif2hdf is made when the utilities in the HDF libraries are made.

Please refer to the instructions on how to make the HDF libraries.

15.23. Compiling C applications that Use HDF4: h4cc

15.23.1. General Description

Compiling the HDF4 library and HDF4 applications is a complex task, encompassing environment settings, particular use of compiler flags, many include files, etc. **h4cc** is a helper script, or wrapper, designed to assist in the task of compiling C applications that use HDF4 by providing several default settings and required flags and listing all of the required include files. Using **h4cc**, the user can take advantage of these defaults while retaining the options of setting environment variables to override the default compiler and linker and overriding the HDF4 include file and library locations on the command line.

h4cc subsumes all other compiler commands in that if a certain command has been used to compile the HDF4 library, then **h4cc** also uses that command. For example, if HDF4 was built using **gcc**, then **h4cc** will use **gcc** in compiling the new program.

Some programs use HDF4 in only a few modules. It is not necessary to use **h4cc** to compile those modules which do not use HDF4. In fact, since **h4cc** is only a convenience script, HDF4 modules can still be compiled in the normal way, taking care to properly specify the HDF4 libraries and include paths.

15.23.2. Command-line Syntax

The **h4cc** command-line syntax is as follows:

```
h4cc -help
h4cc [-echo] [-prefix=dir] [-show] compile_line
```

TABLE 15O **h4cc Options and Compiler Options**

`-help`	Prints a help message.
`-echo`	Shows all the shell commands executed.
`-prefix=`*`dir`*	The directory *`dir`* specifies the location of the HDF4 `lib/` and `include/` subdirectories. Default: the prefix specified when configuring HDF4.
`-show`	Shows the shell commands to be executed without actually executing them.
`compile_line`	The normal compile line options. **h4cc** uses the same compiler otherwise used to compile HDF4. Check the compiler manual for more information regarding the options required.

Several environment variables, listed in the following table, are available that provide another level of control over **h4cc**. When set, they override some of the built-in **h4cc** defaults.

TABLE 15P **Environment Variables**

`HDF4_CC`	Use a different C compiler.
`HDF4_CLINKER`	Use a different linker.

15.23.3. Examples

The following example illustrates the use of **h4cc** to compile the program `hdf_prog`, which consists of modules `prog1.c` and `prog2.c`:

```
# h4cc -c prog1.c
# h4cc -c prog2.c
# h4cc -o hdf_prog prog1.o prog2.o
```

15.24. Compiling Fortran applications that Use HDF4: h4fc

15.24.1. General Description

Compiling the HDF4 library and HDF4 applications is a complex task, encompassing environment settings, particular use of compiler flags, many include files, etc. **h4fc** is a helper script, or wrapper, designed to assist in the task of compiling Fortran applications that use HDF4 by providing several default settings and required flags and listing all of the required include files. Using **h4fc**, the user can take advantage of these defaults while retaining the options of setting environment variables to override the default compiler and linker and overriding the HDF4 include file and library locations on the command line.

h4cc subsumes all other compiler commands in that if a certain cpmmand has been used to compile the HDF4 library, then **h4fc** also uses that command. For example, if HDF4 was built using **f77**, then **h4cc** will use **f77** in compiling the new program.

Some programs use HDF4 in only a few modules. It is not necessary to use **h4fc** to compile those modules which do not use HDF4. In fact, since **h4fc** is only a convenience script, HDF4 modules can still be compiled in the normal way, taking care to properly specify the HDF4 libraries and include paths.

15.24.2. Command-line Syntax

The **h4fc** command-line syntax is as follows:

```
h4fc [-help]
h4fc [-echo] [-prefix=dir] [-show] compile_line
```

TABLE 15Q **h4fc Option Flags**

`-help`	Prints a help message.
`-echo`	Shows all the shell commands executed.
`-prefix=`*`dir`*	The directory *`dir`* specifies the location of the HDF4 `lib/` and `include/` subdirectories. Default: the prefix specified when configuring HDF4.
`-show`	Shows the shell commands to be executed without actually executing them.
`compile_line`	The normal compile line options. **h4fc** uses the same compiler otherwise used to compile HDF4. Check the compiler manual for more information regarding the options required.

Several environment variables, listed in the following table, are available that provide another level of control over **h4fc**. When set, they override some of the built-in **h4fc** defaults.

TABLE 15R **Environment Variables**

`HDF4_F77`	Use a different Fortran compiler.
`HDF4_F77LINKER`	Use a different linker.

15.24.3. Example

The following example illustrates the use of **h4fc** to compile the program `hdf_prog`, which consists of modules `prog1.f` and `prog2.f` and uses the HDF Fortran library:

```
# h4fc -c prog1.f
# h4fc -c prog2.f
# h4fc -o hdf_prog prog1.o prog2.o
```

15.25. Updating HDF4 Compiler Tools after an Installation in a New Location: h4redeploy

15.25.1. General Description

h4redeploy updates the HDF4 compiler tools after the HDF4 software has been installed in a new location.

15.25.2. Command-line Syntax

The **h4redploy** command-line syntax is as follows:

```
h4redeploy [help | -help]
h4redeploy [-echo] [-force] [-prefix=dir] [-tool=tool] [-show]
```

TABLE 15S **h4redeploy Option Flags**

`-help` `help`	Prints a help message.
`-echo`	Shows all the shell commands executed.
`-show`	Shows the shell commands to be executed without actually executing them.
`-force`	Performs the requested actions without offering any prompt requesting confirmation.
`-prefix=`*`dir`*	The directory *`dir`* specifies the location of the HDF4 `lib/` and `include/` subdirectories. Default: the current working directory.
`-tool=`*`tool`*	Specifies the tool to update. *`tool`* must be in the current working directory and must be writeable. Default: `h4fc`

CHAPTER 16 -- Raw Data Information

16.1. Chapter Overview

In 2011, to support the HDF4 File Content Map Project, HDF 4.2.6 introduced a set of routines that allow applications to access the raw data directly by providing the locations and sizes (i.e., offsets and lengths) of the data in an HDF file. The data can be all in one block or scattered in various locations due to linked-block or chunking storage scheme. This chapter describes these data information retrieval functions and provide examples of their usage.

16.2. The Data Information Retrieval Routines

There are several of the data information retrieval functions across the AN, SD, GR, V, and VS interfaces and the prefix of each function's name follows the same rule as other functions in the same interface. They all have "datainfo" in their names because their purpose is data information retrieval. Table 16A lists these routines. Currently, there is no implementation of the Fortran versions for these functions.

TABLE 16A **Raw Data Information Retrieval Routines**

Interface	Routine Name		Description and Reference
	C	FORTRAN-77	
AN	`ANgetdatainfo`	`unavailable`	Retrieves data information of an annotation's data (Section "*Retrieving Data Information of an Annotation: ANgetdatainfo"*)
SD	`SDgetanndatainfo`	`unavailable`	Retrieves data information of an DFSD API annotation's data (Section "*Retrieving Data Information of an Annotation in SD API: SDgetanndatainfo"*)
	`SDgetattdatainfo`	`unavailable`	Retrieves offset and length of an SD API attribute's data (Section "*Retrieving Data Information of an Attribute: SDgetattdatainfo"*)
	`SDgetdatainfo`	`unavailable`	Retrieves offset and length of a data set's data (Section "*Retrieving Data Information of an SDS: SDgetdatainfo"*)
	`SDgetoldat-tdatainfo`	`unavailable`	Retrieves offset and length of a DFSD API attribute's data (Section "*Retrieving Data Information of a DFSD API Attribute: SDgetol-dattdatainfo"*)
GR	`GRgetattdatainfo`	`unavailable`	Retrieves offset and length of a GR API attribute's data (Section "*Retrieving Data Information of a GR API Attribute: GRgetat-tdatainfo"*)
	`GRgetdatainfo`	`unavailable`	Retrieves offset and length of a raster image's data (Section "*Retrieving Data Information of a Raster Image: GRgetdatainfo"*)
V	`Vgetattdatainfo`	`unavailable`	Retrieves offset and length of a V API attribute's data (Section "*Retrieving Data Information of a V API Attribute: Vgetat-tdatainfo"*)
VS	`VSgetattdatainfo`	`unavailable`	Retrieves offset and length of a VS API attribute's data (Section "*Retrieving Data Information of a VS API Attribute: VSgetat-tdatainfo"*)
	`VSgetdatainfo`	`unavailable`	Retrieves offset and length of a vdata or a vdata field's data (Sec-tion "*Retrieving Data Information of a Vdata: VSgetdatainfo"*)

There is no additional header file required for these new functions. As with existing API functions, the header file `mfhdf.h` must be included in programs that invoke SD interface routines, and `hdf.h` for non-SD ones.

16.3. Addition to the AN Interface

There is one routine added to the AN API for raw data information retrieval, **ANgetdatainfo**, and it is described in the following sub-section.

16.3.1. Retrieving Data Information of an Annotation: ANgetdatainfo

ANgetdatainfo retrieves the offset and length locating the data in a specified annotation. The syntax of **ANgetdatainfo** is as follows:

```
C:          status = ANgetdatainfo(ann_id, &offset, &length);

FORTRAN:    Currently unavailable
```

The annotation is specified by its identifier, *ann_id*. The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that annotation's data is stored in one contiguous block only.

ANgetdatainfo returns `SUCCEED` (or `0`), if successful, or `FAIL` (or `-1`), otherwise. The parameters of **ANgetdatainfo** are specified in Table 16B.

TABLE 16B **ANgetdatainfo Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
ANgetdatainfo [intn] **(unavailable)**	ann_id	int32	N/A	Annotation identifier
	offset	int32 *	N/A	Buffer for offset of annotation's data
	length	int32 *	N/A	Buffer for length of annotation's data

16.4. Addition to the SD Interface

There are several functions added to the SD API for raw data information retrieval:

- **SDgetdatainfo** gets offsets/lengths of a data set's data
- **SDgetattdatainfo** gets offset/length of SD API attribute's data
- **SDgetoldattdatainfo** gets offset/length of DFSD API attribute's data
- **SDgetanndatainfo** gets offset/length of an DFSD API annotation's data

These functions are described in the following sub-sections.

16.4.1. Retrieving Data Information of an SDS: SDgetdatainfo

SDgetdatainfo retrieves offset and length of data blocks in a specified data set. The syntax of **SDgetdatainfo** is as follows:

```
C:         info_count = SDgetdatainfo(sds_id, origin, start_block, info_count,
                            offsetarray, lengtharray);

FORTRAN:   Currently unavailable
```

The offsets and lengths are retrieved into the user-supplied lists *offsetarray* and *lengtharray.*

- When the data set is contiguous, i.e., only one block of data, **SDgetdatainfo** will return a single pair of offset and length specifying the position of that data block.
- When the data set's data is stored in linked-blocks, **SDgetdatainfo** will return a list of offsets and a list of lengths, each matching offset/length pair specifying the position of a linked block.
- When the data set has chunked data without linked-block storage, **SDgetdatainfo** will return a single pair of offset and length and, with linked-block storage, two list of offsets and lengths specifying the blocks in the chunk.

The parameter *origin* must be `NULL` when the data is not stored in chunking layout. When the data is chunked, **SDgetdatainfo** can be called on a single chunk and *origin* is used to specify the coordinates of the chunk.

The parameter *info_count* specifies the maximum number of items the offset and length lists are allocated to hold. Applications, however, can pass in `0` for *info_count* and `NULL` for these arrays when only the actual number of data blocks in the data set is desired.

The purpose of the parameter *start_block* was to allow retrieval to start at a random block in the data. Applications would be able to start retrieving at the begining of the data by specifying *start_block* as `0`, or at a block of data by specifying *start_block* as a value between `1` and the number of blocks in the data. However, in release 2.6, *start_block* has no effect except for contiguous data, in which case, **SDgetdatainfo** will fail when *start_block* is greater than `1`. The supporting project did not need this specific feature. Thus, until the feature is supported, applications should pass `0` in for *start_block* to start retrieving at the beginning of the data and up to *info_count* or the total number of data blocks, whichever smaller.

SDgetdatainfo returns the number of offset/length pairs retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **SDgetdatainfo** are specified in Table 16C.

TABLE 16C **SDgetdatainfo Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDgetdatainfo [intn] **(unavailable)**	sds_id	int32	N/A	Data set identifier
	origin	int32 *	N/A	Coordinates of the origin of the chunk to be read
	start_block	uintn	N/A	Indicating where to start reading offsets
	info_count	uintn	N/A	Length of the offset and length lists
	offsetarray	int32 *	N/A	Array to hold offsets of the data blocks
	lengtharray	int32 *	N/A	Array to hold lengths of the data blocks

16.4.2. Retrieving Data Information of an Attribute: SDgetattdatainfo

SDgetattdatainfo retrieves offset and length of the data in a specified attribute. The syntax of **SDgetattdatainfo** is as follows:

```
C:         info_count = SDgetattdatainfo(id, attr_index, &offset, &length);

FORTRAN:   Currently unavailable
```

The attribute is specified by its index and can be one that belongs to an SD file, a data set, or a dimension. The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that attribute's data is stored in one contiguous block only.

There are attributes created by **SDsetattr** and those created by the DFSD API functions. Refer to Appendix C, *Attributes in HDF*, for more details. **SDgetattdatainfo** can only retrieve data information of attributes that were created by **SDsetattr**. If the inquired attribute was created by the DFSD API functions, **SDgetattdatainfo** will return to the caller with error code `DFE_NOVGREP` and the caller can call **SDgetoldattdatainfo** to get the attribute's data information.

SDgetattdatainfo returns the number of offset/length pair retrieved, which should be `1`, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **SDgetattdatainfo** are specified in Table 16D.

TABLE 16D **SDgetattdatainfo Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDgetattdatainfo [intn] **(unavailable)**	id	int32	N/A	SD, SDS, or dimension identifier
	attr_index	int32	N/A	Index of the attribute being inquired
	offset	int32 *	N/A	Buffer for offset of attribute's data
	length	int32 *	N/A	Buffer for length of attribute's data

16.4.3. Retrieving Data Information of a DFSD API Attribute: SDgetoldattdatainfo

SDgetoldattdatainfo retrieves offset and length of the data in a specified attribute, which was created by the DFSD API routines. The attributes created in this manner were not stored as vdatas like those created by **SDsetattr**. These type of attributes are often seen in some older files, circa 1993. However, later files may still contain them if the file was written with the DFSD API routines. In addition, this type of attributes can only be predefined; there are no user-defined attributes in DFSD API.

SDgetoldattdatainfo only works on DFSD-created attributes while its counter part **SDgetattdatainfo** only works on attributes created with **SDsetattr**. An application might call **SDgetattdatainfo** initially. When a DFSD-created attribute is encountered, **SDgetattdatainfo** will fail with the error code `DFE_NOVGREP`, which means there is no vgroup representation for the SDS and the SDS' attributes are stored differently than when they are created with **SDsetattr**. The application must call **SDgetoldattdatainfo** to get the data information of those attributes, if such error code is detected. For further information about this attribute issue, please refer to the Appendix C, *Attributes in HDF*, in this document. The syntax of **SDgetoldattdatainfo** is as follows:

```
C:          info_count = SDgetoldattdatainfo(dim_id, sds_id, attr_name, &offset,
                              &length);

FORTRAN:    Currently unavailable
```

SDgetoldattdatainfo takes both SDS identifier and dimension identifier if the inquired attribute belongs to a dimension. When the inquired attribute belongs to an SDS, the dimension identifier will not be needed, and should be `0`.

The attribute can be one that belongs to a data set or a dimension and is specified by its name, which can be one of the predefined names in Table 16E. The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that attribute's data is stored in one contiguous block only.

TABLE 16E **HDF4 Predefined Attributes**

Predefined Name	Actual Text	Applicable To
`_HDF_LongName`	`"long_name"`	`Dimension & SDS`
`_HDF_Units`	`"units"`	`Dimension & SDS`
`_HDF_Format`	`"format"`	`Dimension & SDS`
`_HDF_CoordSys`	`"coordsys"`	`Only SDS`
`_HDF_ScaleFactorErr`	`"scale_factor_err"`	`Only SDS`
`_HDF_AddOffset`	`"add_offset"`	`Only SDS`
`_HDF_ValidRange`	`"valid_range"`	`Only SDS`
`_HDF_ScaleFactor`	`"scale_factor"`	`Only SDS`
`_HDF_AddOffsetErr`	`"add_offset_err"`	`Only SDS`
`_HDF_CalibratedNt`	`"calibrated_nt"`	`Only SDS`
`_HDF_ValidMax`	`"valid_max"`	`Only SDS`
`_HDF_ValidMin`	`"valid_min"`	`Only SDS`
`_FillValue`	`"_FillValue"`	`Only SDS`

SDgetoldattdatainfo returns the number of offset/length pair retrieved, which should be `1`, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **SDgetoldattdatainfo** are specified in Table 16F.

TABLE 16F **SDgetoldattdatainfo Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
SDgetoldattdatainfo [intn] **(unavailable)**	dim_id	int32	N/A	Dimension identifier
	sds_id	int32	N/A	SDS identifier
	attr_name	char *	N/A	Name of the attribute being inquired
	offset	int32 *	N/A	Buffer for offset of attribute's data
	length	int32 *	N/A	Buffer for length of attribute's data

16.4.4. Retrieving Data Information of an Annotation in SD API: SDgetanndatainfo

SDgetanndatainfo retrieves offsets and lengths of the data belonging to the annotations of a given type. These annotations were created with the DFAN API. The syntax of **SDgetanndatainfo** is as follows:

```
C:          info_count = SDgetanndatainfo(id, annotype, size, offsetarray,
                             lengtharray);

FORTRAN:    Currently unavailable
```

The parameter *id* can be an SD or SDS identifier. However, when *id* is an SD identifier, the annotation's type must be either `AN_FILE_LABEL (or 2)` or `AN_FILE_DESC (or 3)`, and when it is an SDS identifier, the type must be `AN_DATA_LABEL (or 0)` or `AN_DATA_DESC (or 1)`. The offsets and lengths of the specified annotations are retrieved into the user-supplied buffers *offsetarray* and *lengtharray*. Note that annotation's data is stored in one contiguous block only, but there can be more than one annotation of the specified type. The parameter *size* specifies the number of elements *offsetarray* and *lengtharray* can hold.

SDgetanndatainfo returns the number of offset/length pairs retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **SDgetanndatainfo** are specified in Table 16G.

TABLE 16G **SDgetanndatainfo Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	Parameter	Parameter Type		Description
		C	FORTRAN-77	
SDgetanndatainfo [intn] **(unavailable)**	id	int32	N/A	SD or SDS identifier
	annotype	ann_type	N/A	Type of annotations to retrieve data info
	size	uintn	N/A	Length of the offset and length arrays
	offsetarray	int32 *	N/A	Buffer for offset of annotations' data
	lengtharray	int32 *	N/A	Buffer for length of annotations' data

16.5. Addition to the GR Interface

There are two routines added to the GR API for raw data information retrieval, **GRgetdatainfo** and **GRgetattdatainfo**, and they are described in the following sub-sections.

16.5.1. Retrieving Data Information of a Raster Image: GRgetdatainfo

GRgetdatainfo retrieves offset and length of data blocks in a specified raster image. The syntax of **GRgetdatainfo** is as follows:

```
C:          info_count = GRgetdatainfo(ri_id, start_block, info_count, offsetar-
                                ray, lengtharray);

FORTRAN:    Currently unavailable
```

The offsets and lengths are retrieved into the user-supplied lists *offsetarray* and *lengtharray.*

- When the raster image is contiguous, i.e., only one block of data, **GRgetdatainfo** will return a single pair of offset and length specifying the position of that data block.
- When the raster image's data is stored in linked-blocks, **GRgetdatainfo** will return a list of offsets and a list of lengths, each matching offset/length pair specifying the position of a linked block.
- **GRgetdatainfo** does not work with chunked images. (The HDF4 File Content Map Project did not need this feature.)

The parameter *info_count* specifies the maximum number of items the offset and length lists are allocated to hold. Applications, however, can pass in `0` for *info_count* and `NULL` for these arrays when only the actual number of data blocks in the data set is desired.

The purpose of the parameter *start_block* was to allow retrieval to start at a random block in the data. Applications would be able to start retrieving at the begining of the data by specifying *start_block* as `0`, or at a block of data by specifying *start_block* as a value between `1` and the number of blocks in the data. However, in release 2.6, *start_block* has no effect except for contiguous data, in which case, **GRgetdatainfo** will fail when *start_block* is greater than `1`. The supporting project did not need this specific feature. Thus, until the feature is supported, applications should pass `0` in for *start_block* to start retrieving at the beginning of the data and up to *info_count* or the total number of data blocks, whichever smaller.

GRgetdatainfo returns the number of offset/length pairs retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **GRgetdatainfo** are specified in Table 16H.

TABLE 16H **GRgetdatainfo Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRgetdatainfo [intn] **(unavailable)**	ri_id	int32	N/A	Raster image identifier
	start_block	uintn	N/A	Indicating where to start reading offsets
	info_count	uintn	N/A	Length of the offset and length lists
	offsetarray	int32 *	N/A	Array to hold offsets of the data blocks
	lengtharray	int32 *	N/A	Array to hold lengths of the data blocks

16.5.2. Retrieving Data Information of a GR API Attribute: GRgetattdatainfo

GRgetattdatainfo retrieves offset and length of the data in a specified attribute. The syntax of **GRgetattdatainfo** is as follows:

```
C:         info_count = GRgetattdatainfo(id, attr_index, &offset, &length);

FORTRAN:   Currently unavailable
```

The attribute is specified by its index and can be one that belongs to a GR file or a raster image. The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that attribute's data is stored in one contiguous block only.

GRgetattdatainfo returns the number of offset/length pair retrieved, which should be `1`, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **GRgetattdatainfo** are specified in Table 16I.

TABLE 16I **GRgetattdatainfo Parameter List**

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
GRgetattdatainfo [intn] **(unavailable)**	id	int32	N/A	GR or raster image identifier
	attr_index	int32	N/A	Index of the attribute being inquired
	offset	int32 *	N/A	Buffer for offset of attribute's data
	length	int32 *	N/A	Buffer for length of attribute's data

16.6. Addition to the V Interface

There is one routine added to the V API for raw data information retrieval, **Vgetattdatainfo**, and it is described in the following sub-section.

16.6.1. Retrieving Data Information of a V API Attribute: Vgetattdatainfo

Vgetattdatainfo retrieves the offset and length locating the data in a specified attribute. The syntax of **Vgetattdatainfo** is as follows:

```
C:         status = Vgetattdatainfo(vgroup_id, attr_index, &offset, &length);

FORTRAN:   Currently unavailable
```

The annotation is specified by its identifier, *ann_id*. The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that annotation's data is stored in one contiguous block only.

There are two types of attributes for vgroups; those created by **Vsetattr** (new style) and those created by non-**Vsetattr** approaches (old style.) Please refer to the section about **Vnattrs** and **Vnattrs2** and the Appendix Attribute in this *HDF User's Guide* for details. **Vgetattdatainfo** can access both type of attributes. However, an application must use **Vnattrs2** to get the number of attributes instead of **Vnattrs** in order to include both types. Note that, when a vgroup has both types of attributes, the old-style attributes will preceed the new ones, regardless of when they were created. The best way to access these attributes is through a loop.

Vgetattdatainfo returns the number of data blocks, which should be 1, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **Vgetattdatainfo** are specified in Table 16J.

TABLE 16J **Vgetattdatainfo Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
Vgetattdatainfo [intn] **(unavailable)**	vgroup_id	int32	N/A	Annotation identifier
	attr_index	intn	N/A	Index of the inquired attribute
	offset	int32 *	N/A	Buffer for offset of attribute's data
	length	int32 *	N/A	Buffer for length of attribute's data

16.7. Addition to the VS Interface

There are two routines added to the VS API for raw data information retrieval, **VSgetdatainfo** and **VSgetattdatainfo**, and they are described in the following sub-sections.

16.7.1. Retrieving Data Information of a Vdata: VSgetdatainfo

VSgetdatainfo retrieves offset and length of data blocks in a specified vdata. The syntax of **VSgetdatainfo** is as follows:

```
C:         info_count = VSgetdatainfo(vdata_id, start_block, info_count, offse-
                               tarray, lengtharray);

FORTRAN:   Currently unavailable
```

The offsets and lengths are retrieved into the user-supplied lists *offsetarray* and *lengtharray.*

- When the vdata has is contiguous data, i.e., only one block of data, **VSgetdatainfo** will return a single pair of offset and length specifying the position of that data block.
- When the vdata's data is stored in linked-blocks, **VSgetdatainfo** will return a list of offsets and a list of lengths, each matching offset/length pair specifying the position of a linked block.

The parameter *info_count* specifies the maximum number of items the offset and length lists are allocated to hold. Applications, however, can pass in `0` for *info_count* and `NULL` for these arrays when only the actual number of data blocks in the data set is desired.

The purpose of the parameter *start_block* was to allow retrieval to start at a random block in the data. Applications would be able to start retrieving at the begining of the data by specifying *start_block* as `0`, or at a block of data by specifying *start_block* as a value between `1` and the number of blocks in the data. However, in release 2.6, *start_block* has no effect except for contiguous data, in which case, **VSgetdatainfo** will fail when *start_block* is greater than `1`. The supporting project did not need this specific feature. Thus, until the feature is supported, applications should pass `0` in for *start_block* to start retrieving at the beginning of the data and up to *info_count* or the total number of data blocks, whichever smaller.

VSgetdatainfo returns a the number of offset/length pairs retrieved, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **VSgetdatainfo** are specified in Table 16K.

TABLE 16K **VSgetdatainfo Parameter List**

Routine Name [Return Type] **(FORTRAN-77)**	**Parameter**	**Parameter Type**		**Description**
		C	**FORTRAN-77**	
VSgetdatainfo [intn] **(unavailable)**	vdata_id	int32	N/A	Vdata identifier
	start_block	uintn	N/A	Indicating where to start reading offsets
	info_count	uintn	N/A	Length of the offset and length lists
	offsetarray	int32 *	N/A	Array to hold offsets of the data blocks
	lengtharray	int32 *	N/A	Array to hold lengths of the data blocks

16.7.2. Retrieving Data Information of a VS API Attribute: VSgetattdatainfo

VSgetattdatainfo retrieves offset and length of the data in a specified attribute. The syntax of **VSgetattdatainfo** is as follows:

```
C:          info_count = VSgetattdatainfo(vdata_id, findex, attr_index, &offset,
                                      &length);

FORTRAN:    Currently unavailable
```

The attribute is specified by its index, *attr_index*, and can be one that belongs to a vdata or a field of the vdata. If findex is `_HDF_VDATA` (or `-1`), then the attribute is associated with the vdata. If findex is an index of the vdata field, then the attribute is one that is associated with the vdata field. The parameter attr_index specifies the attribute's index within the vdata's or the field's attribute list. Thus, its value must be within [0-number of attributes of the associated list].

The offset and length are retrieved into the user-supplied buffers *offset* and *length*. Note that attribute’s data is stored in one contiguous block only.

VSgetattdatainfo returns the number of offset/length pair retrieved, which should be `1`, if successful, or `FAIL` (or `-1`), otherwise. The parameters of **VSgetattdatainfo** are specified in Table 16L.

TABLE 16L

VSgetattdatainfo Parameter List

Routine Name [Return Type] (FORTRAN-77)	Parameter	Parameter Type		Description
		C	FORTRAN-77	
VSgetattdatainfo [intn] **(unavailable)**	vdata_id	int32	N/A	Vdata identifier
	findex	int32	N/A	Vdata's field index or `_HDF_VDATA`
	attr_index	int32	N/A	Index of the attribute being inquired
	offset	int32 *	N/A	Buffer for offset of attribute's data
	length	int32 *	N/A	Buffer for length of attribute's data

EXAMPLE 1.

Getting Data Information of SDS.

This example demonstrates the use of the routine **SDgetdatainfo** with simple and contiguous data in a data set.

C:

```
#include "mfhdf.h"
#define SIMPLE_FILE  "datainfo_simple.hdf"   /* data file previously written */
main()
{
    /********************** Variable Declaration ************************/
    int32 sd_id, sds_id;
    int32 offset, length;
    uintn info_count = 0;
    intn status;

    /*
     * Open the file for reading.
     */
    sd_id = SDstart(SIMPLE_FILE, DFACC_READ);

    /*******************************************************************
     Read data info for later accessing data without the use of HDF4 library
     *******************************************************************/

    /*
     * Open the second dataset, get the number of data block, which is 1, then
     * retrieve and record the offset/length
     */
    sds_id = SDselect(sd_id, 1);

    /*
     * Passing in 0 for the info count and NULL for the offset and length
     * arrays to get the number of data blocks in the data set.  Note that
     * the second parameter is for chunk coordinates and because this data
     * set is not chunked, NULL should be passed in.  The third parameter
     * indicates to start retrieval at the beginning of the data.
     */
    info_count = SDgetdatainfo(sds_id, NULL, 0, 0, NULL, NULL);

    /*
     * Call SDgetdatainfo again to retrieve the offset and length of the
     * data block.  The info count is now 1 to specify the number of elements
     * in the offset and length arrays.
     */
    status = SDgetdatainfo(sds_id, NULL, 0, info_count, &offset, &length);
```

```
        /*
         * Terminate access to the data set.
         */
        status = SDendaccess(sds_id);

        /*
         * Close the file.
         */
        status = SDend(sd_id);

        /*******************************************************************
         Read data using previously obtained data info without HDF4 library
         *******************************************************************/

        /* Open file and read in data without using SD API */
        {
            int fd; /* for open */
            int32 ret32; /* for DFKconvert */
            ssize_t readlen = 0; /* for read */
            int32 *readibuf, *readibuf_swapped;

            /*
             * Open the file for reading without SD API.
             */
            fd = open(SIMPLE_FILE, O_RDONLY);

            /*
             * Forward to the position of the data.
             */
            lseek(fd, (off_t)offset, SEEK_SET);

            /*
             * Allocate buffers for SDS' data.
             */
            readibuf = (int32 *) HDmalloc(N_VALUES * sizeof(int32));
            readibuf_swapped = (int32 *) HDmalloc(N_VALUES * sizeof(int32));

            /*
             * Read in this block of data.
             */
            readlen = read(fd, (VOIDP) readibuf, (size_t)length);

            /*
             * Convert data back to format on local machine.
             */
            ret32 = DFKconvert(readibuf, readibuf_swapped, DFNT_INT32,
                               N_VALUES, DFACC_WRITE, 0, 0);

            /*
             * Free resources.
             */
            HDfree (readibuf_swapped);
            HDfree (readibuf);

            /*
             * Close the file.
             */
            close(fd);
        }
    }
```

Appendices

Appendix AReserved HDF Tags

A.1. Overview

This appendix includes tables containing brief descriptions of most of the tags that have been reserved for general use. This list will be expanded in future editions to include new tags as they are assigned. A more detailed description of the tags can be found in the *HDF Specification and Developer's Guide*. Also see the *HDF Specification and Developer's Guide* for a description of extended tags, which are not discussed in this Appendix.

Each table contains a list of tags within one category. The titles of the tables, with a functional description of each table, are:

- **Table A: The HDF Utility Tags.** Used by the HDF utilties.
- **Table B: The HDF General Raster Image Tags.** Used to describe aspects of raster image data.
- **Table C: The HDF Composite Image Tags.** Used to describe aspects of composite image data.
- **Table D: The HDF Scientific Data Set Tags:** Used to describe aspects of scientific data set (SDS) data.
- **Table E: The HDF Vset Tags.** Used to describe aspects of HDF Vset data.
- **Table F: The Obsolete HDF Tags:** Used to describe aspects of HDF data elements that have been replaced by newer tags or discontinued.

A.2. Tag Types and Descriptions

The following tables have five columns:

Tag Name contains the abbreviated symbolic names of tags that are often used in an augmented form in HDF programs.

Short Description contains a brief (four word maximum) description of the tag that is commonly used to describe to the tag in HDF manuals and in-line code documentation.

Data Size describes the type of data that is associated with the tag and, where possible, lists the data size.

Tag Value lists the numeric value of the tag symbol in the *hdf.h* header file.

Long Description contains a general description of the tag.

In the tables, the term *String* refers to a sequence of ASCII characters with the null byte possibly occurring at the end, but nowhere else. The term *Text* also refers to a sequence of ASCII characters, but it may contain null characters anywhere in the sequence. An *n* in the Data Size column describes a data unit of variable-length. For more detailed descriptions of these units of data, refer to the appropriate tag entry in the *HDF Specification and Developer’s Guide*.

TABLE AA

The HDF Utility Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_NULL	No Data	None	001	Used for place holding and filling up empty portions of the Data Descriptor Block.
DFTAG_VERSION	Library Version Number	4 bytes + string	030	Specifies the latest version of the HDF library used to write to the file.
DFTAG_NT	Number Type	4 bytes	106	Used by any other element in the file to specifically indicate what a numeric value looks like.
DFTAG_MT	Machine Type	0 bytes	107	Specifies that all unconstrained or partially constrained values in this HDF file are of the default type for that hardware.
DFTAG_FID	File Identifier	String	100	Points to a string that the user wants to associate with this file. This supports the inclusion of a user-supplied title for the file.
DFTAG_FD	File Descriptor	Text	101	Points to a block of text describing the overall file contents. It is intended to be user-supplied comments about the file.
DFTAG_TID	Tag Identifier	String	102	Provides a way to determine the meaning of a tag stored in the file.
DFTAG_TD	Tag Descriptor	Text	103	Similar to DFTAG_TD, but allows more text to be included.
DFTAG_DIL	Data Identifier Label	String	104	Associates the string with the Data Identifier as a label for whatever the identifier points to. By including DILs, any data element can be given a label for future reference. For example, this tag is often used to give titles to raster image data sets.
DFTAG_DIA	Data Identifier Annotation	Text	105	Associates the text block with the Data Identifier as an annotation for whatever that Data Identifier points to. With DIAs, and Data Identifier can have a lengthy, user-provided description of why that particular data element is in the file.
DFTAG_RLE	Run-length Encoding	0 bytes	011	Specifies that run-length encoding (RLE) is used to compress a raster image.
DFTAG_IMC	IMCOMP Compression	0 bytes	012	Specifies that IMCOMP compression is used to compress a raster image.
DFTAG_JPEG	24-bit JPEG Compression	n bytes	013	Provides header information for 24-bit JPEG-compressed raster images.
DFTAG_GREYPEG	8-bit JPEG Compression	n bytes	014	Provides header information for 8-bit JPEG-compressed raster images.

TABLE AB

The HDF General Raster Image Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_RIG	Raster Image Group	n*4 bytes	306	Lists the Data Identifiers (tag/reference number pairs) that uniquely describe a raster image set.
DFTAG_ID	Image Dimension	20 bytes	300	Defines the dimensions of the two-dimensional array the corresponding RI tag refers to.
DFTAG_LD	LUT Dimension	20 bytes	307	Defines the dimensions of the two-dimensional array the corresponding LUT tag refers to.

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_MD	Matte Dimension	20 bytes	308	Defines the dimensions of the two-dimensional array the corresponding MA tag refers to.
DFTAG_RI	Raster Image	x*y bytes	302	Points to a raster image data set.
DFTAG_CI	Compressed Image	n bytes	303	Points to a compressed raster image data set.
DFTAG_LUT	Lookup Table	n bytes	301	Table to be used by the hardware for the purpose of assigning RGB or HSV colors to data values.
DFTAG_MA	Matte Data	n bytes	309	Points to matte data.
DFTAG_CCN	Color Correction	n bytes	310	Specifies the gamma correction for the raster image and color primaries used in the generation of the image.
DFTAG_CFM	Color Format	String	311	Indicates the interpretation to be given to each element of each pixel in a raster image.
DFTAG_AR	Aspect Ratio	4 bytes	312	Indicates the aspect ratio of the image.
DFTAG_XYP	XY Position	8 bytes	500	Specifies the screen X-Y coordinate for raster image sets. (Also used for composite image sets - See the entry for DFTAG_XYP in Table 12.6)

TABLE AC

The HDF Composite Image Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
`DFTAG_DRAW`	Draw	n*4 bytes	400	Specifies a list of Data Identifiers (tag/reference number pairs) which define a composite image.
`DFTAG_XYP`	XY Position	8 bytes	500	Specifies the screen X-Y coordinate for composite image sets. (Also used for raster image sets - See the entry for DFTAG_XYP in Table 12.5)
`DFTAG_RUN`	Run	n bytes	401	Identifies code that is to be executes as a program or script.
`DFTAG_T14`	Tektronix 4014	n bytes	602	**Used as a vector image tag.** Points to a Tektronix 4014 data. The bytes in the data field, when read and sent to a Tektronix 4014 terminal, will be displayed as a vector image.
`DFTAG_T10S`	Tektronix 4015	n bytes	603	**Used as a vector image tag.** Points to a Tektronix 4015 data. The bytes in the data field, when read and sent to a Tektronix 4015 terminal, will be displayed as a vector image.

TABLE AD

The HDF Scientific Data Set Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
`DFTAG_NDG`	Numeric Data Group	n*4 bytes	720	Lists the Data Identifiers (tag/reference number pairs) that describe a scientific data set. Supersedes DFTAG_SDG.
`DFTAG_SDD`	SDS Dimension Record	n bytes	701	Defines the rank and dimensions of the array the corresponding SD refers to.
`DFTAG_SD`	Scientific Data	Real Number	702	Points to scientific data.
`DFTAG_SDS`	SCales	Real Number	703	Identifies the scales to be used when interpreting and displaying data.
`DFTAG_SDL`	Labels	String	704	Labels all dimensions and data.
`DFTAG_SDU`	Units	String	705	Displays units for all dimensions and data.
`DFTAG_SDF`	Formats	String	706	Displays formats for axes and data.
`DFTAG_SDM`	Maximum/minimum	2 Real Numbers	707	Displays the maximum and minimum values for the data.
`DFTAG_SDC`	Coordinate system	String	708	Displays the coordinate system to be used in interpreting data.
`DFTAG_SDLNK`	SDS Link	8 bytes	710	Links and old-style DFTAG_SDG and a DFTAG_NDG in cases where the DFTAG_NDG meets all criteria for a DFTAG_SDG.
`DFTAG_CAL`	Calibration Information	36 bytes	731	The calibration record for the corresponding DFTAG.SD.
`DFTAG_FV`	Fill Value	n bytes	732	The value which has been used to indicate unset values in the corresponding DFTAG_SD.

TABLE AE

The HDF Vset Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
`DFTAG_VG`	Vgroup	14+n bytes	1965	Provides a general-purpose grouping structure.
`DFTAG_VH`	Vdata Description	22+n bytes	1962	Provides information necessary to process a DFTAG_VS.
`DFTAG_VS`	Vdata	n bytes	1963	Contains a block of data that is to be interpreted according to the information in the corresponding DFTAG_VH.

TABLE AF

The Obsolete HDF Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
`DFTAG_IDS`	Image Dimension-8	4 bytes	200	Two 16-bit integers that represent the width and height of an 8-bit raster image in bytes.
`DFTAG_IP8`	Image Palette-8	768 bytes	201	A 256 x 3 byte array representing the red, green and blue elements of the 256-color palette respectively.
`DFTAG_RI8`	Raster Image-8	x*y bytes	202	A row-oriented representation of the elementary 8-bit image data.
`DFTAG_CI8`	Compressed Image-8	n bytes	203	A row-oriented representation of the elementary 8-bit raster image data, with each row compressed using a form of run-length encoding.
`DFTAG_II8`	IMCOMP Image-8	n bytes	204	A 4:1 8-bit raster image, compressed using the IMCOMP algorithm.
`DFTAG_SDG`	Scientific Data Group	n*4 bytes	700	List the Data Identifiers (tag/reference number pairs) that uniquely describe a scientific data set.
`DFTAG_SDT`	Transpose	0 bytes	709	Indicates that data is transposed in the file.

Appendix B HDF Installation Overview

B.1. General HDF Installation Overview

B.1.1. Acquiring the HDF Library Source

You may obtain the latest release of HDF source code and/or selected binaries from:

```
https://portal.hdfgroup.org/downloads/hdf4/hdf4_2_16-2.html
```

The source code can also be found at:

```
https://github.com/HDFGroup/hdf4
```

B.1.2. Building the HDF Library Source

For instructions on building HDF from the source code, please refer to the INSTALL file in the top directory of the HDF source tree.

Appendix C Attributes in HDF

This Appendix gives an overview of attributes in HDF and describes some issues in the library regarding attributes. The information is more helpful when users are working with files produced by older versions of the library.

C.1. Attribute Overview

Attributes are optional components in the HDF data model. They can be used to describe the nature and/or the intended usage of various HDF elements. This type of information is sometimes called user-created ***metadata*** because it is data about data. The HDF elements that can be assigned with attributes include:

- File, data set, and dimension in SD API
- File and raster image in GR API
- Vgroup in V API
- Vdata and vdata field in VS API

At the creation, an HDF attribute requires a name, data values, number type, and number of values. The attribute name is an ASCII string of any length from 1 to `H4_MAX_NC_NAME` (or `256`). The attribute data contains one or more values, in which case all the values must have the same number type as defined at the time the attribute is created. Attributes take the form label=value, where label is the attribute's name and value is the attribute's data. Number of values declares how many data entries the attribute has. The number type can be any type supported by the HDF library. These number types are listed in Table 1A, "Number Type Definitions" in Section I of the *HDF4 Reference Manual*.

For each attribute, an attribute count is maintained that identifies the number of values in the attribute. Each attribute has a unique attribute index, the value of which ranges from 0 to the total number of attributes minus 1. The attribute index is used to locate an attribute in the object which the attribute is attached to. Once the attribute is identified, its values and information can be retrieved.

There are two types of attributes in HDF: ***predefined attributes*** and ***user-defined attributes***.

Predefined attributes have reserved names and, in some cases, predefined number types and/or number of data entries. Predefined attributes are useful because they establish conventions that applications can depend on. They were first introduced in DFSD interface and later in the SD interface. They are further described in Section 3.10. "Predefined Attributes," of the *HDF User's Guide*. The GR interface was added in 1995 and has only one predefined attribute: `FILL_ATTR`, which is described in Section 8.10.1. "Predefined GR Attributes," of the *HDF User's Guide*.

User-defined attributes are defined by the calling program and contain auxiliary information about the element to which the attributes attach. HDF library provides in each interface of SD, GR, V, and VS a set of functions to add and access attributes. They are fully described in the associated chapters.

C.2. Underlaying storage issues

In general, users should not need the details described in this section, unless one is working with older HDF files (circa prior to 1993) and with raw data which relies on the knowledge of data layout in the file. The inclusion of this section in this User's Guide was prompted by the HDF4 File Content Map Project because various API functions being added to support this project require explanation that involves the layout of attributes in the file.

In the early years of HDF, in addition to the predefined attributes such as label, unit, and format, annotations were used to attach metadata to an HDF element such as data set and raster image. When the library was expanded to include user-defined attributes to SD and GR interfaces, metadata once stored as an annotation could be more conveniently stored as an attribute. This expansion introduced the difference in the ways predefined attributes were stored in DFSD interface and in SD/GR interfaces. The user-defined attribute feature then extended into the V and VS interfaces. Along the way, an incompatibility was inadvertently produced in the storage of attributes and their information. The next sections briefly explains these issues and their effects.

C.2.1. Predefined Attributes in DFSD API

Beginning in 1993, when the SD interface and user-created attribute were introduced, an attribute has been stored in a vdata of class `_HDF_ATTRIBUTE` (or "`Attr0.0`",) regardless it is a predefined or user-created attribute. However, prior to this period, there were only predefined attributes in DFSD API and they can be assigned to a data set or a dimension. This early predefined attribute of the data set is stored using tag/ref approach, that is, a pair of tag and ref would point to a string containing the values of the data set's attribute and the dimensions' attributes. The dimension attributes are stored following the SDS attribute. All attributes are separated by null characters. For example, in file `myfile`, there is a two-dimensional data set. The SDS and its dimensions were assigned with pre-defined attributes as followed:

Data set: ***label*** = "`SDS label`", ***unit*** = "`SDS unit`", ***format*** = <no attribute assigned>

Dimension 1: ***label*** = "`Dim1 label`", ***unit*** = <no attribute assigned>, ***format*** = "`Dim1 format`"

Dimension 2: ***label*** = "`Dim2 label`", ***unit*** = "`Dim2 unit`", ***format*** = "`Dim2 format`"

In the file, the attributes' values are stored as followed:

Data set's label attribute tag/ref `(DFTAG_SDL/<ref#>)`

| (point to)

--> "`SDS label<null>Dim1 label<null>Dim2 label<null>`"

Data set's unit attribute tag/ref `(DFTAG_SDU/<ref#>)`

| (point to)

--> "`SDS unit<null><null>Dim2 unit`"

Data set's format attribute tag/ref `(DFTAG_SDF/<ref#>)`

| (point to)

--> "`<null>Dim1 format<null>Dim2 format`"

A complete list of pre-defined attribute tags are provided in Table AG below.

TABLE AG **Pre-defined Attributes in the DFSD and SD APIs**

Tag Name	Description	Data Size	Applicable to
`DFTAG_SDL`	Labels	String	SDS and dimensions
`DFTAG_SDU`	Units	String	SDS and dimensions
`DFTAG_SDF`	Formats	String	SDS and dimensions
`DFTAG_SDM`	Maximum/minimum	2 Real Numbers	Only SDS
`DFTAG_SDC`	Coordinate system	String	Only SDS
`DFTAG_CAL`	Calibration Information	36 bytes	Only SDS
`DFTAG_FV`	Fill Value	n bytes	Only SDS

The HDF library handles the situation properly, so the difference in storage approaches does not effect general applications, which simply read the values of these predefined attributes. It would only become significant when an application needs to get access to the raw data. The HDF4 File Content Map Project is an example. The raw data of this type of attribute is not accessible by the function **SDgetattdatainfo**, which was added to support the HDF4 File Content Map Project. Thus, when such an attribute is encountered, **SDgetattdatainfo** will return the error code `DFE_NOVGREP` to the caller, which will in turn call **SDgetoldattdatainfo** to get the data information of that attribute.

C.2.2. Vgroup Attribute Without Vsetattr

HDF Version 4.0.2, July 19, 1996, and prior did not support attributes in Vgroup and Vdata as for SD and GR interfaces. However, an application could simulate an attribute for a vgroup by creating and writing a vdata of class `_HDF_ATTRIBUTE`, and then adding that vdata to the vgroup via these calls:

```
vdata_ref = VHstoredatam(file_id, ATTR_FIELD_NAME, values, size, type,
                                attr_name, _HDF_ATTRIBUTE, order);
ret_value = Vaddtagref (vgroup_id, DFTAG_VH, vdata2_ref);
```

For simplicity, this type of attributes is referred to as old-style attributes in this document.

A vgroup and vdata were having version number as `VSET_VERSION` (3). Starting in version 4.1.1, HDF began to support attributes in Vgroup and Vdata interfaces. Applications were able to add and manipulate attributes via public functions such as **Vsetattr**/**VSsetatt**, **Vgetattr**/**VSgetattr**, **Vattrinfo**/**VSattrinfo**,… This type of attributes is referred to as new-style attributes in this document. The version number of a vgroup or a vdata that has new-style attributes got promoted from `VSET_VERSION` (3) to `VSET_NEW_VERSION` (4).

In addition, the file format was changed for the vgroup/vdata header to store the number of attributes and the tag/reference number of each attribute. The new attribute API functions use this new information to get access to the attributes, but they are not aware of the old-style attributes. Thus, **Vnattrs** misses counting them and other functions like **Vattrinfo** and **Vgetattr** are unable to get to them.

Starting in version 4.2.6, the library provides the updated functions **Vnattrs2**, **Vattrinfo2**, and **Vgetattr2** for applications to get access to attributes that were not created by **Vsetattr**. These functions access both types of attributes. In addition, the HDF library provides the function **Vnoldattrs** to get the number of old-style attributes in a vgroup. The old-style attributes are likely to present in older files or files that were modified by older applications. Please refer to Section 5.8. "Vgroup Attributes," of the *HDF User's Guide* for details on these functions.

Appendix D Issue of Missing Palettes

This Appendix describes an issue regarding palettes in old and new raster image interfaces, that is, DF interfaces versus GR interface. The information may be helpful when users are working with files produced by older versions of the library.

D.1. Description

HDF4's representation of palettes and rasters has evolved over the lifetime of the library. As new representations were adopted, "old-style" representations were also written to the HDF4 file for backward compatibility. This practice occasionally introduce some issues inadvertently. This appendix presents one of those situations.

As discussed in Chapter 2 of the HDF4 Specification and Developer's Guide, the basic building blocks of an HDF4 file are data objects. A data object has two parts – a data descriptor (DD) and a data element (DE).

The original representation for palettes in HDF4 used DDs with tag `DFTAG_IP8` (`201`) while the later representation used DDs with tag `DFTAG_LUT` (`301`). Typically, when an HDF4 file has a pair of DDs with [tag 201, ref=R] and [tag 301, ref=R], they are old/new representations of the same palette and both refer to the same DE at the same offset. The DFP APIs and other GR APIs dealing with LUTs expect this behavior.

In some cases, an HDF4 file will have a 201 DD and a 301 DD that have the same reference number, but that refer to DEs at different offsets. In these cases, it is impossible to retrieve all the palette information in the file using the DFP and other GR APIs.

DFPgetpal attempts to read a DD with tag 201 first, and only attempts to read a DD with tag 301 and the same reference number if failure occurs for the first read. This effectively means that **DFPgetpal** cannot be used to retrieve a DD with tag 301/ref=R if a DD with tag 201/ref=R exists – even if the two DDs reference DEs at different offsets.

Another limitation of the DFP APIs is rooted in the fact that multiple palette DDs (for example 201/ref=2 and 201/ref=3) may refer to the same DE. This can be an issue even if both DDs in any 201/301 pair have the same offset.

DFPnpals returns the number of palette Data Elements in the file, not the number of palette Data Descriptors. Because multiple palette DDs can reference the same DE, the value (N) returned by **DFPnpals** cannot reliably be used as the upper bound on the number of calls needed to **DFPgetpal** to retrieve all the palettes in the file. Internally, **DFPgetpal** gets the next Palette DD and then uses the offset to retrieve the palette data. If multiple Palette DDs reference the same DE, then making N calls to **DFPgetpal** will not retrieve all of the DDs in the file (there will be more DDs than DEs). If the missed DDs reference DEs that don't appear in the first N DDs, then the palette data in those DEs will never be read.

D.2. Work-Around

To avoid changing the behavior of existing functions, **GRgetpalinfo** was added, starting in release 4.2.8, to get access to all palette DDs in an HDF4 file. With this information, the **Hgetelement** function can be used to retrieve the palette data from the DE associated with each DD. In addition, **GRgetpalinfo** also provides a way to retrieve palettes that are not associated with any raster image.