Introducing the HDF Reference Manual

I About this Manual

The *HDF Reference Manual* is designed to serve as a programmer's guide to the function calls and utilities supported by HDFv3.3. It contains fundamental descriptions, definitions, and examples that demonstrate how to use each function and utility in the HDF library. To use this manual most effectively:

See:		For:
Chapter 1:	Introduction to the HDF Function Calls and Utilities	A brief review of each HDF interface, its data model, and routines.
Chapter 2:	The HDF Function Calls	A complete alphabetical listing of the function calls available in the HDFv3.3 library.
Chapter 3:	The HDF Utilities	A complete alphabetical listing of the utilities available in the HDFv3.3 library.

II Conventions

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

1. `Courier` is used to identify all parameters and code examples.
2. *`Courier/Italic`* is used to identify all variable names.
3. **`Courier/Bold`** is used to identify the name of a function call or utility.

III Using Other HDF Documentation

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

The new HDF documentation will include an *HDF Specification and Developer's Guide*, *Getting Started with HDF* manual, *HDF User's Guide*, and an *HDF Reference Manual*. Of these documents, the *HDFv3.2 Specification and Developer's Guide, Getting Started with HDFv3.2,* and the *HDFv3.3 Reference Manual* are now complete. Until the *HDF User's Guide* is available:

If you need to:	**See the:**
Obtain a current version of HDF	NCSA ftp server (ftp.ncsa.uiuc.edu).
Install HDF	Current *README* on the ftp server.
Get quick help on an HDF function call	HDF man pages.
Learn basic HDF concepts	*Getting Started with HDFv3.2.*
Consult the *HDF Specification*	*HDF Specification and Developer's Guide.*
Consult the *HDF User's Guide*	*NCSA HDF Calling Interfaces and Utilities.*
Consult the *HDF Reference Manual*	*HDF Reference Manualv3.3.*

HDF Reference Manual

Version 3.3
February 1994

University of Illinois at Urbana-Champaign

Trademark Acknowledgments

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

Copyright Restrictions

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

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

Disclaimer

THE UNIVERSITY OF ILLINOIS GIVES NO WARRANTY, EXPRESS OR IMPLIED, FOR THE SOFTWARE AND/OR DOCUMENTATION PROVIDED, INCLUDING, WITHOUT LIMITATION, WARRANTY OF MERCHANTIBILITY AND WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

Chapter 1: Introduction to the HDF Function Calls and Utilities

1.0 Overview of the HDF Interfaces 1
1.1 The Low Level Interface 1
1.1.1 The H Interface 2
1.1.2 The HDF Interface 3
1.1.3 The HE Interface 3
1.2 Single File Application Interfaces 4
1.2.1 8-bit Raster Image Sets: The DFR8 Interface 4
1.2.2 Palettes: The DFP Interface 5
1.2.3 24-bit Raster Image Sets: The DF24 Interface 5
1.2.4 Scientific Data Sets: The DFSD Interface 6
1.2.5 Annotations: The DFAN Interface 8
1.3 Mu.lti-File Application Interfaces 9
1.3.1 Scientific Data Sets: The SD Interface 9
1.3.2 Scientific Data Sets: The NetCDF Interface 10
1.3.3 Vdata: The VS Interface 12
1.3.4 Vdata Query: The VSQ Interface 13
1.3.5 Vdata Fields: The VF Interface 13
1.3.6 Vgroups: The V Interface 14
1.3.7 High Level Vdata/Vgroups: The VH Interface 15
1.4 HDF Command Line Utilities 15

Chapter 2: HDF Function Call Reference Section

Chapter 3: HDF Utility Reference Section

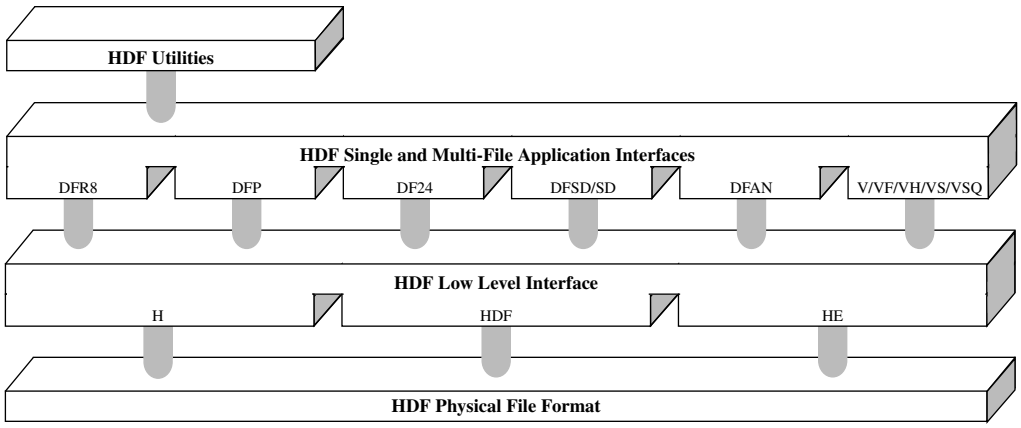
Chapter 1

Introduction to the HDF Function Calls and Utilities

1.0 Overview of the HDF Interfaces

The HDF library consist of three interface layers built upon a physical file format (See Figure 1.0). The first layer, or the ***low level interface***, is reserved for software developers because it provides support for low level details like file I/O, error handling, and memory management. The second layer, containing the ***single*** and ***multi-file application interfaces***, consists of a set of interfaces designed to simplify the process of storing and accessing data. The single file interfaces are limited to operating on one file at a time whereas the multi-file interfaces are not. Finally, the highest HDF layer includes a collection of command line ***utilities*** that operate on HDF files or the data objects they contain.

FIGURE 1.0 **The HDF interfaces and their respective layers.**



1.1 The Low Level Interface

The first interface layer, or the ***low level interface***, is the layer of HDF reserved for software developers. Through its H, HDF, and HE routines, the low level interface provides routines for error handling, file I/O, memory management, and physical storage. For a more detailed discussion of the low level interface than that provided here, consult the *HDF Specification and Developer's Guide*.

1.1.1 The H Interface

The low level ***H interface*** provides a collection of routines for managing HDF files. It is called the H interface because all the routines begin with the letter H. The H routines are divided into the following six categories:

- **File access routines** are used to open and close HDF files.
- **Reading and writing routines** access data objects and allow part or all of a data element to be written or read from an HDF file.
- **Special data object routines** create "linked blocks" and "external elements."
- **General file maintenance routines** truncate data elements to a specified length, and synchronize data on disk with data in memory**.**
- **Data descriptor routines** manipulate the data descriptor block without affecting the data objects stored in the file.
- **Query routines** locate specific data objects in a file, or return information regarding the contents of an HDF file.

Prior to HDFv3.2, all low level routines began with the prefix DF. As of HDFv3.3, the ***DF interface*** is no longer recommended for use. It is only supported to maintain backward compatibility with programs and files created under earlier version of the HDF library.

The H function calls are more explicitly identified in Table 1.1.1 and on their respective pages in Chapter 2. There are no FORTRAN routines for most of the H interface because this interface is intended primarily for the manipulation of existing C data structures.

TABLE 1.1.1 Functions Available in the H Interface

Purpose	Function Name		Description
	C	FORTRAN	
Accessing HDF files	`Hopen`	`hopen`	Provides an access path to an HDF file
	`Hclose`	`hclose`	Closes the access path to the file
Reading and writing data elements	`Hstartread`	None	Locates an object by tag and ref number and returns its descriptor
	`Hnextread`	None	Locates and positions a read access marker
	`Hstartwrite`	None	Initializes the write operation
	`Hendaccess`	None	Disposes of descriptor for tag/ref
	`Hseek`	None	Seeks to a position in a data element
	`Hread`	None	Reads part of a data element.
	`Hwrite`	None	Writes next data segment to data element.
	`Hgetelement`	None	Obtains the data referred to by the tag and ref number passed to it
	`Hputelement`	None	Adds or replaces an element in a file
Creating special data elements	`HLcreate`	None	Creates or transforms a linked block data element
	`HXcreate`	None	Creates or transforms an external file data element
General file maintenance	`Htrunc`	None	Truncate a data element to a length
	`Hsync`	None	Synchronize on-disk HDF file with image in memory

Purpose	Function Name		Description
	C	FORTRAN	
Manipulating data descriptors	Hlength	None	Returns the length of a data element in bytes
	Hoffset	None	Gets the offset (in bytes) of a data element in the file
	Hdupdd	None	Generate new references to already referenced data
	Hdeldd	None	Delete a tag and reference number from the list of data descriptors
	Hnewref	None	Returns a reference number that is guaranteed to be unique in the file
Querying HDF files	Hfind	None	Locate the next object of a search in an HDF file
	Hinquire	None	Returns access information of a data object
	Happendable	None	Allows appending to the end of a data element.
	Hishdf	None	Determine if a file is an HDF file
	Hnumber	hnumber	Find the number of occurrences of tag and reference number in file
	Hgetlibversion	None	Get version information for current HDF library
	Hgetfileversion	None	Get version information for an HDF file
	Hfidinquire	None	Inquire about a file ID

1.1.2 The HDF Interface

As HDF begins expanding its interfaces to include multi-file support for each data model, it is anticipated that the some H routines will evolve into an ***HDF interface***. These routines will use the prefix HDF. In HDFv3.3, there are only two such routines, **`HDFopen`** and **`HDFclose`**. **`HDFopen`** and **`HDFclose`** currently operate as macros for **`Hopen`** and **`Hclose`** respectively.

1.1.3 The HE Interface

The HDF library incorporates an error stack via the ***HE interface***. In addition to other functions, the HE routines add information onto the error stack, print information from the stack, and clear the stack. The HE function calls are more explicitly identified in Table 1.1.3 and on their respective pages in the reference section of this manual.

TABLE 1.1.3 Functions Available in the HE Interface

Purpose	Function Name		Description
	C	FORTRAN	
Error reporting	HEclear	None	Clears the error stack
	HEprint	heprnt	Prints the errors on the error stack to a specified file
	HEpush	None	Pushes a new error onto the stack
	HEreport	None	Adds text to the description of the most recent error
	HEstring	None	Returns the error message associated with a error code
	HEvalue	None	Returns the nth most recent error

1.2 Single File Application Interfaces

The HDF ***single file application interfaces*** include several independent modules each designed to simplify the process of storing and accessing a specific type of data. Single file interfaces are so called because access is limited to one file at a time. These interfaces support the 8-bit raster image(DFR8), 24-bit raster image (DF24), palette (DFP), scientific data (DFSD), and annotation (DFAN) models. All single file interfaces are built upon the H routines; unless otherwise specified, all the low level details can be ignored.

1.2.1 8-bit Raster Image Sets: The DFR8 Interface

The HDF ***8-bit raster interface*** provides a collection of routines for managing 8-bit raster image sets. Any 8-bit raster image accompanied by its dimension record is recognized as an ***8-bit raster image set***. Raster image sets may also include a palette.

Every function in the 8-bit raster interface begins with the prefix DFR8. The equivalent FORTRAN functions use the prefix d8. The DFR8 routines are divided into three groups:

- **Write routines** are used to add image sets to a file, create raster image sets containing palettes, and assign specific tag and reference numbers to individual raster image sets.
- **Read routines** are used to determine the dimensions of an image, read an image set, and select a specific image set.
- **General interface routines** are used to determine the reference number of a raster image set, return the number of raster image sets in a file, and return to the beginning of a file to access multiple raster image sets in sequential order.

The DFR8 function calls are more explicitly identified in Table 1.2.1 and on their respective pages in the reference section of this manual. To learn more about using the ***8-bit raster data model*** or the DFR8 interface consult *Getting Started with HDF* or the *HDF User's Guide*.

TABLE 1.2.1 Functions Available in the DFR8 Interface

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Writing 8-bit raster images	DFR8addimage	d8aimg	Appends another 8-bit image to a file
	DFR8putimage	d8pimg	Writes an 8-bit image to a file
	DFR8setpalette	d8spal	Sets palette for multiple 8-bit images
	DFR8writeref	d8iwref	Stores image using specified reference number
	DFR8setcompress	d8scomp	Sets the compression for the next 8-bit write operation
Reading 8-bit raster images	DFR8getdims	d8gdims	Retrieves dimensions for 8-bit image
	DFR8getimage	d8gimg	Retrieves an 8-bit image and its palette
	DFR8readref	d8irref	Locates an image with a specified reference number
General interface	DFR8lastref	d8lref	Returns reference number of last element accessed
	DFR8nimages	d8inims	Returns number of images in a file
	DFR8restart	d8first	Ignores info about the last access and restarts from beginning

1.2.2 Palettes: The DFP Interface

The HDF ***palette interface*** provides a collection of routines for managing ***palette data***. This interface is most often used for working with multiple palettes stored in a single file or palettes not specifically assigned to a raster image.

Every function in the palette interface begins with the prefix DFP. The equivalent FORTRAN functions use the prefix dp. The DFP routines are divided into three categories:

- **Write routines** are used to add palettes to HDF files, overwrite existing palettes, and assign specific tag and reference numbers to individual palettes.
- **Read routines** are used to read palettes from a file in sequential order or as specified by tag and reference number.
- **General interface routines** are used to determine the reference number of the last palette written to file, return the number of palettes in a file, and return to the beginning of a file to access multiple palettes in sequential order.

The DFP routines are more explicitly defined in Table 1.2.2 and on their respective pages in the reference section of this manual. To learn more about the ***palette model*** or the DFP interface consult *Getting Started with HDF* or the *HDF User's Guide*.

TABLE 1.2.2 Functions Available in the DFP Interface

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Writing palettes	`DFPaddpal`	`dpapal`	Appends a palette to a file
	`DFPputpal`	`dpppal`	Writes a new palette to a file
	`DFPwriteref`	`dpwref`	Sets the reference number for writing the next palette
Reading palettes	`DFPgetpal`	`dpgpal`	Retrieves the next palette in a file
	`DFPreadref`	`dprref`	Sets reference number for retrieving the next palette
General interface	`DFPlastref`	`dplref`	Returns the value of the last reference number read or written
	`DFPnpals`	`dpnpals`	Returns the number of palettes in a file
	`DFPrestart`	`dprest`	Ignores info about the last access and restarts from beginning

1.2.3 24-bit Raster Image Sets: The DF24 Interface

The HDF***24-bit raster interface*** provides a collection of routines for managing 24-bit raster image sets. Any 24-bit raster image array accompanied by its dimension record is recognized as a ***24-bit raster image set***.

Every function in the 24-bit raster interface begins with the prefix DF24. The equivalent FORTRAN functions use the prefix d2. DF24 routines are divided into three categories:

- **Write routines** are used to add 24-bit raster image sets to a file, define the dimensions and interlacing, and assign tag and reference numbers to 24-bit raster image sets.
- **Read routines** are used to determine the dimensions and interlacing for 24-bit images, and to read images both sequentially and via their the tag and reference numbers.

- **General interface routines** report the reference number of the last raster image set written, return the number of raster sets in a file, and restart at the beginning of a file.

The DF24 function calls are more explicitly defined in Table 1.2.3 and on their respective pages in the reference section of this manual. To learn more about the ***24-bit raster model*** or the DF24 interface, consult *Getting Started with HDF* or the *HDF User's Guide*.

TABLE 1.2.3 Functions Available in the DF24 Interface

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Writing 24-bit raster images	DF24addimage	d2iaimg	Appends another 24-bit image to a file
	DF24putimage	d2pimg	Writes an 24-bit image into a file
	DF24setcompress	d2scomp	Sets the compression for the next 24-bit write operation
	DF24setdims	d2sdims	Sets the dimensions of image
	DF24setil	d2setil	Sets the interlacing for the image
	DF24writeref	d2wref	Stores an image using a specified reference number
Reading 24-bit raster images	DF24getdims	d2igdims	Retrieves dimensions for 24-bit image
	DF24getimage	d2igimg	Retrieves a 24-bit image
	DF24readref	d2rref	Locates an image with a specified reference number
	DF24reqil	d2reqil	Request interlace when returning image
General interface	DF24lastref	d2lref	Returns the reference number of last element accessed
	DF24restart	d2first	Ignores info about the last access and restarts from beginning

1.2.4 Scientific Data Sets: The Single File DFSD Interface

There are two HDF interfaces that support multi-dimensional arrays: the single file DFSD interface described here, which permits access to only one file at a time, and the newer SD interface, which permits simultaneous access to more than one file. The SD interface is covered more thoroughly in the multi-file section of this manual.

The single file ***scientific data set interface*** provides a collection of routines for reading and writing arrays of arbitrary rank and number type. Any array accompanied by a record of its rank and number type qualifies as a ***scientific data set***. Scientific data sets may also include predefined ***attribute records***. Attribute records describes one particular facet of the environment from which the scientific data or its dimensions were taken.

Every function in the single file interface begins with the prefix DFSD. The equivalent FORTRAN functions use the prefix ds. DFSD routines are divided into five categories:

- **Read and write routines** create and read scientific data sets by defining their dimensions, rank, number type, fill value, data range, and calibration information.
- **Slab routines** read and write subarrays of scientific data to existing data sets.
- **Data set attribute routines** are used to describe the format, units, labels and coordinate systems assigned to a data set. All attributes are optional.
- **Dimension attribute routines** are used to read and write the format, units, labels, and scales assigned to the dimensions in a data set.

- **General interface routines** are used to determine the reference number of a data set, return the number of data sets in a file, and support hyperslab operations.

The DFSD function calls are more explicitly defined in Table 1.2.4 and on their respective pages in the reference section of this manual. For more information on using the ***scientific data model*** or interface, consult the *Getting Started with HDF* or the *HDF User's Guide*.

TABLE 1.2.4 **Functions Available in the DFSD Interface**

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Reading and writing scientific data sets	DFSDadddata	dsadata	Appends a data set to the file
	DFSDgetcal	dsgcal	Retrieves the calibration information for the data set
	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
	DFSDputdata	dspdata	Overwrites data in a file with new data
	DFSDsetcal	dsscal	Sets the calibration information for the data set
	DFSDsetdims	dssdims	Sets the rank and dimension for the succeeding data set
	DFSDsetNT	dssnt	Sets the number type for the data set
Reading and writing slabs of scientific data	DFSDendslab	dseslab	Terminates a read or write slab operation
	DFSDgetfillvalue	dsgfill	Retrieves the fill value used to complete the data set
	DFSDreadslab	dsrslab	Reads a sub-array of data from a data set
	DFSDsetfillvalue	dssfill	Sets the fill value to use when completing a data set
	DFSDstartslab	dssslab	Begins a read or write slab operation
	DFSDwriteslab	dswslab	Writes a sub-array of data to a data set
Working with data set attributes	DFSDgetdatalen	dsgdaln	Gets the length of the attributes assigned to the data
	DFSDgetdatastrs	dsgdast	Returns label, unit and format for data
	DFSDgetrange	dsgrang	Gets the range of values for the data set
	DFSDsetdatastrs	dssdast	Sets label, unit and format for data
	DFSDsetlengths	dsslens	Sets the length for the data set and dimension attributes
	DFSDsetrange	dssrang	Sets the range of values for the data set
Working with dimension attributes	DFSDgetdimlen	dsgdiln	Gets the length of the attributes assigned to the dimension
	DFSDgetdimscale	dsgdisc	Returns the scale for a dimension
	DFSDgetdimstrs	dsgdist	Returns the label, unit, and format for a dimension
	DFSDsetdimscale	dssdisc	Sets the scale for a dimension
	DFSDsetdimstrs	dssdist	Sets label, unit and format for the dimension
General interface	DFSDclear	dsclear	Clears all possible set values
	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 HDFv3.2
	DFSDreadref	dsrref	Locates a data set with a specified reference number
	DFSDrestart	dsfirst	Sets the access command to the first data set in the file
	DFSDlastref	dslref	Returns the reference number of last data set accessed
	DFSDwriteref	dswref	Assigns a specific reference number to the next data set written

1.2.5 Annotations: The DFAN Interface

The ***annotation interface*** provides a collection of routines for reading and writing text strings assigned to HDF data objects or files. Annotations consist of labels and descriptions. A ***label*** is a sequence of characters terminated by '\0'. A ***description*** is any sequence of characters which may contain one or more '\0' characters.

Every function in the annotation interface begins with the prefix DFAN. The equivalent FORTRAN functions use the prefix da. In the DFAN interface, file labels are abbreviated as ***fid***, and file descriptions are abbreviated as ***fds***. Labels and descriptions assigned to data objects are abbreviated ***lab*** and ***desc*** respectively. The DFAN routines are divided into the following three categories:

- **Write routines** are used to add file labels, file descriptions, object labels, and object descriptions to a file.
- **Read routines** are used to determine the length of an annotation, and to read annotations from a file.
- **General interface routines** are used to list the labels for a given tag, and to return the reference number of the last annotation accessed.

The DFAN function calls are more explicitly defined in Table 1.2.5 and on their respective pages in the reference section of this manual. For more information on using the ***annotation model*** or DFAN interface, consult *Getting Started with HDF* or the *HDF User's Guide*.

TABLE 1.2.5 Function Calls Available in the DFAN Interface

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Writing annotations	DFANputlabel	daplab	Stores an object label
	DFANputdesc	dapdesc	Stores an object description
	DFANaddfid	daafid	Adds a file label to a file
	DFANaddfds	daafds	Adds a file description to a file
Reading annotations	DFANgetlablen	dagllen	Returns length of an object label
	DFANgetlabel	daglab	Returns label object
	DFANgetdesclen	dagdlen	Returns length of object description
	DFANgetdesc	dagdesc	Returns an object description
	DFANgetfidlen	dagfidl	Returns length of file label
	DFANgetfid	dagfid	Returns file label
	DFANgetfdslen	dagfdsl	Returns length of file description
	DFANgetfds	dagfds	Returns file description
General interface	DFANlablist	dallist	Returns a list of labels for a tag type
	DFANlastref	dalref	Returns reference of last annotation accessed

1.3 Multi-File Application Interfaces

The HDF ***multi-file interfaces*** include several modules designed to allow operations on more than one file and more than one data object at the same time. The multi-file interfaces provided in HDFv3.3 are the SD, VS, VSQ, VF, V, and VH interfaces. The SD interface is the multiple file version of the scientific data set interface. It also includes expanded support for interoperability with the netCDF library. The VS, VSQ, and VF interfaces support the Vdata model and have always provided multiple file access. Similarly, the V and VH interfaces also provide multiple file access for the Vgroup data model.

Like the single file interfaces, the multi-file interfaces are built upon the low level H routines. Unlike single file operations however, operations performed via a multi-file interface are not implicitly preceded by **`Hopen`** and followed by **`Hclose`**. Instead, each series of operations on a file must be preceded by an explicit call to open and close the file. Once the file is opened, it remains open until an explicit call is made to close it. Although this process is a bit less convenient for operations on individual files, it allows for operations on more than one file at a time.

1.3.1 Scientific Data Sets: The SD Interface

The ***scientific data set interface*** provides a collection of routines for reading and writing arrays of arbitrary dimension and number type. Multi-dimensional arrays accompanied by a record of their dimension and number type are called ***scientific data sets***. Under the multi-file interface, scientific data sets may include predefined or user defined ***attribute records***. Each attribute record is optional and describes a particular facet of the environment from which the scientific data was taken.

Every function in the multi-file scientific data interface begins with the prefix SD. The equivalent FORTRAN functions use the prefix sf. The SD routines are divided into the following four categories:

- **Access routines** initialize and close the SD interface.
- **Create, read, and write routines** create, read, and write scientific data sets by defining or reading their dimensions, rank, number type, fill value, data range, and calibration information.
- **Dimension attribute routines** describe the attributes of data set dimensions including the name, format, unit, label, and scales. All attributes are optional.
- **General attribute routines** manage information about ***local attributes*** (attributes assigned to a data set) and ***global attributes*** (attributes assigned to the file). Predefined local attributes include the coordinate system, format, labels, max/min values, scales, and units. Also included are routines for user defined local and global attributes.

The SD function calls are more explicitly defined in Table 1.3.1 on page 10 and on their respective pages in the reference section of this manual. For more information about using the multi-file ***scientific data model*** or the SD interface, consult the *HDF User's Guide*.

TABLE 1.3.1 Function Calls Available in the SD Interface

Purpose	Function Name			Description
	Long Name (C/Fortran)	Short Name (Fortran)	DFSD Equivalent	
Access the SD interface	SDend	sfend	None	Close the file and clean up memory
	SDstart	sfstart	None	Initializes the SD interface
Reading and writing data sets	SDcreate	sfcreate	None	Create a new data set
	SDendaccess	sfendacc	None	Dispose of a data set ID
	SDfileinfo	sffinfo	None	Return info about the file contents
	SDgetinfo	sfginfo	None	Return information about a data set
	SDidtoref	sfid2ref	None	Returns a reference number for a named data set.
	SDiscoordvar	sfiscvar	None	Distinguishes between a data set and a coordinate variable
	SDnametoindex	sfn2index	None	Return an index for a named data set.
	SDreaddata	sfrdata	DFSDgetdata	Read a slab of data from a data set
	SDselect	sfselect	None	Gets the sds_id for a specific data set
	SDwritedata	sfwdata	DFSDadddata	Write a slab of data for a data set
Working with dimensions	SDdiminfo	sfdinfo	None	Get information about a dimension
	SDgetdimid	sfdimid	None	Get a pointer to a dimension
	SDgetdimscale	sfsdscale	DFSDgetdimscale	Retrieve scale values for a dimension
	SDsetdimname	sfsdmname	None	Associate a name with a dimension
	SDsetdimscale	sfgdscale	DFSDsetdimscale	Store the values of this dimension
Working with attributes	SDattrinfo	sfgainfo	None	Get information about an attribute
	SDfindattr	sffattr	None	Find the index for named attribute
	SDgetcal	sfgcal	DFSDgetcal	Get the calibration information
	SDgetdatastrs	sfgdtstr	DFSDgetdatastrs	Return the “data strings” for a data set
	SDgetdimstrs	sfsdmstr	DFSDgetdimstrs	Get the attribute strings for a dimension
	SDgetfillvalue	sfgfill	DFSDgetfill	Read the fill value if it exists
	SDgetrange	sfgrange	DFSDgetrange	Get the range of values for the data set
	SDreadattr	sfrattr	None	Read the values of an attribute
	SDsetattr	sfsattr	None	Define a new attribute for a variable
	SDsetcal	sfscal	DFSDsetcal	Set the calibration information
	SDsetdatastrs	sfsdtstr	DFSDsetdatastrs	Set the attribute strings for a data set
	SDsetdimstrs	sfgdmstr	DFSDsetdimstrs	Set the attribute strings for a dimension
	SDsetexternalfile	None	None	Store data in an external file
	SDsetfillvalue	sfsfill	DFSDsetfill	Set the fill value for this data set
	SDsetrange	sfsrange	DFSDsetrange	Set the max and min values

1.3.2 Scientific Data Sets: The NetCDF Interface

The SD interface is designed to be as interoperable as possible with netCDF, an interface developed by the Unidata Program Center. Consequently, the SD interface can read files written by the ***netCDF interface***, and the netCDF interface (as implemented in HDF) can read both netCDF files and HDF files that contain scientific data sets.

Further information regarding the netCDF/HDF interface can be found in Table 1.3.2 and in the *HDF User's Guide*. Additional information on the netCDF interface can be found in the *NetCDF User's Guide* available anonymous FTP from `unidata.ucar.edu`.

TABLE 1.3.2

Function Calls Available in the NC Interface

Purpose	Function Name		SD Equivalent	Description
	Long Name (C/Fortran)	Short Name (Fortran)		
Operations	`nccreate`	`NCCRE`	`SDstart`	Create a file
	`ncopen`	`NCOPN`	`SDstart`	Open a file
	`ncredef`	`NCREDF`	Not Applicable	Put open file into define mode
	`ncendef`	`NCENDF`	Not Applicable	Leave define mode
	`ncclose`	`NCCLOS`	`SDend`	Close an open file
	`ncinquire`	`NCINQ`	`SDfileinfo`	Inquire about an open file
	`ncsync`	`NCSNC`	Not Applicable	Synchronize a file to disk
	`ncabort`	`NCABOR`	Not Applicable	Back out of recent definitions
	`ncsetfill`	`NCSFIL`	Not Implemented	Set fill mode for writes
Dimensions	`ncdimdef`	`NCDDEF`	`SDsetdimname`	Create a dimension
	`ncdimid`	`NCDID`	`SDgetdimid`	Get a dimension ID from its name
	`ncdiminq`	`NCDINQ`	`SDdiminfo`	Inquire about a dimension
	`ncdimrename`	`NCDREN`	Not Implemented	Rename a dimension
Variables	`ncvardef`	`NCVDEF`	`SDcreate`	Create a variable
	`ncvarid`	`NCVID`	`SDnametoindex`	Get a variable ID from its name
	`ncvarinq`	`NCVINQ`	`SDgetinfo`	Get information about a variable
	`ncvarput1`	`NCVPT1`	Not Implemented	Write a single data value
	`ncvarget1`	`NCVGT1`	Not Implemented	Read a single data value
	`ncvarput`	`NCVPT`	`SDwritedata`	Write a hyperslab of values
	`ncvarget`	`NCVGT/NCVGTC`	`SDreaddata`	Read a hyperslab of values
	`ncvarrename`	`NCVREN`	Not Implemented	Rename a variable
	`nctypelen`	`NCTLEN`	`DFKNTsize`	Get number of bytes for a data type
Attributes	`ncattput`	`NCAPT/NCAPTC`	`SDsetattr`	Create an attribute
	`ncattinq`	`NCAINQ`	`SDattrinfo`	Get information about an attribute
	`ncattcopy`	`NCACPY`	Not Implemented	Copy attribute from one file to another
	`ncattget`	`NCAGT/NCAGTC`	`SDreadattr`	Get attributes values
	`ncattname`	`NCANAM`	`SDattrinfo`	Get name of attribute from its number
	`ncattrename`	`NCAREN`	Not Implemented	Rename an attribute
	`ncattdel`	`NCADEL`	Not Implemented	Delete an attribute

1.3.3 Vdata: The VS Interface

The ***Vdata interface*** provides a collection of routines for reading and writing customized tables. Each table is comprised of a series of ***Vdata records*** whose values are stored in fixed length fields. In addition to its records, a Vdata may contain three kinds of identifying information: a ***Vdata name***, ***Vdata class***, and a series of individual ***Vdata field names***.

Functions in the Vdata interface begins with the prefix VS. The equivalent FORTRAN functions use the prefix vsf. Vdata functions are divided into four categories:

- **Access routines** begin access to a Vdata, terminate access to a Vdata, and seek to any position in a Vdata set.
- **Specify routines** define new Vdata fields, assign a name and class, and initialize the access conditions for read and write permission**.**
- **Inquiry routines** check if a Vdata set exists, return information about particular Vdata set such as its class name or field names, its size, or whether it exists as a lone entity.
- **Read/Write routines** retrieve and store Vdata in HDF files.

The VS function calls are more explicitly defined in Table 1.3.3 and on their respective pages in the reference section of this manual. The VS interface is also supplemented by the VSQ, VF, and VH interfaces described in later sections. To learn more about the ***Vdata model*** or interface consult the *HDF User's Guide.*

TABLE 1.3.3 **Function Calls Available in the VS Interface.**

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Accessing the VS interface	VSattach	vsfatch	Attaches to the Vdata with the specified ID
	VSdetach	vsfdtch	Detaches from the current Vdata thereby terminating access
	VSseek	vsfseek	Reads a specified record from a Vdata
Specifying a Vdata	VSfdefine	vsffdef	Defines a new field for a Vdata
	VSsetclass	vsfscls	Assigns a class name to the Vdata
	VSsetfields	vsfsfld	Establishes permission to read or write to a field
	VSsetinterlace	vsfsint	Sets the interlace type for the Vdata
	VSsetname	vsfsnam	Assigns a name to the Vdata
Inquiring about Vdata	VSgetid	vsfgid	Returns the ID for the Vdata after the Vdata with the given ID
	VSgetname	vsfgnam	Returns the name (if any) of the specified Vdata
	VSgetfields	vsfgfld	Returns the field names (if any) for the Vdata
	VSinquire	vsfinq	This routine is the general Vdata inquiry routine
	VSlone	vsflone	Lists reference numbers for Vdata which do not belong to a Vgroup
	VSfexist	vsfex	Checks whether all the specified field(s) exists
	VSgetclass	vsfgcls	Returns the class name (if any) of the Vdata
	VSsizeof	vsfsiz	Computes the byte size of the field(s) for the local machine
	VSgetinterlace	vsfgint	Returns the interlace of the Vdata
	VSfind	vsffnd	Locates a Vdata set with a given name
Reading and writing Vdata	VSread	vsfread	Reads data from a Vdata
	VSwrite	vsfwrit	Writes data to a Vdata

1.3.4 Vdata Query: The VSQ Interface

The ***Vdata Query interface*** provides a collection of routines for inquiring about existing Vdata. These routines provide information such as the number of records in a Vdata, its field names, number types, and name. All functions in the Vdata Query interface begin with the prefix VSQ. There are no equivalent FORTRAN functions.

The VSQ function calls are more explicitly defined in Table 1.3.4 and on their respective pages in the reference section of this manual. To learn more about the ***Vdata model*** or the VSQ interface consult the *HDF User's Guide.*

TABLE 1.3.4 **Functions Available in the VSQ Interface**

Purpose	Function Name (C Only)	Description
Query for Vdata information	`VSQuerycount`	Returns the number of records in the Vdata
	`VSQueryfields`	Returns the names of the fields in the Vdata
	`VSQueryinterlace`	Returns the type of interlace in the Vdata
	`VSQueryname`	Returns the name of the Vdata
	`VSQueryref`	Retrieves the reference number for the Vdata
	`VSQuerytag`	Retrieves the tag for the specified Vdata
	`VSQueryvsize`	Retrieves the local size in bytes of a Vdata record

1.3.5 Vdata Fields: The VF Interface

The ***Vdata Fields interface*** provides a collection of routines for inquiring about the fields in an existing Vdata. These routines provide information such as the field name, size, order, and number type. All functions in the Vdata Field interface begins with the prefix VF. There are no equivalent FORTRAN functions.

The VF function calls are more explicitly defined in Table 1.3.5 and on their respective pages in the reference section of this manual. To learn more about the ***Vdata model*** or the VF interface consult the *HDF User's Guide.*

TABLE 1.3.5 **Functions Available in the VF Interface**

Purpose	Function Name (C Only)	Description
Query for Vdata field information	`VFfieldesize`	Retrieve the size (as stored in an HDF file) of a specified field in a Vdata
	`VFfieldisize`	Retrieve the size (as stored in memory) of a specified field in a give Vdata
	`VFfieldname`	Retrieves the fieldname for a specified field in a given Vdata
	`VFfieldorder`	Retrieves the order of a specified field in a given Vdata
	`VFfieldtype`	Retrieves the number type for a specified field in a given Vdata
	`VFnfields`	Retrieve the total number of fields in the specified Vdata

1.3.6 Vgroups: The V Interface

The ***Vgroup interface*** provides a collection of routines for reading and writing customized data sets. Each Vgroup may contain one or more Vdata sets, Vgroups, or data objects stored via other HDF data models. In addition to its members, a Vgroup may also be given a ***Vgroup name*** and a ***Vgroup class***.

Every function in the Vgroup interface begins with the prefix V. The equivalent FORTRAN functions use the prefix vf. The Vgroup functions are divided into four categories:

- **Access routines** open and close the V interface and begin and end access to Vgroups.
- **Specify routines** assign names, classes and members to Vgroups.
- **Inquiry routines** return the name and class of a Vgroup, determine the number, ID, and format for the members of a Vgroup.

The V function calls are more explicitly defined in Table 1.3.6 and on their respective pages in the reference section of this manual. To learn more about the ***Vgroup model*** or interface consult the *HDF User's Guide.*

TABLE 1.3.6 **Function Calls Available in the V Interface**

Purpose	Function Name		Description
	Long Name (C/Fortran)	Short Name (Fortran)	
Accessing the V interface	`Vattach`	`vfatch`	Attaches to a Vgroup when given a Vgroup ID
	`Vdetach`	`vfdtch`	Detaches the Vgroup thereby terminating access
	`Vend`	`vfend`	Opens an HDF file
	`Vstart`	`vfstart`	Closes an HDF file
Specifying a Vgroup	`Vsetclass`	`vfscls`	Assigns a class name to a Vgroup
	`Vsetname`	`vfsnam`	Assigns a name to a Vgroup
	`Vaddtagref`	`vfsdtr`	Adds an HDF object into a given Vgroup
	`Vinsert`	`vfinsrt`	Adds Vdata or Vgroups into other Vgroups
	`Vgetid`	`vfgid`	Returns the Vgroup following the Vgroup with the given ID
Inquiring about Vgroups	`Vgetclass`	`vfgcls`	Returns the class name (if any) of the Vgroup
	`Vgetname`	`vfgnam`	Returns the name (if any) of the Vgroup
	`Vinquire`	`vfinq`	This is the general Vgroup inquiry routine
	`Vlone`	`vflone`	Returns the ID for every Vgroup that is not part of another Vgroup
	`Vinqtagref`	`vfinqtr`	Tests whether an HDF object is a member of a Vgroup
	`Vntagrefs`		Returns a count of the number of HDF objects in a given Vgroup
	`Vgettagrefs`	`vfgttrs`	Returns the tag and reference number for the HDF objects in a given Vgroup
	`Vgettagref`	`vfgttr`	Returns tag and reference number for an object at a given position in a Vgroup

1.3.7 High Level VData/Vgroups: The VH Interface

The high level ***Vdata/Vgroup interface*** provides a collection of routines for creating simple Vdatas and Vgroups with a single function call. All functions in the High Level interface begins with the prefix ***VH***. There are no equivalent FORTRAN functions.

The VH function calls are more explicitly defined in Table 1.3.7 and on their respective pages in the reference section of this manual. To learn more about the Vdata or Vgroup model and the VH interface consult the *HDF User's Guide.*

TABLE 1.3.7 **Functions Available in the VH Interface**

Purpose	Function Name (C Only)	Description
Creating a Vdata or Vgroup	`VHmakegroup`	Groups a collection of tags and reference numbers into a Vgroup
	`VHstoredata`	Creates Vdatas containing single field records with one component per field
	`VHstoredatam`	Creates Vdatas containing single field records with multiple components per field

1.4 HDF Command Line Utilities

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

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

There are currently 16 different HDF utilities, which fall into the following categories:

- **Inquiry utilities** are primarily used to examine the contents of an HDF file.
- **Conversion utilities** include a variety of programs for converting between different HDF formats and raw data formats.
- **Compression utilities** are useful for compressing HDF files using standard compression algorithms and compacting techniques.
- **Vdata/Vgroup utilities** are useful for inquiring about files that contain Vdata and Vgroups.

The HDF utilities are more explicitly defined in Table 1.4 on page 16 and on their respective pages in the reference section of this manual. To learn more about the utilities consult *Getting Started with HDF* or the *HDF User's Guide.*

TABLE 1.4 **Utilities included in the HDF distribution.**

Purpose		Utility	Definition
Inquiry		`hdfls`	Displays the tags and reference numbers for the data objects contained in a file.
		`hdfed`	Invokes and HDF editor which allows you to browse through HDF files and manipulate the data objects.
		`hdfrseq`	Displays sequences of remote 8-bit images directly to the screen.
Conversion	**Raw to HDF**	`r8tohdf`	Converts raw 8-bit images to HDF raster image sets and writes them to a file.
		`paltohdf`	Converts raw palettes to HDF format and writes them to file.
		`fp2hdf`	Converts 32-bit floating-point data to scientific data or 8-bit raster image sets.
	HDF to Raw	`hdftor8`	Converts HDF raster image sets to raw format and separates them into image files and palette files.
		`hdftopal`	Converts palettes in HDF files to raw format and writes them to separate files.
	HDF to HDF	`fp2hdf`	Converts 32-bit floating-point data to scientific data or 8-bit raster image sets.
		`hdf24to8`	Converts 24-bit HDF images to 8-bit raster image sets.
		`hdf24to8`	Converts 8-bit raster images sets into 24-bit raster image sets.
		`ristosds`	Converts several 8-bit raster images to one 3D scientific data set.
	HDF to JFIF	`hdf2jpeg`	Copies JPEG compressed images from an HDF file into separate JFIF files
	JFIF to HDF	`jpeg2hdf`	Copies a JPEG compressed image from a JFIF file into an HDF file.
Compression		`hdfpack`	Condenses HDF files by removing the empty spaces left by deleted objects.
		`hdfcomp`	Compresses 8-bit raster image in an HDF file using run-length encoding or IMCOMP compression.
Vdata/Vgroups		`vcompat`	Converts HDFv1.0 Vdata or Vgroups to HDFv2.0 or later
		`vmake`	Creates Vgroups by linking together Vdata, Vgroups, or both
		`vshow`	Dumps Vdata and Vgroup information contained in an HDF file.

Chapter 2

HDF Function Call Reference Section

2.0 Overview of the Function Calls Reference Section

This section of the *HDF Reference Manual* contains a listing of every function call contained in the HDF v3.3 library. The pages are organized alphabetically according to the C long name for each routine. Each function call begins on a new page and takes the following form:

Function_Name

`return_type function_name(type` *`parameter1`*`, type` *`parameter2`*`)`

`parameter1`	Definition of first parameter in the synopsis
`parameter2`	Definition of second parameter in the synopsis

Purpose	Section containing a short description of the function
Return value	Section describing the return value (if any) and error conditions for the function.
Description	Section detailing the specific use of the named function. It defines any special circumstances surrounding the use of the function, identifies any calls that must precede the named function, and describes any known complications affecting the use of the function.
Example	This optional section consists of one or more examples of the use of the function. The examples are often just fragments of code (not complete programs) for illustrative purposes only.
FORTRAN	This section provides the synopsis for the equivalent FORTRAN function.

DFANaddfds

```
intn DFANaddfds(int32 file_id, char *description, int32 desc_len)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
description	IN:	Sequence of ASCII characters (may include NULL '\0')
desc_len	IN:	Length of the description

Purpose — Adds a file description to a file.

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

Description — These annotations are associated with the file, not with any particular object within the file. The parameter *description* can contain any sequence of ASCII characters. It does not have to be a string. Use the general purpose routines **Hopen** or **HDFopen** and **Hclose** or **HDFclose** to manage file access because the file annotation routines will not open and close HDF files for you.

Example — This example illustrates the use of **DFANaddfid** and **DFANaddfds** to write to an HDF file a label and description for the file itself.

```
#include "hdf.h"
#define MAXLABLEN 80
#define MAXDESCLEN 1000
...
int32 file_id;
char label[MAXLABLEN+1], description[MAXDESCLEN+1];

file_id = Hopen("myfile", DFACC_WRITE, 0);

/* store a file label in the file */
strcpy (label, "File #1");
DFANaddfid(file_id, label);

/* store description in file */
DFANaddfds(file_id, description, desc_len(description));

Hclose(file_id);
...
```

FORTRAN

```
integer function daafds(file_id, description, desc_len)
   integer*4      file_id
   character*(*)  description
   integer*4      desc_len
```

DFANaddfid

daafid

DFANaddfid

```
intn DFANaddfid(int32 file_id, char *label)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
label	IN:	A single string of null-terminated text

Purpose
Writes a file label to a file.

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

Description
These annotations are associated with the file, not with any particular object within the file. The label must be a single string. Use the general purpose routines **Hopen** or **HDFopen** and **Hclose** or **HDFclose** to manage file access because the file annotation routines will not open and close HDF files for you.

In the FORTRAN version, the string length for the label should be close to the actual expected string length, because in FORTRAN string lengths generally are assumed to be the declared length of the array that holds the string.

FORTRAN

```
integer function daafid(file_id, label)
   integer*4      file_id
   character*(*)  label
```

dagdesc

DFANgetdesc

```
intn DFANgetdesc(char *filename, uint16 tag, uint16 ref, char *desc_buff,
          int32 buff_len)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object assigned the description
ref	IN:	Reference number of the data object assigned the description
desc_buff	OUT:	The buffer allocated to hold the description
buff_len	IN:	Size of the buffer allocated to hold the description

Purpose Reads the description assigned to the data object with the given tag and reference number.

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

Description The parameter *buff_len* gives the amount of space that is available for storing the description. The length of *buff_len* must be at least one greater than the anticipated length of the description, because a `'\0'` byte is appended to the annotation.

FORTRAN

```
integer function dagdesc(filename, tag, ref, desc_buff, buff_len)
    character*(*)  filename, desc_buff
    integer*2      tag, ref
    integer*4      buff_len
```

DFANgetdesclen

dagdlen

DFANgetdesclen

```
int32 DFANgetdesclen(char *filename, uint16 tag, uint16 ref)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object assigned the description
ref	IN:	Reference number of the data object assigned the description

Purpose — To get the length of a description of the data object with the given tag and reference number.

Return value — Returns the length of a description if successful and `FAIL(-1)` otherwise.

Description — This routine should be used to insure that there is enough space allocated for a description before actually reading it.

FORTRAN

```
integer function dagdlen(filename, tag, ref)
   character*(*)  filename
   integer*2      tag, ref
```

dagfds

DFANgetfds

```
int32 DFANgetfds(int32 file_id, char *desc_buff, int32 buff_len, intn isfirst)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
desc_buff	OUT:	The buffer allocated to hold the description
buff_len	IN:	Size of the buffer allocated to hold the description
isfirst	IN:	1 to read the first description; 0 to read the next description

Purpose — Reads the next file description from a file.

Return value — Returns the length of the file description if successful and FAIL(-1) otherwise.

Description — If *isfirst* is 0, **DFANgetfds** gets the next file description from an HDF file. For example, if there are three file descriptions in a file, three successive calls to **DFANgetfds** will get all three descriptions. If *isfirst* is 1, **DFANgetfds** gets the first file description.

FORTRAN

```
integer function dagfds(file_id, desc_buff, buff_len, isfirst)
    integer*4       file_id
    character*(*)   desc_buff
    integer*4       buff_len
    integer         isfirst
```

DFANgetfdslen

dagfdsl

DFANgetfdslen

```
int32 DFANgetfdslen(int32 file_id, intn isfirst)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
isfirst	IN:	`1` to read the first description; `0` to read the next description

Purpose — Returns the length of a file description.

Return value — Returns the length of the file description if successful and `FAIL(-1)` otherwise.

Description — When **DFANgetfdslen** is first called for a given file, it returns the length of the first file description. In order to get the lengths of successive file descriptions, you must call **DFANgetfds** between calls to **DFANgetfdslen**. Successive calls to **DFANgetfdslen** without calling **DFANgetfds** between them will return the length of the same file description.

FORTRAN

```
integer function dagfdsl(file_id, isfirst)
   integer*4      file_id
   integer        isfirst
```

DFANgetfid

```
int32 DFANgetfid(int32 file_id, char *desc_buff, int32 buff_len, intn isfirst)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
label_buff	OUT:	The buffer allocated to hold the label
buff_len	IN:	Size of the buffer allocated to hold the label
isfirst	IN:	1 to read the first label; 0 to read the next label

Purpose — Reads a file label from a file.

Return value — Returns the length of the file description if successful and FAIL(-1) otherwise.

Description — If *isfirst* is 0, **DFANgetfid** gets the next file label from the file. If *isfirst* is 1, **DFANgetfid** gets the first file label in the file. If the *buff_len* is not large enough, the label is truncated to *buff_len*-1 characters in the buffer label_buff.

FORTRAN

```
integer function dagfid(file_id, desc_buff, buff_len, isfirst)
    integer*4       file_id
    character*(*)   desc_buff
    integer*4       buff_len
    integer         isfirst
```

DFANgetfidlen

dagfidl

DFANgetfidlen

```
int32 DFANgetfidlen(int32 file_id, intn isfirst)
```

file_id	IN:	The HDF file identifier returned by **Hopen** or **HDFopen**
isfirst	IN:	1 to read the first label; 0 to read the next label

Purpose — Returns the length of a file label.

Return value — Returns the length of the file label if successful and FAIL(-1) otherwise.

Description — When **DFANgetfidlen** is first called for a given file, it returns the length of the first file label. In order to get the lengths of successive file labels, you must call **DFANgetfid** between calls to **DFANgetfidlen**. Otherwise, successive calls to **DFANgetfidlen** will return the length of the same file label.

FORTRAN

```
integer function dagfidl(file_id, isfirst)
    integer*4       file_id
    integer         isfirst
```

DFANgetlabel

```
intn DFANgetlabel(char *filename, uint16 tag, uint16 ref, char *label_buff,
          int32 buff_len)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object assigned the label
ref	IN:	Reference number of the data object assigned the label
label_buff	OUT:	The buffer allocated to hold the label
buff_len	IN:	Size of the buffer allocated to hold the label

Purpose Reads the label assigned to the data object with the given tag and reference number.

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

Description The parameter *buff_len* gives the amount of space that is available for storing the label. The length of *buff_len* must be at least one greater than the anticipated length of the label, because a `'\0'` byte is appended to the annotation.

FORTRAN

```
integer function daglab(filename, tag, ref, label_buff, buff_len)
    character*(*)  filename, label_buff
    integer*2      tag, ref
    integer*4      buff_len
```

DFANgetlablen

dagllen

DFANgetlablen

```
int32 DFANgetlablen(char *filename, uint16 tag, uint16 ref)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object assigned the label
ref	IN:	Reference number the data object assigned the label

Purpose — Returns the length of a label assigned to the object with a given tag and reference number.

Return value — Returns the length of the label if successful and `FAIL(-1)` otherwise.

Description — This routine should be used to insure that there is enough space allocated for a label before actually reading it.

FORTRAN

```
integer function dagllen(filename, tag, ref)
    character*(*)  filename
    integer*2      tag, ref
```

DFANlablist

```
int DFANlablist(char *filename, uint16 tag, unit16 ref_list[], char *label_list,
          int list_len, intn label_len, intn start_pos)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag to use when searching for reference numbers and labels
ref_list	OUT:	The buffer allocated to hold the reference numbers
label_list	OUT:	The buffer (array of strings) allocated to hold the labels
list_len	IN:	Size of reference number list and the label list
label_len	IN:	Maximum length allowed for a label
start_pos	IN:	Entries are returned from the *start_pos* entry up to the *list_len* entry

Purpose Returns a list of all reference numbers and labels (if labels exist) for a given tag.

Return value Returns the number of reference numbers found if successful and FAIL(-1) otherwise.

Description The *list_len* determines the number of available entries in the reference number and label lists, *label_len* is the maximum length allowed for a label, and *start_pos* tells which label to start reading for the given tag. (If *start_pos* is 1, for instance, all labels will be read; if *start_pos* is 4, all but the first 3 labels will be read.) The *ref_list* contains a list of reference numbers for all objects with a given tag. The *label_list* contains a corresponding list of labels, where they exist. If there is no label stored for a given object, the corresponding entry in *label_list* is an empty string.

Taken together, the *ref_list* and *label_list* constitute a directory of all objects and their labels (where they exist) for a given tag. The list, *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. Hence, this routine provides a mechanism for direct access to data objects in HDF files.

FORTRAN

```
integer function dallist(filename, tag, ref_list, label_list,
list_len, label_len, start_pos)
    character*(*)  filename, label_list
    integer*2      tag, ref_list(*)
    integer        list_len, label_len, start_pos
```

dalref

DFANlastref

```
uint16 DFANlastref(void)
```

Purpose	Returns the reference number for the annotation last written or read.
Return value	Returns the reference number if successful and `FAIL(-1)` otherwise.
Description	**`DFANlastref`** returns the reference number for the last annotation read or written.
FORTRAN	`integer function dalref()`

DFANputdesc

```
int DFANputdesc(char *filename, uint16 tag, uint16 ref, char *description,
          int32 desc_len)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object to be assigned the description
ref	IN:	Reference number the data object to be assigned the description
description	IN:	Sequence of ASCII characters (may include NULL '\0')
desc_len	IN:	Length of the description

Purpose Writes a description for the data object with the given tag and reference number.

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

Description The parameter *description* can contain any sequence of ASCII characters; it does not have to be a string.

FORTRAN

```
integer function dapdesc(filename, tag, ref, description, desc_len)
    character*(*)  filename, description
    integer*2      tag, ref
    integer*4      desc_len
```

DFANputlabel

daplab

DFANputlabel

```
intn DFANputlabel(char *filename, uint16 tag, uint16 ref, char *label)
```

filename	IN:	Name of the HDF file
tag	IN:	Tag of the data object to be assigned the label
ref	IN:	Reference number the data object to be assigned the label
label	IN:	A single string of null-terminated text

Purpose — Assigns a label to the data object with the given tag and reference number.

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

FORTRAN

```
integer function daplab(filename, tag, ref, label)
   character*(*)  filename, label
   integer*4      tag, ref
```

DFPaddpal

```
intn DFPaddpal(char *filename, VOIDP palette)
```

filename	IN:	Name of HDF file
palette	IN:	768-byte space with palette

Purpose Appends a palette to a file.

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

Description If the named file does not exist, it is created and the palette written to it.

FORTRAN

```
integer function dpapal(filename, palette)
   character*(*)  filename
   <valid type>   palette(*)
```

DFPgetpal

dpgpal

DFPgetpal

```
intn DFPgetpal(char *filename, VOIDP palette)
```

filename	IN:	Name of HDF file
palette	OUT:	768-byte space for palette

Purpose — Gets the next palette from file and store it in the array palette.

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

Description — The array *palette* is assumed to be allocated at least 768 bytes. Successive additional calls to **DFPgetpal** retrieve the palettes in the file in the sequence in which they are stored.

FORTRAN

```
integer function dpgpal(filename, palette)
   character*(*)  filename
   <valid type>   palette(*)
```

DFPlastref

```
intn DFPlastref()
```

Purpose	Determines the value of the reference number most recently read or written by a palette function call.
Return value	Returns the reference number if successful and `FAIL(-1)` otherwise.
Description	Returns the value of the reference number most recently read or written
FORTRAN	`integer function dplref()`

dpnpals

DFPnpals

```
intn DFPnpals(char *filename)
```

filename	IN:	Name of HDF file

Purpose	Tells how many palettes are present in a file.
Return value	Returns the number of palettes if successful and `FAIL(-1)` otherwise.
Description	Indicates number of palettes in a file.
FORTRAN	`integer function dpnpals(`*filename*`)` `  character*(*)  `*filename*

DFPputpal

```
intn DFPputpal(char *filename, VOIDP palette, intn overwrite, char *filemode)
```

filename	IN:	Name of HDF file
palette	IN:	768—byte space for palette
overwrite	IN:	Use "1" to overwrite last palette; use "0" to write a new palette
filemode	IN:	If "a", append palette to file; if "w", create new file

Purpose — Writes a palette to file.

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

Description — This routine provides more control than **DFPaddpal**. Note however, that the combination *filemode*="w" and *overwrite*=1 has no meaning and will generate an error. To overwrite a palette, filename must be the same filename as the last file accessed through the DFP interface.

FORTRAN

```
integer function dpppal(filename, palette, overwrite,
filemode)
    character*(*)  filename
    <valid type>   palette(*)
    integer        overwrite
    character*(*)  filemode
```

DFPreadref

dprref

DFPreadref

```
intn DFPreadref(char *filename, uint16 ref)
```

filename	IN:	Name of HDF file
ref	IN:	Reference number to be used in next **DFPgetpal** call

Purpose: Sets the reference number of the palette that **DFPgetpal** will retrieve next.

Return value: Returns SUCCEED(0) if palette with specified reference number exists and FAIL(-1) otherwise.

Description: Sets the reference number for the next palette to be retrieved.

FORTRAN

```
integer function dprref(filename, ref)
   character*(*)  filename
   integer*2      ref
```

dprest

DFPrestart

```
intn DFPrestart()
```

Purpose	Sets up the next call to **DFPgetpal** to read the first palette in the file.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	Specifies that the next call to **DFPgetpal** reads the first palette in the file, rather than the next unread palette.
FORTRAN	`integer function dprest()`

DFPwriteref

dpwref

DFPwriteref

```
intn DFPwriteref(char *filename, uint16 ref)
```

filename	IN:	Name of HDF file
ref	IN:	Reference number to be assigned to the next palette written to a file

Purpose Sets the reference number of the next palette to be written.

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

Description Sets the reference number for the next palette to be written. The `filename` is ignored. The next palette written, regardless of the `filename`, is assigned the reference number `ref`.

FORTRAN

```
integer function dpwref(filename, ref)
   character*(*)  filename
   integer*2      ref
```

DFR8addimage

```
intn DFR8addimage(char *filename, VOIDP image, int32 width, int32 height,
          uint16 compress)
```

filename	IN:	Name of HDF file
image	IN:	Array holding image to add to file
width	IN:	Number of columns in the image
height	IN:	Number of rows in the image
compress	IN:	Type of compression to use, if any

Purpose **DFR8addimage** appends the RIS8 for the image to the file.

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

Description To append to the file the RIS8 for the image. In all other respects, **DFR8addimage** is functionally equivalent to **DFR8putimage**.

FORTRAN

```
integer function d8aimg(filename, image, width, height,
compress)
    character*(*)   filename
    <valid type>    image(*)
    integer*4       width, height
    integer*2       compress
```

DFR8getdims

d8gdims

DFR8getdims

```
intn DFR8getdims(char *filename, int32 *width, int32 *height, intn *ispalette)
```

filename	IN:	Name of HDF file containing the 8-bit raster image(s)
width	OUT:	Number of columns in the next image in the file
height	OUT:	Number of rows in the next image in the file
ispalette	OUT:	Returns "1" if there is a palette, otherwise returns "0"

Purpose

To open the file, find the next image, retrieve the dimensions of the image, and determine whether there is a palette associated with the image.

Return value

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

Description

`DFR8getdims` retrieves the dimensions of the image and indicates whether a palette is associated and stored 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. Thus, images are read in the same order in which they were written to the file.

Normally, **`DFR8getdims`** is called before **`DFR8getimage`** so that if necessary, space allocations for the image and palette can be checked, and the dimensions can be verified. If this information is already known, **`DFR8getdims`** need not be called.

FORTRAN

```
integer function d8gdims(filename, width, height, ispalette)
    character*(*)  filename
    integer*4      width, height
    integer        ispalette
```

DFR8getimage

```
intn DFR8getimage(char *filename, uint8 *image, int32 width, int32 height,
          uint8 *palette)
```

filename	IN:	Name of file with RIS8 image
image	OUT:	Array that will hold image
width	IN:	Width of buffer allocated to store image
height	IN:	Height of buffer allocated to store image
palette	OUT:	Palette to go with image

Purpose — To retrieve the image and its palette, if it is present, and store them in the specified arrays.

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

Description — If palette is `NULL`, no palette is loaded, even if there is one stored with the image. If the image in the file is compressed, **`DFR8getimage`** automatically decompresses it. If **`DFR8getdims`** has not been called, **`DFR8getimage`** finds the next image in the same way that **`DFR8getdims`** does.

The variables `width` and `height` give the number of columns and rows, respectively, in the array which you've allocated in memory to store the image. The image may actually be smaller than the allocated space.

The order in which you declare dimensions is different between C and FORTRAN. Ordering varies because FORTRAN arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the horizontal coordinate varies fastest). When **`d8gimg`** reads an image from a file, it assumes row-major order. The FORTRAN declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension. To take this into account as you read image in your program, you need to read in the image “on its side.”

FORTRAN

```
integer function d8gimg(filename, image, width, height,
palette)
    character*(*)  filename
    character*1    image(*), palette(*)
    integer*4      width, height
```

DFR8lastref

d8lref

DFR8lastref

```
uint16 DFR8lastref()
```

Purpose	To get last reference number written or read for an RIS8.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	This routine is primarily used for attaching annotations to images and adding images to Vgroups. **`DFR8lastref`** returns the reference number of last raster image set read or written.
FORTRAN	`integer function d8lref()`

DFR8nimages

```
intn DFR8nimages(char *filename)
```

filename	IN:	Name of HDF file

Purpose — To count the number of 8-bit raster images contained in an HDF file.

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

Description — **`DFR8nimages`** counts the number of 8-bit images stored in the file.

FORTRAN

```
integer function d8nims(filename)
    character*(*)  filename
```

DFR8putimage

d8pimg

DFR8putimage

```
intn DFR8putimage(char *filename, VOIDP image, int32 width, int32 height,
          uint16 compress)
```

filename	IN:	Name of file to store the raster image in
image	IN:	Array with image to put in file
width	IN:	Number of columns in the image
height	IN:	Number of rows in the image
compress	IN:	Type of compression to use: 0=none, DFTAG_RLE, or DFTAG_IMCOMP

Purpose

To write out the RIS8 for the image as the first image in the file, overwriting any information that was previously in the file.

Return value

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

Description

DFR8putimage writes out the raster image set for the image as the first image in the file. The argument compress identifies the scheme to be used for compressing the data, if any. If IMCOMP compression is used, the image must include a palette.

DFR8putimage overwrites any information that exists in the HDF file. To write an image to a file by appending it, rather than overwriting it, use **DFR8addimage**.

In FORTRAN, the dimensions of the array image must be the same as the dimensions of the image itself.

The order in which you declare dimensions is different between C and FORTRAN. Ordering varies because FORTRAN arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the horizontal coordinate varies fastest). When **DFR8putimage** writes an image to a file, it assumes row-major order. The FORTRAN declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension, the reverse of the way it is done in C. To take this into account as you build your image in your FORTRAN program, you need to build the image "on its side."

FORTRAN

```
integer function d8pimg(filename, image, width, height,
compress)
    character*(*)  filename
    <valid type>   image(*)
    integer*4      width, height
    integer*2      compress
```

DFR8readref

```
intn DFR8readref(char *filename, uint16 ref)
```

filename	IN:	Name of HDF file containing image
ref	IN:	Reference number for next **DFR8getimage**

Purpose

To specify the reference number of the image to be read when **DFR8getimage** is next called.

Return value

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

Description

DFR8readref sets the reference number of the image to get next. This routine is most likely to be used in conjunction with **DFANgetlablist**, which returns a list of labels for a given tag together with their reference numbers. It provides, in a sense, a random access to images.

There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the sequence number for an image.

FORTRAN

```
integer function d8rref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

DFR8restart

d8first

DFR8restart

```
intn DFR8restart()
```

Purpose	To cause the next get to read from the first RIS8 in the file, rather than the RIS8 following the one that was most recently read.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	**`DFR8restart`** causes the next get command to read from the first raster image set in the file
FORTRAN	`integer function d8first()`

DFR8setcompress

```
intn DFR8setcompress(int32 type, comp_info *cinfo)
```

type	IN:	Type of compression
cinfo	IN:	Pointer to compression information structure

Purpose Set the type of compression to use when writing the next 8-bit raster image.

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

Description This routines provides a method for compressing the next raster image written. The *type* can be one of the following values: COMP_NONE, COMP_JPEG, COMP_RLE, COMP_IMCOMP. COMP_NONE is the default for storing images if this routine is not called, therefore images are not compressed by default. COMP_JPEG compresses images with a JPEG algorithm, which is a lossy method. COMP_RLE uses lossless run-length encoding to store the image. COMP_IMCOMP uses a lossy compression algorithm called IMCOMP, and is included for backward compatibility only.

The *cinfo* parameter is a pointer to a union defined as:

```
union {
    struct {
        intn quality;
        intn force_baseline;
    } jpeg;
} comp_info;
```

This union is defined to provide future expansion, but is currently only used by the COMP_JPEG compression type. A pointer to a valid comp_info union is required for all compression types other than COMP_JPEG, but the values in the union are not used. The comp_info union is declared in the header file hdf.h and is shown here for informative purposes only, it should not be re-declared in a user program.

For COMP_JPEG compression, the *quality* member of the *jpeg* structure must be set to the quality of the stored image. This number can vary from 100, the best quality, to 0, terrible quality. All images stored with COMP_JPEG compression are stored in a lossy manner, even images stored with a quality of 100. The ratio of size to perceived image quality varies from image to image, some experimentation may be required to determine an acceptable *quality* factor for a given application. The *force_baseline* parameter determines whether the quantization tables used during compression are forced to the range 0-255. The *force_baseline* parameter should normally be set to 1 (forcing baseline results), unless special applications require non-baseline images to be used.

FORTRAN None

DFR8setpalette

d8spal

DFR8setpalette

```
intn DFR8setpalette(uint8 *palette)
```

palette	IN:	Palette to go with the image

Purpose	Indicate which palette, if any, is to be used for subsequent image sets.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	Sets the default palette to be used for subsequent images. The specified palette continues as the default palette until it is changed by a new call to **`DFR8setpalette`**.
FORTRAN	`integer function d8spal(`*`palette`*`)` `  character*1 `*`palette`*`(*)`

DFR8writeref

```
intn DFR8writeref(char *filename, uint16 ref)
```

filename	IN:	HDF file containing image
ref	IN:	Reference number for next **DFR8putimage** or **DFR8addimage**

Purpose — To specify the reference number of the image to be written when **DFR8addimage** or **DFR8putimage** is next called.

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

Description — **DFR8writeref** assigns the specified reference number to the next image written to file. It is unlikely that you will need this routine, but if you do, use it with caution. There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the sequence number for an image. In addition, using an existing reference number will overwrite the existing 8-bit raster image data.

FORTRAN

```
integer function d8wref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

DFSDadddata

```
intn DFSDadddata(char *filename, intn rank, int32 dimsizes[], VOIDP data)
```

filename	IN:	Name of HDF file to store the data set
rank	IN:	Number of dimensions in the data array to be written
dimsizes	IN:	Array holding the size of each dimension
data	IN:	Array holding the data to be stored

Purpose Appends a scientific data set in its entirety to an existing HDF file if the file exists; if not, a new file is created.

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

Description In addition to appending a multidimensional array of data to an HDF file, **`DFSDadddata`** automatically stores any information pertinent to the data set. It will not overwrite existing data in the file. The array *`data`* can be of any valid type. However, if no number type has been set by **`DFSDsetNT`**, it is assumed that the data is of type `float32`.

Calling **`DFSDadddata`** will write the scientific data set and all associated information. That is, when **`DFSDadddata`** is called, any information set by a **`DFSDset*`** call is written to the file, along with the data array itself.

Example This example stores a 5x20x5000 array of type `float32` in a scientific data set. The scientific data set is added to a file called `'myfile.hdf'`, with no other attribute information.

```
#include "hdf.h"
float32 points[5][20][5000];
int dims[3];
...

dims[0] = 5;
dims[1] = 20;
dims[2] = 5000;

DFSDadddata("myfile.hdf",3, dims, points);
```

FORTRAN

```
integer function dsadata(filename, rank, dimsizes, data)
   character*(*)  filename
   integer        rank
   integer*4      dimsizes(*)
   <valid type>   data(*)
```

DFSDclear

dsclear

DFSDclear

```
intn DFSDclear()
```

Purpose	Clears all possible set values.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	After a call to **`DFSDclear`**, values set by any **`DFSDset*`** call will not be written unless they have been set again.
FORTRAN	`integer function dsclear()`

dseslab

DFSDendslab

```
intn DFSDendslab()
```

Purpose	Terminates a slab operation.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	Ends a sequence of slab calls started by **`DFSDstartslab`** by closing the file.
FORTRAN	`integer function dseslab()`

DFSDendslice

dseslc

DFSDendslice

```
intn DFSDendslice()
```

Purpose	Terminates the write operation after storing a slice of data in a scientific data set.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	**`DFSDendslice`** must be called after all the slices are written. It checks to ensure that the entire data set has been written, and if it has not, returns an error code. **`DFSDendslice`** is obsolete in favor of **`DFSDendslab`**. **`DFSDendslab`** is the recommended function call to use when terminating hyperslab (previously known as data slices) operations. HDFv3.3 will continue to support **`DFSDendslice`** only to maintain backward compatibility with HDF applications built on earlier versions of the library.
FORTRAN	`integer function dseslc()`

DFSDgetcal

```
int32 DFSDgetcal(float64 *cal, float64 *cal_err, float64 *offset,float64 *offset_err,
          int32 *num_type)
```

cal	OUT:	Calibration factor
cal_err	OUT:	Calibration error
offset	OUT:	Uncalibrated offset
offset_err	OUT:	Uncalibrated offset error
num_type	OUT:	Number type of uncalibrated data

Purpose Gets calibration information associated with the data.

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

Description This routine reads the calibration record, if there is one, attached to a scientific data set. A calibration record contains four 64-bit floating point values followed by a 32-bit integer, to be interpreted as follows:

`cal`	calibration factor
`cal_err`	calibration error
`offset`	uncalibrated offset
`offset_err`	uncalibrated offset error
`num_type`	number type of uncalibrated data

The relationship between a value ‘iy’ stored in a data set and the actual value ‘y’ is defined as:

$$y = cal * (iy - offset)$$

The variable *offset_err* contains a potential error of offset, and *cal_err* contains a potential error of *cal*. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

Example Suppose the values in the calibrated data set `iy[]` are the following integers:

```
iy[6] = {2, 4, 5, 11, 26, 81}
```

By defining `cal=0.50` and `offset=-200.0` and applying the calibration formula, the calibrated data set `iy[]` returns to its original form as a floating point array:

```
y[6] = {1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5}
```

FORTRAN

```
integer function dsgcal(cal, cal_err, offset, offset_err, num_type)
   real*8          cal, cal_err, offset, offset_err
   integer*4       num_type
```

DFSDgetdata

dsgdata

DFSDgetdata

```
intn DFSDgetdata(char *filename, intn rank, int32 dimsizes[], VOIDP data)
```

filename	IN:	Name of file containing scientific data set
rank	IN:	Number of dimensions
dimsizes	IN:	Array that holds dims of buffer that will hold the data
data	OUT:	Array for holding the data

Purpose

Reads the next data set in the file.

Return value

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

Description

If you do not know the values of *rank* or *dimsizes*, you must call **DFSDgetdims** to get them and then use them to provide the right amount of space for the array *data*. If you do not know the number type of the data in a scientific data set, you can call **DFSDgetNT** to find it out.

Subsequent calls to **DFSDgetdata** (or to **DFSDgetdims** and **DFSDgetdata**) will sequentially read scientific data sets from the file. For example, if **DFSDgetdata** is called three times in succession, the third call reads data from the third scientific data set in the file. Of course, if you do not know the values of *rank* or *dimsizes*, you must call **DFSDgetdims** to get them and then use them to provide the right amount of space for the array data.

If **DFSDgetdims** or **DFSDgetdata** is called and there are no more scientific data sets left in the file, an error code is returned and nothing is read. **DFSDrestart** can be used to override this convention.

Example

The following code reads an array whose dimensions are known to be 100x200, and whose number type is known to be `int16` (16-bit integer).

```
#include "hdf.h"
unit16 density[100][200];
int32 sizes[2], ret;

sizes[0] = 100;
sizes[1] = 200;
ret = DFSDgetdata("myfile.hdf", 2, sizes, density);
```

FORTRAN

```
integer function dsgdata(filename, rank, dimsizes, data)
   character*(*)  filename
   integer        rank
   integer*4      dimsizes(*)
   <valid type>   data(*)
```

DFSDgetdatalen

```
intn DFSDgetdatalen(intn *label_len, intn *unit_len, intn *format_len,
          intn *coords_len)
```

label_len	OUT:	Maximum length for the label string
unit_len	OUT:	Maximum length for the unit string
format_len	OUT:	Maximum length for the format string
coords_len	OUT:	Maximum length for the coordinate system string

Purpose Gets actual length of the label, unit, format, and coordinate system strings.

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

Description The space allocated for the label, unit, format, and coordinate system strings must be at least 1 byte larger than the actual length of the string.

FORTRAN

```
integer function dsgdaln(label_len, unit_len, format_len,
coords_len)
    integer label_len, unit_len, format_len, coords_len
```

DFSDgetdatastrs

dsgdast

DFSDgetdatastrs

```
intn DFSDgetdatastrs(char *label, char *unit, char *format, char *coordsys)
```

label	OUT:	Label that describes the data
unit	OUT:	Unit to be used with the data
format	OUT:	Format to be used in displaying data
coordsys	OUT:	Coordinate system

Purpose
Retrieves information about the data from its associated attribute strings.

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

Description
`DFSDgetdatastrs` reads the label, unit, and format strings for the data. The parameter *`coordsys`* gives the coordinate system that is to be used for interpreting the dimension information.

FORTRAN

```
integer function dsgdast(label, unit, format, coordsys)
    character*(*) label, unit, format, coordsys
```

DFSDgetdimlen

```
intn DFSDgetdimlen(intn dim, intn *label_len, intn *unit_len, intn *format_len)
```

dim	OUT:	Dimension the label, unit, and format refer to
label_len	OUT:	Length of the label
unit_len	OUT:	Length of the unit
format_len	OUT:	Length of the format

Purpose — Gets the length of the label, unit, and format for the specified dimension.

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

Description — The space allocated to hold the label, unit, and format strings must be at least 1 byte larger than the actual length of the string.

FORTRAN

```
integer function dsgdiln(dim, label_len, unit_len, format_len)
    integer dim, label_len, unit_len, format_len
```

DFSDgetdims

dsgdims

DFSDgetdims

```
intn DFSDgetdims(char *filename, intn *rank, int32 dimsizes[], intn maxrank)
```

filename	IN:	Name of file with scientific data set
rank	OUT:	Number of dimensions
dimsizes	OUT:	Array for holding dimensions of the data set in the file
maxrank	IN:	Size of array dimsizes

Purpose

Gets the number of dimensions and the size of each dimension for the next data set in the file.

Return value

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

Description

`DFSDgetdims` gets the number of dimensions (*rank*) of the data set and the sizes of the dimensions (*dimsizes*) for the next scientific data set in the file. The input argument *maxrank* tells **`DFSDgetdims`** the size of the array that is allocated for storing the *dimsizes* array. The value of *rank* cannot exceed the value of *maxrank*.

The allocation of space for reading in the scientific data set should correspond to the values read in by **`DFSDgetdims`**. The first value in the array *dimsizes* should equal the first dimension of the array that is allocated to hold the data set; the second value in *dimsizes* should equal the second dimension of the data set, and so forth.

FORTRAN

```
integer function dsgdims(filename, rank, dimsizes, maxrank)
    character*(*)  filename
    integer        rank, maxrank
    integer*4      dimsizes(*)
```

DFSDgetdimscale

```
intn DFSDgetdimscale(intn dim, int32 size, VOIDP scale)
```

dim	IN:	Dimension this scale corresponds to
size	IN:	Size of the scale array
scale	OUT:	Array of values defining reference points along a specified dimension

Purpose Gets the scale corresponding to the specified dimension.

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

Description The DFSD interface supported in HDFv3.3, requires that the dimension scales are of the same number type as the corresponding data. To store dimension scales of a different number type than the corresponding data, see the multi-file SD interface.

Example In this example an 800 x 500 data array is read from `'myfile.hdf'`, together with a scale for each dimension. The scales are assumed to be of type `float32`.

```
intn rank;
int32 dimsizes[2];
float32 yscale[800], xscale[500];
float32 pressure[800][500];

DFSDgetdims ("SDex2.hdf", rank, dimsizes, 2);
DFSDgetdata ("SDex2.hdf", rank, dimsizes, pressure);
DFSDgetdimscale (1, dimsizes[0], yscale);
DFSDgetdimscale (2, dimsizes[1], xscale);
```

FORTRAN

```
integer function dsgdisc(dim, size, scale)
   integer        dim
   integer*4      size
   <valid type>   scale(*)
```

DFSDgetdimstrs

dsgdist

DFSDgetdimstrs

```
intn DFSDgetdimstrs(intn dim, char *label, char *unit, char *format)
```

dim	IN:	Dimension this label, unit and format refer to
label	OUT:	Label that describes this dimension
unit	OUT:	Unit to be used with this dimension
format	OUT:	Format to be used in displaying scale for this dimension

Purpose Gets the label, unit, and format strings corresponding to the specified dimension.

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

Description The space allocated for the label, unit, and format string must be at least 1 byte larger than the length of the string. If the length is unknown when the program is written, declare the array size as `1+maxlen_label`, `_unit`, or `_format` after they are set by **`DFSDsetlengths`**. The maximum default string length is 255.

FORTRAN

```
integer function dsgdist(dim, label, unit, format)
    integer        dim
    character*(*)  label, unit, format
```

DFSDgetfillvalue

```
intn DFSDgetfillvalue(VOIDP fill_value)
```

fill_value	OUT:	Number to be stored in any unwritten locations of a scientific data set

Purpose Gets the value used to fill in any unwritten location in a scientific data set.

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

Description Returns the value used to fill the unwritten locations in a scientific data set. The fill value is set by **`DFSDsetfillvalue`** and returned in the variable `fill_value`.

Note that **`DFSDgetfillvalue`** does not take a file name as an argument. As a result, a DFSD call to initialize the file information structures is required before calling **`DFSDgetfillvalue`**. One such call is **`DFSDgetdims`**.

FORTRAN

```
integer function dsgfill(fill_value)
    <valid type>   fill_value
```

dsgnt

DFSDgetNT

```
intn DFSDgetNT(int32 *num_type)
```

num_type OUT: Number type of data in the scientific data set

Purpose Gets the number type of the next data set to be read.

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

Description Note that **`DFSDgetNT`** does not take a file name as an argument. As a result, a DFSD call to initialize the file information structures is required before calling **`DFSDgetNT`**. One such call is **`DFSDgetdims`**.

Valid values for **`DFSDgetNT`** are of the general form `DFNT_<numbertype>`. The following are valid symbolic names and their number types:

32-bit float	`DFNT_FLOAT32`	5
64-bit float	`DFNT_FLOAT64`	6
8-bit signed int	`DFNT_INT8`	20
8-bit unsigned int	`DFNT_UINT8`	21
16-bit signed int	`DFNT_INT16`	22
16-bit unsigned int	`DFNT_UINT16`	23
32-bit signed int	`DFNT_INT32`	24
32-bit unsigned int	`DFNT_UINT32`	25

FORTRAN
```
integer function dsgnt(num_type)
    integer*4 num_type
```

DFSDgetrange

```
intn DFSDgetrange(VOIDP max, VOIDP min)
```

max	OUT:	High value stored with the scientific data set
min	OUT:	Low value stored with the scientific data set

Purpose Reads the maximum and minimum values as they are stored with the scientific data set.

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

Description The *max* and *min* values must be set via a call to **DFSDsetrange**. They are not automatically stored when a data set is written to a file. Since these values are supposed to relate to the data itself, it is assumed that they are of the same number type as the data itself. One implication of this is that in the C version of **DFSDgetrange** the arguments are pointers, rather than simple variables, whereas in the FORTRAN version they are simple variables of the same type as the data array.

Neither **DFSDgetrange** nor **DFSDgetdata** compare the *max* and *min* values stored with the data set to the actual values in the data set; they merely retrieve the data. As a result, the maximum and minimum values may not always reflect the actual maximum and minimum values in the data set. In some cases the max and min values may actually lie outside the range of values in the data set.

Example In this example 16-bit data is read from an HDF file. Notice that *max* and *min* are the same data type as the data set array `data`.

```
int16 max, min, data[100][100];
...
DFSDgetrange(&max, &min);
DFSDgetdata("myfile.hdf", rank, dims, data);
```

FORTRAN

```
integer function dsgrang(max, min)
    <valid type> max, min
```

DFSDgetslice

dsgslc

DFSDgetslice

```
intn DFSDgetslice(char *filename, int32 winst[], int32 windims[], VOIDP data,
          int32 dims[])
```

filename	IN:	Name of HDF file
winst	IN:	Array containing the coordinates for the start of the slice
windims	IN:	Array containing the dimensions of the slice
data	OUT:	Array for returning slice
dims	OUT:	Dimensions of array data

Purpose

Reads part of a scientific data set from a file.

Return value

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

Description

DFSDgetslice accesses the data set last accessed by **DFSDgetdims**. If **DFSDgetdims** has not been called for the named file, **DFSDgetslice** gets a slice from the next data set in the file.

Array *winst* specifies the coordinates of the start of the slice. Array *windims* gives the size of the slice. The number of elements in *winst* and *windims* must be equal to the rank of the data set. For example, if the file contains a 3D data set, *winst* may contain the values {2, 4, 3}, while *windims* contains the values {3, 1, 4} and the *dims* should be at least {3, 1, 4}, the same size as the slice. This will extract a 3 x 4, two-dimensional slice, containing the elements between (2, 4, 3) and (4, 4, 6) from the original data set.

The *data* array is the array into which the slice is read. It must be at least as big as the desired slice. The *dims* array is the array containing the actual dimensions of the array *data*. The user assigns values to *dims* before calling **DFSDgetslice**.

All parameters assume FORTRAN-style 1-based arrays.

DFSDgetslice is obsolete in favor of **DFSDreadslab**. **DFSDreadslab** is the recommended function call to use when reading hyperslabs (previously known as data slices). HDFv3.3 will continue to support **DFSDgetslice** only to maintain backward compatibility with HDF applications built on earlier versions of the library.

dsgslc

Example Reading Two Slices.

```
#include "hdf.h"
main c
{
    int i, rank;
    int32 dimsizes[2];

    DFSDgetdims("my_file", &rank, dimsizes, 2);

/* Starting at (3,4) read 4 x 6 window. Use (3,4) rather than (2,3)
*//* because FORTRAN-style indexing is used.*/

    getit("myfile", 3,4,4,6);
    starting at (1,10) read 10 x 2 window */
    getit("myfile", 1,10,10,2);
}

getit(filename, st0, st1, rows, cols)
int st0, st1, rows, cols;
char *filename;
    {
    int i, j;
    int32 winst[2], windims[2], dims[2];
    float32 data[500];

    winst[0]=st0; winst[1]=st1;
    dims[0] = windims[0] = rows;
    dims[1] = windims[1] = cols;
    DFSDgetslice(filename, winst, windims, data, dims);

    for (i=0; i<rows; i++) {
                    printf("\n");
                    for (j=0; j<cols; j++)
                    printf("%5.0f%c",data[i*cols+j], ' ');
                    }
    printf("\n");
}
```

FORTRAN

```
integer function dsgslc(filename, winst, windims, data, dims)
    character*(*)   filename
    integer*4       winst(*), windims(*), dims(*)
    <valid type>    data(*)
```

dslref

DFSDlastref

```
intn DFSDlastref()
```

Purpose	Gets the most recent reference number used in writing or reading a scientific data set.
Return value	Returns the reference number for the last accessed scientific data set if successful and `FAIL(-1)` otherwise.
Description	**`DFSDlastref`** returns the value of the last reference number of a scientific data set read from or written to the file.
FORTRAN	`integer function dslref()`

DFSDndatasets

```
intn DFSDndatasets(char *filename)
```

filename	IN:	Name of the HDF file containing the scientific data sets

Purpose: Returns the number of scientific data sets in the file.

Return value: Returns the number of data sets if successful and `FAIL(-1)` otherwise.

Description: In HDFv3.3, **`DFSDndatasets`** replaces **`DFSDnumber`**. In order to maintain backward compatibility with existing HDF applications, HDF will continue to support **`DFSDnumber`**. However, it is recommended that all new applications use **`DFSDndatasets`** instead of **`DFSDnumber`**.

FORTRAN

```
integer function dsnum(filename)
    character*(*) filename
```

DFSDpre32sdg

dsp32sd

DFSDpre32sdg

```
intn DFSDpre32sdg(char *filename, uint16 ref, intn *ispre32)
```

filename	IN:	The name of the HDF file containing the scientific data set
ref	IN:	Reference number of SDG
ispre32	OUT:	Pointer to results of the pre-HDF 3.2 inquiry

Purpose Tests if the scientific data set with the specified reference number was created by an HDF library that precedes HDF 3.2.

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

Description This routine is for programmers who need to know whether a scientific data set was written by a version of the HDF library earlier than HDF3.2. If the scientific data set was created with a version of HDF prior to v3.2, `ispre32` will be set to `1`, otherwise it will be set to `0`. Based on this information, programmers can decide whether or not to transpose the corresponding array.

FORTRAN

```
integer function dsp32sd(filename, ref, ispre32)
    character*(*)  filename
    integer*2      ref
    integer        ispre32
```

dspdata

DFSDputdata

```
intn DFSDputdata(char *filename, intn rank, int32 dimsizes[], VOIDP data)
```

filename	IN:	Name of file to store the scientific data set in
rank	IN:	Number of dimensions of data array to be stored
dimsizes	IN:	Array that holds sizes of dimensions
data	IN:	Array holding data to be stored

Purpose — Writes a scientific data and related information to an HDF file.

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

Description — **`DFSDputdata`** will write data to an existing file by destroying the contents of the original file. Use it with caution. If a new filename is used, **`DFSDputdata`** functions exactly like **`DFSDadddata`**.

FORTRAN

```
integer function dspdata(filename, rank, dimsizes, data)
    character*(*)  filename
    integer        rank
    integer*4      dimsizes(*)
    <valid type>   data(*)
```

DFSDputslice

dspslc

DFSDputslice

```
intn DFSDputslice(int32 windims[], VOIDP source, int32 dims[])
```

windims	IN:	Window dimensions that specify the size of the slice to be written
source	IN:	The array in memory containing the slice
dims	IN:	The dimension of the array source

Purpose

Writes part of a scientific data set to a file.

Return value

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

Description

DFSDputslice takes some part of an array in memory and stores it as part of the scientific data set array last specified by **DFSDsetdims**. Slices must be stored contiguously.

Array *windims* ("window dimensions") specifies the size of the slice to be written. The *windims* array must contain as many elements as there are dimensions in the entire scientific data set array. The `source` argument is an array in memory containing the slice and `dims` is an array containing the dimensions of the array `source`.

Notice that *windims* and `dims` need not be the same. The *windims* argument could refer to a sub-array of `source`, in which case only a portion of `source` is written to the scientific data array.

All parameters assume FORTRAN-style 1-based arrays.

DFSDputslice is obsolete in favor of **DFSDwriteslab**. **DFSDwriteslab** is the recommended function call to use when writing hyperslabs (previously known as data slices). HDFv3.3 will continue to support **DFSDputslice** only to maintain backward compatibility with HDF applications built on earlier versions of the library.

dspslc

Example

Suppose we want to create a 7x12 scientific data set array, and we want to write it using slices. Suppose also that the array from which we get our data is a 10x12 array in memory called `source`, and we want to write the 7x12 "window" from top of that array.

```
intn rank;
int SDSdims[2], sourcedims[2], windims[2];
float data[10][12];

/*code that builds the array source goes here */
...

SDSdims[0]=7;
SDSdims[1]=12;
sourcedims[0]=10;
sourcedims[1]=12;

DFSDsetdims(2, SDSdims);

write out scientific data set in slices */
DFSDstartslice(filename);

windims[0]=2; windims[1]=12; /* {(1,1) to (2,12)} */
DFSDputslice(windims, &data[0][0], sourcedims);

windims[0]=4; windims[1]=12; /* {(3,1) to (6,12)} */
DFSDputslice(windims, &data[2][0], sourcedims);

windims[0]=1; windims[1]=4; /* {(7,1) to (7,4)} */
DFSDputslice(windims, &data[6][0], sourcedims);

windims[0]=1; windims[1]=8; /* {(7,5) to (7,12)} */
DFSDputslice(windims, &data[6][4], sourcedims);

windims[0]=3; windims[1]=12; /* {(8,1) to (10,12)} */
DFSDputslice(windims, &data[7][0], sourcedims);

DFSDendslice();
```

FORTRAN

```
integer function dspslc(windims, source, dims)
   integer*4      windims(*), dims(*)
   <valid type>   source(*)
```

dsrref

DFSDreadref

```
intn DFSDreadref(char *filename, uint16 ref)
```

filename	IN:	The name of the HDF file containing the scientific data set
ref	IN:	Reference number for next **DFSDgetdata** call

Purpose Specifies the reference number for the data set to be read during the next read operation.

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

Description **DFSDreadref** sets the reference number for the scientific data set to get next. This routine is most likely to be used in conjunction with **DFANgetlablist**, which returns a list of labels for a given tag together with their reference numbers. It provides a sort of random access to scientific data sets.

There is no guarantee that reference numbers appear in sequence in an HDF file, so it is not generally safe to assume that a reference number is a sequence number for a scientific data set.

FORTRAN

```
integer function dsrref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

dsrslab

DFSDreadslab

```
intn DFSDreadslab(char *filename, int32 start[], int32 slab_size[],
          int32 stride[], VOIDP buffer, int32 buffer_size[])
```

filename	IN:	Name of HDF file
start	IN:	Array of size [rank] containing the coordinates for the start of the slab
slab_size	IN:	Array of size [rank] containing the size of each dimension in the slab
stride	IN:	Sub-sampling (not yet implemented)
buffer	OUT:	Space allocated to hold the extracted slab
buffer_size	OUT:	Array containing the dimensions of the *buffer*

Description — Reads a sub-array of data from any scientific data set.

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

Description — **DFSDreadslab** will sequence to the next scientific data set if **DFSDgetdims** or **DFSDgetdata** are not called earlier. The *start*`[]` indices are relative to 1. The rank of *start*`[]` must be the same as the number of dimensions of the specified variable. The elements of `[]` must be no larger than the dimensions of the scientific data set in order. The stride feature is not currently implemented. For now just pass the *start*`[]` array as the argument for *stride*`[]` where it will be ignored.

To extract a slab of lower dimension than that of the data set, enter 1 in the *slab_size* array for each omitted dimension. For example, to extract a 2D slab from a 3D data set, specify the beginning coordinates in 3 dimensions and enter a 1 for the missing dimension in the *slab_size* array. More specifically, to extract a 3 x 4 slab containing the elements (6, 7, 8) through (8, 7, 11) specify the beginning coordinates as {6, 7, 8} and the slab size as {3, 1, 4}.

FORTRAN

```
integer function dsrslab(filename, start, slab_size, stride,
buffer, buffersize)
    character*(*)   filename
    integer*4       start(*), slab_size(*),
    integer*4       stride(*), buffer_size(*)
    <valid type>    buffer(*)
```

dsfirst

DFSDrestart

```
intn DFSDrestart()
```

Purpose	Forces the next read operation to read the first scientific data set in the file.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	To cause the next read command to read from the first scientific data set in the file, rather than the scientific data set following the one that was most recently read.
FORTRAN	`integer function dsfirst()`

.

DFSDsetcal

```
intn DFSDsetcal(float64 cal, float64 cal_err, float64 off, float64 off_err,
          int32 num_type)
```

cal	IN:	Calibration factor
cal_err	IN:	Calibration error
off	IN:	Uncalibrated offset
off_err	IN:	Uncalibrated offset error
num_type	IN:	Number type of uncalibrated data

Purpose — Sets calibration information associated with data

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

Description — This routine sets the calibration record associated with a data set. A calibration record contains four 64-bit floating point values followed by a 32-bit integer, to be interpreted as follows:

`cal`	calibration factor
`cal_err`	calibration error
`offset`	uncalibrated offset
`offset_err`	uncalibrated offset error
`num_type`	number type of uncalibrated data

The relationship between a value 'iy' stored in a data set and the actual value 'y' is defined as:

y = cal * (iy - offset)

The variable *offset_err* contains a potential error of offset, and *cal_err* contains a potential error of *cal*. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

DFSDsetcal works like other **DFSDset*** routines, with one exception: the calibration information is automatically cleared after a call to **DFSDputdata** or **DFSDadddata**. Hence, **DFSDsetcal** must be called anew for each data set that is to be written.

dsscal

Example

Suppose the values in a data set `y[]` are as follows:

```
y[6]={1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5}
```

By defining `cal=0.50` and `offset=-200.0` and applying the calibration formula, the calibrated data set `iy[]` becomes as follows:

```
iy[6]={2, 4, 5, 11, 26, 81}
```

The array `iy[]` can then be stored as integers.

FORTRAN

```
integer function dsscal(cal, cal_err, offset, offset_err, num_type)
   real*8          cal, cal_err, offset, offset_err
   integer*4       num_type
```

DFSDsetdatastrs

```
intn DFSDsetdatastrs(char *label, char *unit, char *format, char *coordsys)
```

label	IN:	Label that describes the data
unit	IN:	Unit to be used with the data
format	IN:	Format to be used in displaying the data
coordsys	IN:	Coordinate system

Purpose — Sets the label, unit, format, and coordinate system for the next data set written to file.

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

Description — **`DFSDsetdatastrs`** sets label, unit, and format specifications for the data.

Example — In this example a 200 x 300 data array is written to a file called '`myfile.hdf`', together with label, unit, and format information about the data. In this example we assume that the *`coordsys`* parameter is of no interest to the user, so the empty string (`""`) is given as the fourth parameter to **`DFSDsetdatastrs`**.

```
float32 press1[200][300];
int dims[2];
...

dims[0] = 200;
dims[1] = 300;

DFSDsetdims(2, dims);
DFSDsetdatastrs ("pressure 1", "Pascals", "E15.9", " ");
DFSDadddata("myfile.hdf", 2, dims, press1);
```

FORTRAN

```
integer function dssdast(label, unit, format, coordsys)
   character*(*) label, unit, format, coordsys
```

DFSDsetdims

dssdims

DFSDsetdims

```
intn DFSDsetdims(intn rank, int32 dimsizes[])
```

rank	IN:	Number of dimensions
dimsizes	IN:	Dimensions of the scientific data set

Purpose

Sets the rank and dimension sizes for all subsequent scientific data sets written to the file.

Return value

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

Description

`DFSDsetdims` sets the rank and dimension sizes for succeeding scientific data sets. This routine must be called before calling either **`DFSDsetdimstrs`** or **`DFSDsetdimscale`**. **`DFSDsetdims`** need not be called if other set routines are not called and the correct dimensions are supplied in **`DFSDputdata`** or **`DFSDadddata`**.

If the rank or dimension sizes change, all previous set calls are cleared, except for the number type which is set by calling **`DFSDsetNT`**.

FORTRAN

```
integer function dssdims(rank, dimsizes)
   integer         rank
   integer*4       dimsizes(*)
```

DFSDsetdimscale

```
intn DFSDsetdimscale(intn dim, int32 dimsize, VOIDP scale)
```

dim	IN:	Dimension this scale corresponds to
dimsize	IN:	Size of scale array
scale	IN:	Array of values defining reference points along a specified dimension

Purpose Defines the scale for a dimension.

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

Description A scale is a 1-dimensional array whose values describe reference points along one dimension of the data set. For example, a 2-dimensional data set representing points on a map could have two scales, one representing points of latitude, and the other points of longitude.

Example In this example a 200 x 300 data array is written to a file called `"myfile.hdf"`, together with scales for each dimension. It is assumed that the arrays `latscale` and `longscale` have been assigned values that define the corresponding scales.

```
float32 press1[200][300];
float32 latscale[200], longscale[300];
int dims[2];
...
dims[0] = 200;
dims[1] = 300;

DFSDsetdims(2, dims);
DFSDsetdimscale(1, dims[0], latscale);
DFSDsetdimscale(2, dims[1], longscale);
DFSDadddata("myfile.hdf", 2, dims, press1);
```

FORTRAN

```
integer function dssdisc(dim, dimsize, scale)
    integer        dim
    integer*4      dimsize(*)
    <valid type>   scale(*)
```

DFSDsetdimstrs

dssdist

DFSDsetdimstrs

```
intn DFSDsetdimstrs(intn dim, char *label, char *unit, char *format)
```

dim	IN:	Dimension this label, unit and format refer to
label	IN:	Label that describes this dimension
unit	IN:	Unit to be used with this dimension
format	IN:	Format to be used to display scale

Purpose Sets the label, unit, and format strings corresponding to the specified dimension.

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

Description In both FORTRAN and C programs, `dim=1` for the first dimension, and `dim=2` for the second dimension. If the user is not interested in one or more strings, empty strings can be used as parameters for the **`DFSDsetdimstrs`** call. For example, `DFSDsetdimstrs(1, "vertical", "", "")` will set the label for the first dimension to "vertical" and set the unit and format to empty strings.

Example In this example a 200x300 data array is written to a file called `'myfile.hdf'`, together with label, unit, and format information about each dimension.

```
float32 press1[200][300];
int      dims[2];
...
dims[0] = 200;
dims[1] = 300;

DFSDsetdims(2, dims);
DFSDsetdimstrs(1, "vertical", "cm", "F10.2");
DFSDsetdimstrs(2, "horizontal", "m", "F10.3");
DFSDadddata(myfile.hdf', 2, dims, press1);
```

FORTRAN

```
integer function dssdist(dim, label, unit, format)
   integer        dim
   character*(*)  label, unit, format
```

dssfill

DFSDsetfillvalue

```
intn DFSDsetfillvalue(VOIDP fill_value)
```

fill_value IN: Number to be stored in any unwritten locations of a scientific data set

Purpose Set the value used to fill in any unwritten location in a scientific data set.

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

Description This routine sets the fill value to be used for the specified data set. Since data is written in hyperslabs, it is possible that not all of the possible locations in a given data set are written to. The fill value is placed into all unwritten locations.

It is assumed that the fill value has the same number type as the data set.

FORTRAN
```
integer function dssfill(fill_value)
    <valid type>   fill_value
```

DFSDsetlengths

dsslens

DFSDsetlengths

```
intn DFSDsetlengths(intn label_len, intn unit_len, intn format_len, intn coords_len)
```

label_len	IN:	Maximum length of any label string
unit_len	IN:	Maximum length of any unit string
format_len	IN:	Maximum length of any format string
coords_len	IN:	Maximum length of any coordinate system string

Purpose

Sets the maximum lengths for the strings that will hold labels, units, formats, and the name of the coordinate system.

Return value

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

Description

`DFSDsetlengths`sets maximum lengths for strings that will hold labels, units, formats, and the name of the coordinate system. These lengths are used by the routines **`DFSDgetdimstrs`** and **`DFSDgetdatastrs`** to determine the maximum lengths of strings that they get from the file.

Normally, **`DFSDsetlengths`** is not needed. If it is not called, default maximum lengths of 255 are used for all strings.

FORTRAN

```
integer function dsslens(label_len, unit_len, format_len,
coords_len)
    integer label_len, unit_len, format_len, coords_len
```

DFSDsetNT

```
intn DFSDsetNT(int32 num_type)
```

num_type IN: Number type of data to be written

Purpose Sets the number type for the data to be written in the next write operation.

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

Description **`DFSDsetNT`** must be called if a number type other than `float32` is to be stored. **`DFSDsetNT`** and **`DFSDsetdims`** can be called in any order, but they should be called before any other **`DFSDset*`** functions and before **`DFSDputdata`** or **`DFSDadddata`**.

Valid parameter values for **`DFSDsetNT`** are of the general form `DFNT_<numbertype>`. If you include the file `hdf.h` in your program, you can use the following symbolic names for the number types:

32-bit float	`DFNT_FLOAT32 5`
64-bit float	`DFNT_FLOAT64 6`
8-bit signed int	`DFNT_INT8 20`
8-bit unsigned int	`DFNT_UINT8 21`
16-bit signed int	`DFNT_INT16 22`
16-bit unsigned int	`DFNT_UINT16 23`
32-bit signed int	`DFNT_INT32 24`
32-bit unsigned int	`DFNT_UINT32 25`

Example Assuming that `DFNT_INT8` has been defined and `i8data` is an array with 8-bit integer data, the following code fragments write out 8-bit integers to a scientific data set.

```
DFSDsetNT(DFNT_INT8);
DFSDadddata("myfile.hdf", rank, dims, i8data);
```

FORTRAN

```
integer function dssnt(num_type)
    integer*4 num_type
```

DFSDsetrange

dssrang

DFSDsetrange

```
intn DFSDsetrange(VOIDP max, VOIDP min)
```

max	IN:	High value in range
min	IN:	Low value in range

Purpose

Stores the specified maximum and minimum data values.

Return value

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

Description

Since the max/min values are supposed to relate to the data itself, it is assumed that the number type of *max* and *min* is the same as the type of the data. One implication of this is that in the C version of **DFSDsetrange** the arguments are pointers, rather than simple variables, whereas in the FORTRAN version they are simple variables of the same type as the data array.

This routine does not compute the maximum and minimum values; it merely stores the values it is given. As a result, the maximum and minimum values may not always reflect the actual maximum and minimum values in the data array.

When the maximum and minimum values are written to a file, the HDF element that holds these values is cleared, because it is assumed that subsequent data sets will have different values for *max* and *min*.

Example

In this example 16-bit data is written to an HDF file. Notice that *max* and *min* must be the same data type as the scientific data set array data.

```
int16 max, min, data[100][100];
...
DFSDsetrange(&max, &min);
DFSDadddata("myfile.hdf", rank, dims, data);
```

FORTRAN

```
integer function dssrang(max, min)
    <valid type> max, min
```

DFSDstartslab

```
intn DFSDstartslab(char *filename)
```

filename IN: Name of the HDF file

Purpose Prepares the DFSD interface to write a slab of data to a scientific data set.

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

Description **`DFSDsetdims`** must be called before calling **`DFSDstartslab`**. No call which needs a file open may be made after a **`DFSDstartslab`** call until **`DFSDendslab`** is called. This routine will write out the fill values if **`DFSDsetfillvalue`** is called before this routine.

FORTRAN
```
integer function dssslab(filename)
    character*(*) filename
```

DFSDstartslice

dssslc

DFSDstartslice

```
intn DFSDstartslice(char *filename)
```

filename	IN:	Name of the HDF file

Purpose

Prepares the interface to write a slice to a file.

Return value

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

Description

Before calling **DFSDstartslice**, you must call **DFSDsetdims** to specify the dimensions of the data set to be written to the file. **DFSDstartslice** always appends a new data set to an existing file.

Remember, you must call **DFSDstartslice** before calling **DFSDputslice** or **DFSDendslice**.

DFSDstartslice is obsolete in favor of **DFSDstartslab**. **DFSDstartslab** is the recommended function call to use when beginning hyperslab operations. HDFv3.3 will continue to support **DFSDstartslice** only to maintain backward compatibility with HDF applications built on earlier versions of the library.

FORTRAN

```
integer function dssslc(filename)
   character*(*) filename
```

DFSDwriteref

```
intn DFSDwriteref(char *filename, uint16 ref)
```

filename	IN:	Name of the HDF file
ref	IN:	Reference number for next add or put operation

Purpose To specify the reference number of the data set to be written when **DFSDputdata** or **DFSDadddata** is next called.

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

Description **DFSDwriteref** sets the reference number of the data set to write next. There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the sequence number for a data set.

FORTRAN

```
integer function dswref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

DFSDwriteslab

dswslab

DFSDwriteslab

```
intn DFSDwriteslab(int32 start[], int32 stride[], int32 count[], VOIDP data)
```

start	IN:	Array containing the coordinates of the start of the slab
stride	IN:	Array containing the dimensions for subsampling
count	IN:	Array of size= rank containing the size of the slab
data	IN:	Array to hold the floating point data to write

Purpose
Writes a slab of data to a scientific data set.

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

Description
The *start* indices are relative to 1. The rank of *start* must be the same as the number of dimensions of the specified variable. The elements of *start* must be no larger than the scientific data set's dimensions in order. The stride feature is not currently implemented. For now just pas the *start* array as the argument for *stride* where it will be ignored.

The rank of *count* must be the same as the number of dimensions of the specified variable. The elements of *count* must be no larger than the scientific data sets's dimensions in order. The order in which the data will be written into the specified hyperslab is with the last dimension varying fastest. The *data* should be of the appropriate type for the data set. Note that neither the compiler nor HDF software can detect if the wrong type of data is used.

FORTRAN

```
integer function dswslab(start, stride, count, data)
   integer*4      start(*), stride(*), count(*)
   <valid type>   data(*)
```

DF24addimage

```
intn DF24addimage(char *filename, VOIDP image, int32 width, int32 height)
```

filename	IN:	Name of HDF file
image	IN:	Pointer to the image array
width	IN:	Number of columns in the image
height	IN:	Number of rows in the image

Purpose — To write out a 24-bit image.

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

Description — **`DF24addimage`** appends a 24-bit raster image set for the image to the file. Array *`image`* is assumed to be `width` x `height` x 3 bytes. In FORTRAN, the dimensions of the array *`image`* must be the same as the dimensions of the image itself.

The order in which you declare dimensions is different between C and FORTRAN. Ordering varies because FORTRAN arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the last coordinate varies fastest).

When **`DF24putimage`** writes an image to a file, it assumes row-major order. The FORTRAN declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension. To take this into account as you build your image in your FORTRAN program, you need to build the image "on its side."

FORTRAN

```
integer function d2aimg(filename, image, width, height)
    character*(*)  filename
    <valid type>   image(*)
    integer*4      width, height
```

DF24getdims

d2gdims

DF24getdims

```
intn DF24getdims(char *filename, int32 *width, int32 *height, intn *il)
```

filename	IN:	Name of HDF file
width	OUT:	For returning width of image
height	OUT:	For returning height of image
il	OUT:	For returning interlace of image in file

Purpose

To get dimensions and interlace storage scheme of next image RIS.

Return value

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

Description

`DF24getdims` retrieves the dimensions and interlace of the image. If the file is being opened for the first time, **`DF24getdims`** returns information about the first image in the file. If an image has already been read, **`DF24getdims`** finds the next image. Thus, images are read in the same order in which they were written to the file.

If you know the dimensions and interlace of the image beforehand, there is no need to call **`DF24getdims`**. Simply allocate arrays with the proper dimensions for the image and let **`DF24getimage`** read in the images. If, however, you do not know the values of *`width`* and *`height`*, you must call **`DF24getdims`** to get them and then use them to determine the right amount of space needed for the array image.

Successive calls to **`DF24getdims`** and **`DF24getimage`**, respectively, retrieve all of the images in the file in the sequence in which they were written.

Interlace codes: `0` = pixel interlacing; `1` = scan-line interlacing; `2` = scan-plane interlacing.

FORTRAN

```
integer function d2gdims(filename, width, height, il)
   character*(*)  filename
   integer*4      width, height
   integer        il
```

DF24getimage

```
intn DF24getimage(char *filename, VOIDP image, int32 width, int32 height)
```

filename	IN:	Name of HDF file
image	OUT:	Pointer to space to return image
width	IN:	Number of columns in the image
height	IN:	Number of rows in the image

Purpose — To get image from next 24-bit raster image set.

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

Description — **`DF24getimage`** retrieves the image and stores it in an array. If **`DFR24getdims`** has not been called, **`DFR24getimage`** finds the next image in the same way that **`DFR24getdims`** does.

The amount of space allocated for the image should be `width` x `height` x 3 bytes.

To specify that the next call to **`DF24getimage`** should read the raster image using an interlace other than the interlace used to store the image in the file, make a call to **`DF24reqil`**.

FORTRAN

```
integer function d2gimg(filename, image, width, height)
   character*(*)  filename
   <valid type>   image(*)
   integer*4      width, height
```

DF24lastref

d2lref

DF24lastref

```
uint16 DF24lastref()
```

Purpose	To get last reference number written or read for a 24-bit raster image set.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	This routine is primarily used for attaching annotations to 24-bit images and adding 24-bit images to Vgroups. **`DF24lastref`** returns the reference number of last 24-bit raster image read or written.
FORTRAN	`integer function d2lref()`

DF24nimages

```
intn DF24nimages(char *filename)
```

filename	IN:	Name of the HDF file

Purpose	To count the number of 24-bit raster images contained in an HDF file.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	**`DF24nimages`** counts the number of 24-bit images stored in the file.
FORTRAN	`integer function d2nims(filename)` `  character*(*)  filename`

DF24putimage

d2pimg

DF24putimage

```
intn DF24putimage(char *filename, VOIDP image, int32 width, int32 height)
```

filename	IN:	Name of HDF file
image	IN:	Pointer to the image array
width	IN:	Number of columns in the image
height	IN:	Number of rows in the image

Purpose
To write out a 24-bit image as the first image in the file, overwriting any information previously stored in the file.

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

Description
`DF24putimage` writes a 24-bit raster image to a file. The array *`image`* is assumed to be `width` x `height` x 3 bytes. **`DF24putimage`** overwrites any information that exists in the HDF file. To append a new image to a file instead of overwriting an existing file, use **`DF24addimage`**.

FORTRAN

```
integer function d2pimg(filename, image, width, height)
   character*(*)   filename
   <valid type>    image(*)
   integer*4       width, height
```

DF24readref

```
intn DF24readref(char *filename, uint16 ref)
```

filename	IN:	Name of HDF file containing image
ref	IN:	Reference number for next **DF24getimage**

Purpose

To specify the reference number of the image to be read when **DF24getimage** is next called.

Return value

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

Description

DF24readref sets the reference number for the image to get next. You will most likely use this routine in conjunction with **DFANgetlablist**, which returns a list of labels for a given tag together with their reference numbers. It provides, in a sense, a random access to images.

There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the sequence number for an image.

FORTRAN

```
integer function d2rref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

DF24reqil

d2reqil

DF24reqil

```
intn DF24reqil(intn il)
```

il	IN:	Interlace to get next image with

Purpose — To cause next **DF24getimage** to store image in memory with the specified interlace.

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

Description — **DF24reqil** specifies an interlace to be used in place of the interlace indicated in the file when the next raster image is read. Regardless of what interlace scheme is used to store the image, **DF24reqil** causes the image to be loaded into memory and be interlaced according to the specification of *il*.

Since a call to **DF24reqil** may require a substantial reordering of the data, I/O performance could be adversely affected; e.g. it could result in much slower I/O performance than would be achieved if no change in interlace were requested.

Interlace codes: `0` = pixel interlacing; `1` = scan-line interlacing; `2` = scan-plane interlacing.

FORTRAN

```
integer function d2reqil(il)
    integer il
```

DF24restart

```
intn DF24restart()
```

Purpose	To cause the next get to read from the first RIS24 in the file, rather than the RIS24 following the one that was most recently read.
Return value	Returns `SUCCEED(0)` if successful and `FAIL(-1)` otherwise.
Description	**`DF24restart`** causes the next get command to read from the first 24-bit raster image set in the file, rather than the next one.
FORTRAN	`integer function d2first()`

DF24setcompress

```
intn DF24setcompress(int32 type, comp_info *cinfo)
```

type	IN:	Type of compression
cinfo	IN:	Pointer to compression information structure

Purpose Set the type of compression to use when writing the next 24-bit raster image.

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

Description This routines provides a method for compressing the next raster image written. The *type* can be one of the following values: COMP_NONE, COMP_JPEG, COMP_RLE, COMP_IMCOMP. COMP_NONE is the default for storing images if this routine is not called, therefore images are not compressed by default. COMP_JPEG compresses images with a JPEG algorithm, which is a lossy method. COMP_RLE uses lossless run-length encoding to store the image. COMP_IMCOMP uses a lossy compression algorithm called IMCOMP, and is included for backward compatibility only.

The *cinfo* parameter is a pointer to a union defined as:

```
union {
    struct {
        intn quality;
        intn force_baseline;
    } jpeg;
} comp_info;
```

This union is defined to provide future expansion, but is currently only used by the COMP_JPEG compression type. A pointer to a valid comp_info union is required for all compression types other than COMP_JPEG, but the values in the union are not used. The comp_info union is declared in the header file hdf.h and is shown here for informative purposes only, it should not be re-declared in a user program.

For COMP_JPEG compression, the *quality* member of the *jpeg* structure must be set to the quality of the stored image. This number can vary from 100, the best quality, to 0, terrible quality. All images stored with COMP_JPEG compression are stored in a lossy manner, even images stored with a quality of 100. The ratio of size to perceived image quality varies from image to image, some experimentation may be required to determine an acceptable *quality* factor for a given application. The *force_baseline* parameter determines whether the quantization tables used during compression are forced to the range 0-255. The *force_baseline* parameter should normally be set to 1 (forcing baseline results), unless special applications require non-baseline images to be used.

FORTRAN None

DF24setdims

```
intn DF24setdims(int32 width, int32 height)
```

width	IN:	Number of columns in the image
height	IN:	Number or rows in the image

Purpose — Set the dimensions for the next image to be written to a file.

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

Description — Sets the dimensions of the next image to be written to the HDF file.

FORTRAN

```
integer function d2sdims(width, height)
    integer*4 width, height
```

DF24setil

d2setil

DF24setil

```
intn DF24setil(intn il)
```

il	IN:	Interlace of image

Purpose — To set interlace scheme to be used on subsequent writes.

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

Description — **`DF24setil`** sets the interlace to be used when writing out the raster image set for the 24-bit image. If **`DF24setil`** is not called, the interlace code is assumed to be 0.

Interlace codes: `0` = pixel interlacing; `1` = scan-line interlacing; `2` = scan-plane interlacing.

FORTRAN

```
integer function d2setil(il)
    integer il
```

DF24writeref

```
intn DF24writeref(char *filename, uint16 ref)
```

filename	IN:	HDF file containing image
ref	IN:	Reference number for next **DF24putimage** or **DF24addimage**

Purpose — To specify the reference number of the image to be written when **DF24addimage** or **DF24putimage** is next called.

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

Description — **DF24writeref** sets the reference number of the image to write next. It is unlikely that you will need this routine, but if you do, use it with caution. There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the sequence number for an image.

FORTRAN

```
integer function d2wref(filename, ref)
    character*(*)  filename
    integer*2      ref
```

Happendable

```
intn Happendable(int32 H_id)
```

H_id	IN:	Access identifier returned by **Hstartwrite**

Purpose	Appends data to a data set without the use of linked blocks.
Return value	Returns `SUCCEED(0)` if data element can be appended and `FAIL(-1)` otherwise
Description	If a data element is at the end of a file **Happendable** allows **Hwrite** to write past the end of the file without the use of linked blocks.
FORTRAN	None

Hclose

hclose

Hclose

```
intn Hclose(int32 file_id)
```

file_id	IN:	The HDF file identifier returned by **Hopen**

Purpose — Closes the access path to the file.

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

Description — The file identifier *file_id* is validated before the file is closed. If the identifier is valid, the function closes the access path to the file.

If there are still access identifiers attached to the file, the error `DFE_OPENAID` is placed on the error stack, `FAIL(-1)` is returned, and the file remains open. This is a fairly common error when developing new interfaces. See **Hendaccess** for discussion of this problem.

FORTRAN

```
integer function hclose(file_id)
    integer*4 file_id
```

Hdeldd

```
intn Hdeldd(int32 file_id, uint16 tag, uint16 ref)
```

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of data descriptor to delete
ref	IN:	Reference number of data descriptor to delete

Purpose Deletes a tag and reference number from the data descriptor list.

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

Description Removes the tag and reference number for a data object from the data descriptor list. Once the data descriptor is removed, the data in the data object becomes inaccessible. To remove inaccessible data from an HDF file, use the utility **hdfpack**.

Hdeldd only deletes the specified tag and reference number from the data descriptor list. Data objects containing the deleted tag and reference number are not automatically updated. For example, if the tag and reference number deleted from the descriptor list referenced an object in a Vgroup, the tag and reference number will still exist in the Vgroup even though the data is inaccessible.

FORTRAN None

Hdupdd

```
intn Hdupdd(int32 file_id, uint16 tag, uint16 ref, uint16 old_tag, uint16 old_ref)
```

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag for the duplicate data descriptor
ref	IN:	Reference number for the duplicate data descriptor
old_tag	IN:	Tag for original data descriptor
old_ref	IN:	Reference number for the original data descriptor

Purpose Generates a new data descriptor for the data in an existing data object.

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

Description Duplicates the specified data descriptor in the data descriptor list. Data descriptors contain the following information: `tag`, `reference number`, `length`, and `offset`. When a data descriptor is duplicated, it's `length` and `offset` are added to the data descriptor list under a new tag and reference number.

Because duplicate data descriptors contain the same `length` and `offset` information, both descriptors point to the same data element in the file. In other words, duplicating a data descriptor does not duplicate the data contained in the original data object. It simply creates a second data object by redundantly pointing to the same data location in the file.

The tag and reference number for the duplicate data descriptor are assigned when **Hdupdd** is called. If the specified tag and reference number already exist in the file, the original data descriptor is not duplicated and **Hdupdd** returns an error.

FORTRAN None

Hendaccess

```
intn Hendaccess(int32 H_id)
```

H_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**

Purpose Terminates access to a data object by disposing of the access identifier.

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

Description The number of active access identifiers is limited to MAX_ACC as defined in the hdf header file. Because of this restriction, it is very important to call **Hendaccess** immediately following the last operation on a data element.

When developing new interfaces, a fairly common mistake is to omit calling **Hendaccess** for all of the elements accessed. When this happens, **Hclose** will return FAIL, and a dump of the error stack will report how many access identifiers are still active (see **HEprint**).

This is a rather difficult problem to debug, because the low levels of the HDF library cannot determine who and where an access identifier was originated. As a result, there is no automated way to determine which access identifiers have yet to be released.

FORTRAN None

Hfidinquire

```
intn Hfidinquire(int32 file_id, char **filename, intn *access, intn *attach)
```

file_id	IN:	HDF file identifier returned by **Hopen**
filename	OUT:	Complete path and filename for the HDF file
access	OUT:	Access mode file is opened with
attach	OUT:	Number of access identifiers attached to the file

Purpose Inquires about a file by using its file identifier.

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

Description Gets the complete path name, access mode, and number of access identifiers associated with a file. Files are identified by passing **Hfidinquire** a *file_id*. The *filename* parameter is a pointer to a character pointer which will be modified when the function returns. Upon completion, *filename* is set to point to the file name in internal storage. All output parameters must be non-`NULL` pointers.

FORTRAN None

Hfind

```
intn Hfind(int32 file_id, uint16 search_tag, uint16 search_ref, uint16 *find_tag,
          uint16 *find_ref, int32 *find_offset, int32 *find_length, intn direction)
```

file_id	IN:	HDF file identifier returned by **Hopen**
search_tag	IN:	The tag to search for or DFTAG_WILDCARD
search_ref	IN:	Reference number to search for or DFREF_WILDCARD
find_tag	IN:	If (**find_tag*==0) and (**find_ref*==0) then start search
	OUT:	Tag matching the *search_tag*
find_ref	IN:	If (**find_tag*==0) and (**find_ref*==0) then start search
	OUT:	Reference number matching the *search_ref*
find_offset	OUT:	Offset of the data element found
find_length	OUT:	Length of the data element found
direction	IN:	Direction to search in - DF_FORWARD searches forward from the current location, and DF_BACKWARD searches backward from the current location

Purpose — Locates the next object of a search in an HDF file

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

Description — **Hfind** searches for the next data element that matches the specified tag and reference number. Wildcards apply. If direction is DF_FORWARD, search from current position forwards in the file, otherwise DF_BACKWARD searches backward from the current position in the file.

If *find_tag* and *find_ref* are both set to 0, this indicates the beginning of a search, and the search will start from the beginning of the file if the direction is DF_FORWARD and from the end of the file if the direction is DF_BACKWARD.

FORTRAN — None

Hgetelement

Hgetelement

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of the data element to read
ref	IN:	Reference number of the data element to read
data	OUT:	Buffer to read into

Purpose — Obtains the data referred to by the tag and reference number that is passed to it.

Return value — Returns the number of bytes read if successful and `FAIL(-1)` otherwise.

Description — Reads the data element for the specified tag and reference number and puts it into the buffer pointed to by *data*. It is assumed that the space allocated for the buffer is large enough to hold the data.

FORTRAN — None

Hgetfileversion

```
intn Hgetfileversion(int32 file_id, uint32 *major_v, uint32 *minor_v, uint32 *release,
          char string[])
```

file_id	IN:	HDF file identifier returned by **Hopen**
major_v	OUT:	Major version number
minor_v	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose — Gets version information for an HDF file.

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

Description — Returns the HDF version number stored in the given file. It is still an open question as to what exactly the version number of a file should mean, so we recommend that code not depend on this buffer. The *string* parameter is limited to a length of `LIBVSTR_LEN` characters as defined in `hfile.h`.

FORTRAN — None

Hgetlibversion

Hgetlibversion

```
intn Hgetlibversion(uint32 *major_v, uint32 *minor_v, uint32 *release, char string[])
```

major_v	OUT:	Major version number
minor_v	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose
Gets version information for current HDF library.

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

Description
Returns the version of the HDF library. The version information is statistically compiled into the HDF library, so it is not necessary to have any open files for this function to execute. The *string* buffer is limited to a length of `LIBVSTR_LEN` characters as defined in `hfile.h`.

FORTRAN
None

Hinquire

```
intn Hinquire(int32 H_id, int32 *file_id, uint16 *tag, uint16 *ref, int32 *length,
          int32 *offset, int32 *position, int16 *access, int16 *special)
```

H_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
file_id	OUT:	HDF file identifier returned by **Hopen**
tag	OUT:	Tag of the element pointed to
ref	OUT:	Reference number of the element pointed to
length	OUT:	Length of the element pointed to
offset	OUT:	Offset of the element in the file
position	OUT:	Current position within the data element
access	OUT:	The access type for this data element
special	OUT:	Special code

Purpose — Returns access information about a data element.

Return value — Returns `SUCCEED(0)` if the access identifier points to a valid data element and `FAIL(-1)` otherwise.

Description — Inquires about the statistics of the data object specified by *H_id*. If *H_id* is a valid access identifier the access type (read or write) is set regardless of whether or not the return value is `FAIL (-1)`. If *H_id* is invalid, the function returns `FAIL (-1)` and the access type is set to zero. To avoid excess information, pass `NULL` for any unnecessary pointer.

FORTRAN — None

Hishdf

```
intn Hishdf(char *filename)
```

filename	IN:	Complete path and filename for the file to check

Purpose — Determines if a file is an HDF file.

Return value — Returns `TRUE(1)` if the file is an HDF file and `FALSE(0)` otherwise.

Description — The decision of where a file is and if a file is and HDF file is based solely on the value stored in the first four bytes of an HDF file. It is possible that **`Hishdf`** will identify a file as an HDF file but **`Hopen`** will be unable to open the file (for example if the data descriptor list is corrupt).

FORTRAN — None

HLcreate

```
int32 HLcreate(int32 file_id, uint16 tag, uint16 ref, int32 block_len, int32 n_blocks)
```

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of new data descriptor
ref	IN:	Reference number of new data descriptor
block_len	IN:	Length of blocks to be used
n_blocks	IN:	Number of blocks to use per linked block record

Purpose

Creates a linked block special data element.

Return value

Returns an access identifier if successful and FAIL(-1) otherwise.

Description

Appending data to existing data objects is a problem in early versions of the HDF library because objects are required to be stored contiguously. When appending, the HDF library deleted the existing element and moved it to the end of the file. HDF 3.2 added the concept of linked block elements which allow unlimited appending without deleting or copying existing data.

HLcreate creates a new linked block element if given an unused tag and reference number. If an existing tag and reference number are used, **HLcreate** promotes the existing data element to a linked block element. To create a linked data element, specify the number of data blocks for the element in *n_blocks* and their length in *block_len*. All data blocks must be the same length unless the element was promoted from a non-linked block data element. In this case, all but the original data block must conform to the length specification.

HLcreate creates linked block elements by inserting a linked block structure between a data descriptor and its associated data element. The linked block structure is used to keep track of the different blocks of data appended to the same data object. All data blocks associated with a new linked block element are stored at the same location. If a linked block element is promoted from a standard data element, the block containing the original data remains in its original position with the remaining data blocks stored together at a different file location. This non-contiguous configuration is known to cause problems with **Hoffset** and **Hlength** which assume that data in a data element is stored contiguously.

Promoting existing data elements to linked block elements will not automatically close opened access identifiers. Open access identifiers prevent a file from closing, therefore it is important to use **Hendaccess** to remove any open identifiers before attempting to close a file.

FORTRAN

None

Hlength

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of the data element
ref	IN:	Reference number of the data element

Purpose — Returns the length of a data object specified by the tag and reference number.

Return value — Returns the length of data element if found and `FAIL (-1)` otherwise.

Description — **Hlength** calls **Hstartread**, **Hinquire**, and **Hendaccess** to determine the length of a data element. **Hlength** uses **Hstartread** to obtain an access identifier for the specified data object. If an access identifier already exists (i.e. **Hstartread** was called earlier in the program), it is more efficient to call **Hinquire** directly.

Hlength will return the correct data length for linked block elements, however it is important to remember that the data in linked block elements is not stored contiguously.

FORTRAN — None

Hnewref

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

file_id	IN:	HDF file identifier returned by **Hopen**

Purpose	Returns a reference number that is guaranteed to be unique to the file.
Return value	Returns the reference number if successful and 0 otherwise.
Description	Returns a reference number that can be used with any tag to produce a unique tag and reference number pair. Successive calls to **Hnewref** will generate a strictly increasing sequence until the highest possible reference number has been returned, then **Hnewref** will return unused reference numbers starting from 1.
FORTRAN	None

Hnextread

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

H_id	IN:	Access identifier returned by **Hstartread** or previous **Hnextread**
tag	IN:	Tag to search for
ref	IN:	Reference number to search for
origin	IN:	Position to begin search: DF_START or DF_CURRENT

Purpose

Locates and positions a read access identifier on the next data element with the specified tag and reference number.

Return value

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

Description

Searches for the next data descriptor that matches specified tag and reference number. Wildcards apply. If *origin* is DF_START, the search will start at the beginning of the data descriptor list. If *origin* is DF_CURRENT, the search will begin at the current position. Searching from the end of a file is not yet implemented.

If the search is successful, the access identifier reflects the new data element, otherwise it is not modified.

FORTRAN

None

hnumber

Hnumber

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag to be counted

Purpose

Returns the number of instances of a tag in a file.

Return value

Returns the number of instances of a tag in a file if successful, and FAIL(-1) otherwise.

Description

Hnumber determines how many objects with the specified tag are in a file. To count the total number of objects in a file, set the *tag* argument to DFTAG_WILDCARD. Note that a return value of zero is not a fail condition.

FORTRAN

```
integer function hnumber(file_id, tag)
    integer*4 file_id
    integer*2 tag
```

Hoffset

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of the data element
ref	IN:	Reference number of the data element

Purpose — Gets the offset for a data element in the file.

Return value — Returns the offset of data element if the data element exists and `FAIL(-1)` otherwise.

Description — **Hoffset** calls **Hstartread**, **Hinquire**, and **Hendaccess** to determine the length of a data element. **Hoffset** uses **Hstartread** to obtain an access identifier for the specified data object. If an access identifier already exists (i.e. **Hstartread** was called earlier in the program), it is more efficient to call **Hinquire** directly.

Hoffset will return the correct offset for a linked block element, however it is important to remember that the data in linked block elements is not stored contiguously. The offset returned by **Hoffset** only reflects the position of the first data block.

Hoffset will also cause problems if used to determine the offset of an external element. In this case, **Hoffset** returns zero, an invalid offset for all HDF files.

FORTRAN — None

Hopen

```
int32 Hopen(char *filename, intn access, int16 n_dds)
```

filename	IN:	Complete path and filename for the file to be opened
access	IN:	DFACC_READ, DFACC_RDWR, or DFACC_CREATE
n_dds	IN:	Number of data descriptors in a block if a new file is to be created

Purpose

Provides an access path to an HDF file by reading all the data descriptor blocks into memory.

Return value

Returns the file identifier if successful and FAIL(-1) otherwise.

Description

If given a new file name, **Hopen** will create a new file using the specified access type and number of data descriptors. If given an existing file name, **Hopen** will open the file using the specified access type and ignore the *n_dds* argument.

HDF provides several constants for use as access codes:

Recommended:

DFACC_READ	Open for read only. If file does not exist, error
DFACC_RDWR	Open for read/write. If file does not exist, create it
DFACC_CREATE	If file exists, delete it, then open a new file for read/write

Others:

DFACC_ALL	Same as DFACC_RDWR
DFACC_WRITE	Same as DFACC_RDWR

If a file is opened and an attempt is made to reopen the file using DFACC_CREATE, HDF will issue the error: DFE_ALROPEN. If the file is opened with read only access and an attempt is made to reopen the file for write access using DFACC_RDWR, DFACC_WRITE, or DFACC_ALL, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the named file is opened with the relevant permissions, the data descriptors are set up in memory, and the associated *file_id* is returned. For new files, the appropriate file headers are also set up.

FORTRAN

```
integer function hopen(filename, access, n_dds)
   character*(*)  filename
   integer        access
   integer*2      n_dds
```

Hputelement

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag of the data element to add or replace
ref	IN:	Reference number of the data element to add or replace
data	IN:	Pointer to data buffer
length	IN:	Length of data to write

Purpose — Adds or replaces a data element in an HDF file.

Return value — Returns the number of bytes written if successful and `FAIL(-1)` otherwise.

Description — Writes a data element or replaces an existing data element in a HDF file.

FORTRAN — None

Hread

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

H_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
length	IN:	Length of segment to read in
data	OUT:	Pointer to the data array to read to

Purpose — Reads the next segment in a data element.

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

Description — **Hread** begins reading at the current position of the access identifier, reads the specified number of bytes, and moves the access identifier to the position immediately following the last accessed byte. Calling **Hread** with the `length=0` reads the entire data element. To reposition an access identifier before writing data, use **Hseek**.

If *length* is longer than the data element, the read operation is terminated at the end of the data element, and the number of read bytes is returned. Although only one access identifier is allowed per data element, it is possible to interlace reads from multiple data elements in the same file. It is assumed that *data* is large enough to hold the specified data length.

FORTRAN — None

Hseek

```
intn Hseek(int32 H_id, int32 offset, intn origin)
```

H_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
offset	IN:	Number of bytes to seek to from the origin
origin	IN:	Position to add the *offset* to; DF_START or DF_CURRENT

Purpose Sets the access pointer to an offset within a data element.

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

Description Sets the seek position for the next **Hread** or **Hwrite** operation by moving an access identifier to the specified position in a data element. The *origin* argument, together with the *offset* argument, determine the byte location for the access identifier. If the *origin* is DF_START, the *offset* is added to the beginning of the data element. If *origin* is DF_CURRENT, the *offset* is added to the current position of the access identifier.

This routine fails if the access identifier if *H_id* is invalid or if the seek position is outside the range of the data element.

FORTRAN None

Hstartread

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag to search for
ref	IN:	Reference number to search for

Purpose Creates and positions a read access identifier in the data element identified by the specified tag and reference number.

Return value Returns the access identifier for a data element if successful and FAIL(-1) otherwise.

Description This function begins by searching the data descriptor list for the data element with the specified tag and reference number. All searches begin at the head of the data descriptor list.

If the search is successful, an access identifier is created and positioned at the beginning of the data element, otherwise an error is returned. Wildcards can be used for any *tag* or *ref* (DFTAG_WILDCARD, DFREF_WILDCARD). All access identifiers must be released through **Hendaccess** before closing the HDF file.

FORTRAN None

Hstartwrite

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

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag to write to
ref	IN:	Reference number to write to
length	IN:	The length of the data element

Purpose Positions a write access identifier at the beginning of a new or existing data element.

Return value Returns the access identifier for a data object if successful and `FAIL(-1)` otherwise.

Description **Hstartwrite** begins by searching the data descriptor list for the specified *tag* and *ref*. If the tag and reference number exist, **Hstartwrite** positions an access identifier at the beginning of the existing data element. Data in the existing data element is not overwritten until a call is made to **Hwrite**. If the tag and reference number are not found, **Hstartwrite** allocates space for a new data object of length *length* and positions an access identifier at the head of the data element.

FORTRAN None

Hsync

```
intn Hsync(int32 file_id)
```

file_id	IN:	HDF file identifier returned by **Hopen**

Purpose Synchronizes on-disk HDF file with the memory representation.

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

Description This routine is currently vacuous as the on-disk representation of an HDF file is always the same as its in-memory representation. However, future releases of the HDF library may employ buffering schemes, so this might not always be the case. **Hsync** will be provided to force the two representations to become consistent.

FORTRAN None

Htrunc

```
int32 Htrunc(int32 H_id, int32 trunc_len)
```

H_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
trunc_len	IN:	Length to truncate element

Purpose	Truncates a data element to a specified length
Return value	Returns the length of a data element if found and `FAIL(-1)` otherwise.
Description	Truncates the data object specified by the *H_id* to the length *trunc_len*. This routine is used primarily for resizing data elements containing compressed data.
FORTRAN	None

Hwrite

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

H_id	IN:	Access identifier returned by **Hstartwrite**
len	IN:	Length of segment to write
data	IN:	Pointer to the data to write

Purpose — Writes the next data segment to a specified data element.

Return value — Returns the length of the segment actually written if successful and `FAIL(-1)` otherwise.

Description — **Hwrite** begins writing at the current position of the access identifier, writes the specified number of bytes, and moves the access identifier to the position immediately following the last accessed byte. Calling **Hwrite** with the `length=0` returns an error. To reposition an access identifier before writing data, use **Hseek**.

If the space allocated in the data element is smaller than the length of *data*, the data is truncated to the length of the data element. Although only one access identifier is allowed per data element, it is possible to interlace writes to multiple data elements in the same file.

FORTRAN — None

HXcreate

```
int32 HXcreate(int32 file_id, uint16 tag, uint16 ref, char *extern_name,
          int32 offset, int32 length)
```

file_id	IN:	HDF file identifier returned by **Hopen**
tag	IN:	Tag for the external element
ref	IN:	Reference number for the external element
extern_name	IN:	Name of the external file
offset	IN:	Byte offset for the external element in the external file
length	IN:	Length of the external element

Purpose Creates an external element.

Return value Returns the access identifier for the external element if successful and FAIL(-1) otherwise.

Description An external element is a data element stored in a different file than its data descriptor. **HXcreate** can create new external elements, move existing data elements from and HDF file to an external file, or incorporate data from an external file into an HDF file.

HXcreate creates a new external element if given an unused tag and reference number and the file name of a nonexistent file. Specify the *offset* as the number of bytes between the beginning of the external file and the beginning of the external element. After creating the new external element, use **Hwrite** to write data to it.

HXcreate moves a data element from an HDF file to an external file if given an existing tag and reference number. During this operation, the original data element is deleted from the HDF file and copied into the external file. Any data located at the specified offset in the external file is overwritten. Specify the *offset* as if creating a new external element. The *length* argument is ignored.

HXcreate incorporates data from an external file into an HDF file if given and new tag and reference number and the file name of an existing external file. This operation does not modify the external file, it simply creates a data descriptor in the specified HDF file that points to the external data. The *offset* and *length* should reflect the location and length of the data in the external file.

Moving existing data elements to external files will not automatically close open access identifiers. Open access identifiers prevent a file from closing, therefore it is important to use **Hendaccess** to remove any open identifiers before attempting to close a file.

Distributing a single data element over multiple external files is not supported.

FORTRAN None

HDFclose

```
intn HDFclose(int32 file_id)
```

file_id	IN:	The HDF file identifier returned by **Hopen**

Purpose — Closes the access path to the file.

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

Description — The file identifier *file_id* is validated before the file is closed. If the identifier is valid, the function closes the access path to the file.

If there are still access identifiers attached to the file, the error `DFE_OPENAID` is returned and the file is not closed. This is a fairly common error when developing new interfaces. See **Hendaccess** for discussion of this problem.

FORTRAN

```
integer function hdfclose(file_id)
    integer*4 file_id
```

HDFopen

hdfopen

HDFopen

```
int32 HDFopen(char *filename, intn access, int16 n_dds)
```

filename	IN:	Complete path and filename for the file to be opened
access	IN:	DFACC_READ, DFACC_WRITE, or DFACC_CREATE
n_dds	IN:	Number of data descriptors in a block if a new file is to be created

Purpose

Provides an access path to an HDF file by reading all the data descriptor blocks into memory.

Return value

Returns the file identifier if successful and FAIL(-1) otherwise.

Description

If given a new file name, **HDFopen** will create a new file using the specified access type and number of data descriptors. If given an existing file name, **HDFopen** will open the file using the specified access type and ignore the *n_dds* argument.

HDF provides several constants for use as access codes:

Recommended:

DFACC_READ	Open for read only. If file does not exist, error
DFACC_RDWR	Open for read/write. If file does not exist, create it
DFACC_CREATE	If file exists, delete it, then open a new file for read/write

Others:

DFACC_ALL	Same as DFACC_RDWR
DFACC_WRITE	Same as DFACC_RDWR

If a file is opened and an attempt is made to reopen the file using DFACC_CREATE, HDF will issue the error: DFE_ALROPEN. If the file is opened with read only access and an attempt is made to reopen the file for write access using DFACC_RDWR, DFACC_WRITE, or DFACC_ALL, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the named file is opened with the relevant permissions, the data descriptors are set up in memory, and the associated *file_id* is returned. For new files, the appropriate file headers are also set up.

FORTRAN

```
integer function hdfopen(filename, access, n_dds)
   character*(*)  filename
   integer        access
   integer*2      n_dds
```

HEclear

```
VOID HEclear(void)
```

Purpose	Clears all information on reported errors off of the error stack.
Return value	None
Description	**`HEpush`** creates an error stack. **`HEclear`** is then used to clear this stack after any errors are processed.
FORTRAN	None

heprnt

HEprint

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

stream	IN:	Stream to print error message to
level	IN:	Level of error stack to print

Purpose — Prints information on the error stack.

Return value — None

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

The following information is printed:

1. ASCII description of the error
2. reporting routine
3. reporting routine as source file name
4. line at which the error was reported

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

The FORTRAN routine uses one less parameter than the C routine because it doesn't allow the user to specify the print stream. Instead it prints automatically to `stdout`.

FORTRAN

```
subroutine heprnt(level)
   integer*4 level
```

HEpush

```
VOID HEpush(int16 error_code, char *funct_name, char *file_name, intn line)
```

error_code	IN:	Internal number of the error
funct_name	IN:	Name of function in which the error occurred
file_name	IN:	Name of file in which the error occurred
line	IN:	Line number in the file that error occurred

Purpose — Pushes a new error onto the error stack.

Return value — None

Description — **HEpush** pushes the file name, function name, line number, and generic description of the error onto the error stack. **HEreport** can then be used to give a more case-specific description of the error.

If the stack is full, the error is ignored.**HEpush** assumes that the character strings *funct_name* and *file_name* are in semi-permanent storage, so only pointers to the strings are saved.

FORTRAN — None

heprnt

HEreport

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

format IN: `Printf` compatible output string

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

Return value None

Description By default, an error report made with **`HEpush`** places on the error stack the file name, function name, line number, and a generic description of the error type. `HEreport` can then be used to give a more case-specific description of the error. Only one additional annotation can be attached to each error report.

The arguments to this routine follow the style of `printf`.

Example

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

FORTRAN None

HEstring

```
char *HEstring(int16 error_code)
```

error_code	IN:	HDF error code

Purpose	Returns the error message associated with error code.
Return value	Returns a pointer to a string associated with the error code if successful.
Description	Returns a text description for the given error code. These strings are statically declared and should not be `free()ed` by the user. If no string can be found to describe an error code, a generic default message is returned.
FORTRAN	None

sfgainfo

SDattrinfo

```
intn SDattrinfo(int32 s_id, int32 attr_index, char *attr_name, int32 *number_type,
          int32 *count)
```

s_id	IN:	An *sd_id* for a global attribute; *sds_id* for a local attribute
attr_index	IN:	Index of the attribute to read
attr_name	OUT:	Space to hold the name assigned to the attribute/data set/dimension
number_type	OUT:	Space to hold the number type for the attribute values
count	OUT:	Space to hold the total number of values in the specified attribute

Purpose — Gets information about a global or local attribute.

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

Description — This routine should be used before reading the value of an attribute with **`SDreadattr`**. **`SDreadattr`** requires a buffer to hold the values of the attribute. The size of this buffer should be at least `count*DFKNTsize(number_type)` bytes long.

FORTRAN

```
integer function sfgainfo(s_id, attr_index, attr_name, number_type,
count)
    character*(*)  attr_name
    integer*4      s_id, attr_index, number_type, count
```

SDcreate

sfcreate

SDcreate

```
int32 SDcreate(int32 sd_id, char *name, int32 number_type, int32 rank,
          int32 dimsizes[])
```

sd_id	IN:	The SD interface ID returned from **SDstart**
name	IN:	ASCII string defining a variable name
number_type	IN:	Number type for the values in the data set
rank	IN:	The number of dimensions in the data set
dimsizes	IN:	The size of each dimension

Purpose — Creates a new data set.

Return value — Returns the *sds_id* if successful and FAIL(-1) otherwise.

Description — If name is NULL a fake name will be generated. The name will be truncated to the MAX_NC_NAME length as specified in the HDF header files.

Once a data set has been created, it is impossible to change its name, number type, or rank. However, it is possible to create a data set and close the file before writing any data values to it. The values can be added or modified at a future time. If you wish to add data or modify an existing data set, use **SDselect** to get the *sds_id*.

In C, if *dimsizes*[0] is assigned the value SD_UNLIMITED then this dimension is considered to be an "unlimited dimension" and the user can append data along this dimension at will. In Fortran, *dimsizes*(*rank*) is the only dimension that can be unlimited. This is useful when the eventual size of a data set is not known at creation time.

MAX_VAR_DIMS is the maximum rank a data set can have.

FORTRAN

```
integer function sfcreate(sd_id, name, number_type, rank, dimsizes)
    character*(*)  name
    integer*4      sd_id, number_type, rank, dimsizes(*)
```

sfgdinfo

SDdiminfo

```
intn SDdiminfo(int32 dim_id, char *name, int32 *count, int32 *number_type,
          int32 *nattrs)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
name	OUT:	Array to retrieve dimension name
count	OUT:	Space allocated to store the size of the dimension
number_type	OUT:	Space allocated to store the number type for the data stored in the data set
nattrs	OUT:	Space for attribute count assigned to the dimension's coordinate variable

Purpose — Gets information about a dimension.

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

Description — If scale information has been stored for this dimension via **SDsetdimscale**, *number_type* will contain the number type otherwise the *number_type* will be `0`. If the user has not named the dimension via **SDsetdimname** a place holder will be generated. If the name is unimportant, `NULL` can be passed in for the *name* parameter.

FORTRAN

```
integer function sfgdinfo(dim_id, name, count, number_type,
nattrs)
    character*(*)  name
    integer*4      dim_id, count, number_type, nattrs
```

SDend

sfend

SDend

```
intn SDend(int32 sd_id)
```

sd_id	IN:	The SD interface ID returned from **SDstart**

Purpose — Closes the file and cleans up memory.

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

Description — **SDend** closes the file when done with SD interface activities. If a user program exits without calling this function, changes made to the file are likely to get lost.

FORTRAN

```
integer function sfend(sd_id)
    integer*4 sd_id
```

sfendacc

SDendaccess

```
intn SDendaccess(int32 sds_id)
```

sds_id	IN:	The data set ID returned from **SDselect**

Purpose — Disposes of a data set ID.

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

Description — When done interacting with a specific data set, this routine should be called to release the internal data structures. This routine should be called once for each call to **SDcreate** or **SDselect**. Failing to call this function may result in loss of data.

FORTRAN

```
integer function sfendacc(sds_id)
    integer*4 sds_id
```

SDfileinfo

sffinfo

SDfileinfo

```
intn SDfileinfo(int32 sd_id, int32 *ndatasets, int32 *nglobal_attr)
```

sd_id	IN:	The SD interface ID returned from **SDstart**
ndatasets	OUT:	Number of data sets in the file
nglobal_attr	OUT:	Number of global attributes in the file

Purpose
Determines the number of data sets and global attributes in a file.

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

Description
SDfileinfo returns the number of currently defined data sets and global attributes in the file. “Global attributes” refer to attributes assigned to the file as opposed to “local attributes” which are assigned to specific data sets. For example, `file_creator_date` would be a global attribute, while `units` would typically be data set specific. Global attributes are created by **SDsetattr** using the *sd_id* rather than an *sds_id*.

FORTRAN

```
integer function sffinfo(sds_id, ndatasets, nglobal_attr)
    integer*4 sds_id, ndatasets, nglobal_attr
```

SDfindattr

```
int32 SDfindattr(int32 s_id, char *attr_name)
```

s_id	IN:	An *sd_id* for global attribute; *sds_id* for a local attribute
attr_name	IN:	Name assigned to the attribute

Purpose — Finds the index for an attribute with a given name.

Return value — Returns the *attr_index* if successful and FAIL(-1) otherwise.

Description — The *attr_index* returned by this function can be passed to **SDattrinfo** and **SDreadattr**. This routine is case sensitive. Wild cards are not allowed in the look up string.

FORTRAN

```
integer function sffattr(s_id, attr_name)
    integer*4       s_id
    character*(*)   attr_name
```

SDgetcal

sfgcal

SDgetcal

```
intn SDgetcal(int32 sds_id, float64 *cal, float64 *cal_err, float64 *offset,
          float64 *offset_err, int32 *number_type)
```

sds_id	IN:	The data set ID returned from **SDselect**
cal	IN:	Calibration factor
cal_err	IN:	Calibration error
offset	IN:	Uncalibrated offset
offset_err	IN:	Uncalibrated offset error
number_type	IN:	Number type of uncalibrated data

Purpose — Gets the calibration information associated with the specified data set.

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

Description — **SDgetcal** reads the calibration record attached to a data set. A calibration record contains four 64-bit floating point values followed by a 32-bit integer, to be interpreted as follows:

cal	calibration factor
cal_err	calibration error
offset	uncalibrated offset
offset_err	uncalibrated offset error
number_type	number type of uncalibrated data

The relationship between a value 'iy' stored in a data set and the actual value is defined as:

y = cal * (iy - offset)

The variable *offset_err* contains a potential error of *offset*, and *cal_err* contains a potential error of *cal*. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

FORTRAN

```
integer function sfgcal(sds_id, cal, cal_err, offset, offset_err,
number_type)
    integer*4       sds_id
    real*8          cal, cal_err, offset, offset_err
    integer*4       number_type
```

SDgetdatastrs

```
intn SDgetdatastrs(int32 sds_id, char *label, char *unit, char *format,
          char *coordsys, intn len)
```

sds_id	IN:	The data set ID returned from **SDselect**
label	OUT:	Label that describes the data
unit	OUT:	Unit to be used with the data
format	OUT:	Format to be used in displaying data
coordsys	OUT:	Coordinate system to be used with the data
len	IN:	Maximum length string it is safe to return

Purpose — Returns the data strings associated with the specified data set.

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

Description — **SDgetdatastrs** returns the data strings stored by **SDsetdatastrs**. If a particular string was not stored the first character of the return string will be `'\0'`. Each string buffer is assumed to be at least *len* characters long including the space to hold the `'\0'` string termination character. If a user does not want a string back, `NULL` can be passed for any of the string values.

FORTRAN

```
integer function sfgdtstr(sds_id, label, unit, format,
coordsys, len)
    integer*4       sds_id
    character*(*)   label, unit, format, coordsys
    integer         len
```

SDgetdimid

sfdimid

SDgetdimid

```
int32 SDgetdimid(int32 sds_id, intn dim_number)
```

sds_id	IN:	The data set ID returned from **SDselect**
dim_number	IN:	Number of dimension to get a pointer to

Purpose — Gets the dimension ID associated with the specified dimension in the data set.

Return value — Returns a *dim_id* if successful and `FAIL(-1)` otherwise.

Description — The *dim_id* is required by all other calls that will deal with this particular dimension. Dimensions are zero based so: `0 <= dim_number < rank`.

FORTRAN

```
integer function sfdimid(sds_id, dim_number)
    integer*4 sds_id, dim_number
```

SDgetdimscale

```
intn SDgetdimscale(int32 dim_id, VOIDP data)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
data	OUT:	Buffer allocated to hold the scale values for the specified dimension

Purpose — Retrieves the scale values for the specified dimension.

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

Description — It is assumed that the user has already called **SDdiminfo** and has thus allocated sufficient space to hold the values. It is not possible to read only some of the scale values in a hyperslab fashion. **SDgetdimscale** returns all of the scale values stored with the given dimension.

FORTRAN

```
integer function sfgdscale(dim_id, data)
    integer*4       dim_id
    <valid type>    (*)
```

SDgetdimstrs

sfgdmstr

SDgetdimstrs

```
intn SDgetdimstrs(int32 dim_id, char *label, char *unit, char *format, intn len)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
label	OUT:	Label that describes this dimension
unit	OUT:	Unit to be used with this dimension
format	OUT:	Format to be used in displaying scale for this dimension
len	IN:	Maximum string length it is safe to return

Purpose — Gets the label, unit, and format strings for a given dimension.

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

Description — This routine returns the data strings stored by **SDsetdimstrs**. If a particular string was not stored, the first character of the return string will be `'\0'`. Each string buffer is assumed to be at least *len* characters long. If a user does not want a string back, `NULL` can be passed for any of the string values.

FORTRAN

```
integer function sfgdmstr(dim_id, label, unit, format, len)
   integer*4      dim_id
   character*(*)  label, unit, format
   integer        len
```

sfgfill

SDgetfillvalue

```
intn SDgetfillvalue(int32 sds_id, VOIDP fill_val)
```

sds_id	IN:	The data set ID returned from **SDselect**
fill_val	OUT:	Space allocated to store the fill value

Purpose — Reads the fill value for the given data set if it exists.

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

Description — It is assumed that the number type of the fill value is the same as for the data set.

FORTRAN

```
integer function sfgfill(sds_id, fill_val)
    integer*4       sds_id
    <valid type>    fill_val
```

SDgetinfo

sfginfo

SDgetinfo

```
intn SDgetinfo(int32 sds_id, char *sds_name, int32 *rank, int32 dimsizes[],
          int32 *number_type, int32 *nattrs)
```

sds_id	IN:	The data set ID returned from **SDselect**
sds_name	OUT:	Space to store the name (if any) of the data set
rank	OUT:	Space to store the number of dimensions in the data set
dimsizes	OUT:	Space to store the size of each dimension in the data set
number_type	OUT:	Space to store the number type for the data stored in the data set
nattrs	OUT:	Space to store the number of "netCDF-style" attributes for this data set

Purpose — Gets the name, rank, dimension sizes, number type and attribute count for the specified data set.

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

Description — Returns basic information about a given data set. All fields must be provided. The data set name can be at most `MAX_NC_NAME` characters long and the rank of the data set is limited to `MAX_VAR_DIMS`.

FORTRAN

```
integer function sfginfo(sds_id, sds_name, rank, dimsizes,
number_type, nattrs)
    character*(*)  sds_name
    integer*4      sds_id, rank, dimsizes(*)
    integer*4      number_type, nattrs
```

SDgetrange

```
intn SDgetrange(int32 sds_id, VOIDP max, VOIDP min)
```

sds_id	IN:	The data set ID returned from **SDselect**
max	OUT:	High value in range
min	OUT:	Low value in range

Purpose — Reads the maximum and minimum values as they are stored with the scientific data set.

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

Description — The *max* and *min* must be set via a call to **SDsetrange**. They are not automatically stored when a data set is written to a file.

Since the *max* and *min* values are supposed to relate to the data itself, it is assumed that they are of the same number type as the data itself. One implication of this is that in the C version of **SDgetrange** the arguments are pointers, rather than simple variables, whereas in the FORTRAN version they are simple variables of the same number type as the data array.

FORTRAN

```
integer function sfgrange(sds_id, max, min)
    integer*4       sds_id
    <valid type>    max
    <valid type>    min
```

SDidtoref

sfid2ref

SDidtoref

```
int32 SDidtoref(int32 sds_id)
```

sds_id	IN:	The data set ID returned from **SDselect**

Purpose — Uses a data set ID to determine the reference number assigned to the specified data set.

Return value — Returns *sds_ref* if successful and FAIL(-1) otherwise.

Description — This reference number can be used to add the data set to a Vgroup as well as a means of using the HDF annotations interface to annotate the data set.

FORTRAN

```
integer function sfid2ref(sds_id)
    integer*4 sds_id
```

SDiscoordvar

```
intn SDiscoordvar(int32 sds_id)
```

sds_id	IN:	The data set ID returned from **SDselect**

Purpose — Determines if a data set is a coordinate variable

Return value — Returns TRUE if the data set is a coordinate variable, and FALSE otherwise.

Description — This routine can be used to tell whether a given data set is a "coordinate variable" or not. Coordinate variables are created to store metadata associated with dimensions. Due to netCDF compatibility, coordinate variables are also considered data sets which sometimes is a source of confusion.

FORTRAN

```
integer function sfiscvar(sds_id)
    integer*4 sds_id
```

SDnametoindex

sfn2index

SDnametoindex

```
int32 SDnametoindex(int32 sd_id, char *sds_name)
```

sd_id	IN:	The SD interface ID returned from **SDstart**
sds_name	IN:	Name of the data set to index

Purpose — Uses the name of a scientific data set to determine the index assigned to it.

Return value — Returns an *sds_index* if successful and FAIL(-1) otherwise.

Description — The *sds_index* can be passed to **SDselect** to return an *sds_id* for the named data set. This routine is case sensitive and will not accept wild cards. In addition, this routine will only return the *sds_index* for the first data set with the given name.

FORTRAN

```
integer function sfn2index(sd_id, sds_name)
   integer*4      sd_id
   character*(*)  sds_name
```

SDreadattr

```
intn SDreadattr(int32 s_id, int32 attr_index, VOIDP data)
```

s_id	IN:	*sd_id* if reading a global attribute; an *sds_id* if reading a local attribute
attr_index	IN:	Index of the attribute to read
data	OUT:	Buffer allocated to hold attribute values

Purpose — Reads the values of an attribute.

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

Description — It's assumed that the user has called **`SDattrinfo`** and that the buffer is large enough to handle the data. If an attribute has multiple values stored for it, this routine will return all of them. It is not possible to read a subset of attribute values.

FORTRAN

```
integer function sfrattr(s_id, attr_index, data)
    integer*4       s_id
    integer*4       attr_index
    <valid type>    data(*)
```

SDreaddata

sfrdata

SDreaddata

```
intn SDreaddata(int32 sds_id, int32 start[], int32 stride[], int32 edge[],
          VOIDP buffer)
```

sds_id	IN:	The data set ID returned from **SDselect**
start	IN:	Array specifying the starting location
stride	IN:	Array specifying the number of values to skip along each dimension
edge	IN:	Array specifying the number of values to read along each dimension
buffer	OUT:	Buffer to store the data

Purpose
Reads a hyperslab of data from a data set.

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

Description
The array *start* gives the multi-dimensional index of the starting corner of the hyperslab to read. The values are zero based.

The array *edge* gives the number of values to read along each dimension of the hyperslab.

The array *stride* allows for sub-sampling along each dimension. If a stride value is specified for a dimension, that many values will be skipped over when reading along that dimension. Specifying *stride*=NULL reads contiguous data. No matter what stride value is provided, data is always placed contiguously in buffer.

FORTRAN

```
integer function sfrdata(sds_id, start, stride, edge, buffer)
   integer*4      sds_id, start(*), stride(*), edge(*)
   <valid type>   buffer(*)
```

sfref2index

SDreftoindex

```
int32 SDreftoindex(int32 sd_id, int32 sds_ref)
```

sd_id	IN:	The SD interface ID returned from **SDstart**
sds_ref	IN:	Reference number for the specified data set

Purpose — Uses the reference number to determine the index assigned to a data set.

Return value — Returns an *sds_index* if successful and FAIL(-1) otherwise.

Description — This *sds_index* can be passed to **SDselect** to return a data set ID (*sds_id*).

FORTRAN

```
integer function sfref2index(sd_id, sds_ref)
    integer*4 sd_id, sds_ref
```

SDselect

sfselect

SDselect

```
int32 SDselect(int32 sd_id, int32 index)
```

sd_id	IN:	The SD interface ID returned from **SDstart**
sds_index	IN:	The position of the data set in the file

Purpose — Gets the sds_id for a specific data set.

Return value — Returns the *sds_id* if successful and `FAIL(-1)` otherwise.

Description — To get an *sds_id* for the Nth data set, use N as the index. N must be a number greater than or equal to 0 and less than the total number of data sets in the file. The total number of data sets in a file may be obtained from a call to **SDfileinfo**. The function **SDnametoindex** can be used to find the index of a data set if its name is known.

The integration with netCDF has required that dimension metadata be stored as "coordinate variables." This fact sometimes causes problems when calling **SDselect** since the coordinate variables can lead to different data set ordering than is expected. In situations such as these, users should use the routine **SDiscoordvar** to determine if a given data set is a coordinate variable or not.

FORTRAN

```
integer function sfselect(sd_id, index)
    integer*4 sd_id, index
```

sfsattr

SDsetattr

```
intn SDsetattr(int32 s_id, char *attr_name, int32 num_type, int32 count,
          VOIDP values)
```

s_id	IN:	A *sd_id* to create a global attribute; a *sds_id* to create a local attribute
attr_name	IN:	Name to assign to the attribute
num_type	IN:	Number type for the values in the attribute
count	IN:	Total number of values to store in the attribute
values	IN:	Data values to store in the attribute

Purpose Defines a new type of attribute for the given variable.

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

Description This routine provides a generic way for users to define metadata. It implements the `label=value` data abstraction.

The *attr_name* argument can be any ASCII string for which one or more values may be stored. If more than one value is stored, all values must have the same number type. If an attribute with the given scope and name exists, it will be over written.

If an *sd_id* is given instead of an *sds_id*, a global attribute is created which applies to the whole file. Global attributes refer to all objects in a file as opposed to specific data sets. For example, `file_creator_date` is a global attribute, where as `units` is typically considered to be data set specific.

Example Store a `valid_range` attribute for a data set.

```
int32 range[2];
int32 sds_id;
int32 status;

...
sds_id = SDcreate...;
...

range[0] = 5;
range[1] = 100;
status = SDsetattr(sds_id, "valid_range", DFNT_INT32, 2, range);
```

FORTRAN

```
integer function sfsattr(s_id, attr_name, num_type, count, values)
   character*(*)  attr_name, values
   integer*4      s_id, num_type, count
   <valid type>   values(*)
```

SDsetcal

sfscal

SDsetcal

```
intn SDsetcal(int32 sds_id, float64 cal, float64 cal_err, float64 offset,
          float64 offset_err, int32 number_type)
```

sds_id	IN:	The data set ID returned from **SDselect**
cal	IN:	Calibration factor
cal_err	IN:	Calibration error
offset	IN:	Uncalibrated offset
offset_err	IN:	Uncalibrated offset error
number_type	IN:	Number type of uncalibrated data

Purpose Sets the calibration information.

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

Description **SDsetcal** stores the calibration record associated with a data set. A calibration record contains the following information:

cal	calibration factor
cal_err	calibration error
offset	uncalibrated offset
offset_err	uncalibrated offset error
number_type	number type of uncalibrated data

The relationship between a value 'iy' stored in a data set and the actual value is defined as:

y = cal * (iy - offset)

The variable *offset_err* contains a potential error of *offset*, and *cal_err* contains a potential error of *cal*. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

SDsetcal works like other **SDset*** routines, with one exception: the calibration information is automatically cleared after a call to **SDreaddata** or **SDwritedata**. Hence, **SDsetcal** must be called anew for each data set that is to be written.

FORTRAN

```
integer function sfscal(sds_id, cal, cal_err, offset, offset_err,
number_type)
    integer*4       sds_id, number_type
    real*8          cal, cal_err, offset, offset_err
```

SDsetdatastrs

```
intn SDsetdatastrs(int32 sds_id, char *label, char *unit, char *format,
          char *coordsys)
```

sds_id	IN:	The data set ID returned from **SDselect**
label	IN:	Label that describes the data
unit	IN:	Unit to be used with the data
format	IN:	Format to be used in displaying the data
coordsys	IN:	Coordinate system to be used with the data

Purpose — Sets the label, unit, format, and coordinate system strings for a given data set.

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

Description — This routine can be used to set the label, unit, format, and coordinate system strings for a given data set. `NULL` can be passed for any strings the user does not want to set.

FORTRAN

```
integer function sfsdtstr(sds_id, label, unit, format,
coordsys)
    integer*4       sds_id
    character*(*)   label, unit, format, coordsys
```

SDsetdimname

sfsdmname

SDsetdimname

```
intn SDsetdimname(int32 dim_id, char *dim_name)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
dim_name	IN:	ASCII string to name dimension

Purpose — Assigns a name to the specified dimension.

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

Description — If another dimension exists with the same name it is assumed that they refer to the same dimension object and changes to one will be reflected in the other. Naming dimensions is optional but encouraged. Dimensions that are not explicitly named by the user will have generated names used as place holders.

FORTRAN

```
integer function sfsdmname(dim_id, dim_name)
    integer*4      dim_id
    character*(*)  dim_name
```

sfsdscale

SDsetdimscale

```
intn SDsetdimscale(int32 dim_id, int32 count, int32 number_type, VOIDP data)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
count	IN:	Total number of values along the specified dimension
number_type	IN:	Number type for the values along the specified dimension
data	IN:	Value for each increment along the specified dimension

Purpose — Stores the values of the specified dimension.

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

Description — Users should note that it is possible to store values for a dimension without having to name it. Even though its redundant we have left the *count* argument in for backward compatibility. Note that there is now a *number_type* parameter so the dimension scales no longer have to be the same number type as the data set.

FORTRAN

```
integer function sfsdscale(dim_id, count, number_type, data)
    integer*4       dim_id
    integer*4       count, number_type
    <valid type>    data(*)
```

SDsetdimstrs

sfsdmstr

SDsetdimstrs

```
intn SDsetdimstrs(int32 dim_id, char *label, char *unit, char *format)
```

dim_id	IN:	Dimension ID returned from **SDgetdimid**
label	IN:	Label that describes this dimension
unit	IN:	Unit to be used with this dimension
format	IN:	Format to be used to display scale

Purpose — Sets the label, unit, and format strings for a given dimension.

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

Description — **SDsetdimstrs** sets the label, unit, and format specifications for a dimension and its scale. If `NULL` is passed for a string, that string will not be written.

FORTRAN

```
integer function sfsdmstr(dim_id, label, unit, format)
   integer*4      dim_id
   character*(*)  label, unit, format
```

SDsetexternalfile

```
intn SDsetexternalfile(int32 sds_id, char *filename, int32 offset)
```

sds_id	IN:	The data set ID returned from **SDselect**
filename	IN:	Name of the external file
offset	IN:	Byte count from the beginning of the external file to where the data starts

Purpose — Stores data in an external file.

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

Description — **SDsetexternalfile** allows users to move the data values for a data set into an external data file. Only the data set values can be stored externally, all other data must stay in the original file.

Data can only be moved once for any given data set and it is the user's responsibility to make sure the external data file is transported with the "original" file. The *offset* is the number of bytes from the beginning of the external file to where the external data begins. This routine can only be called on HDF 3.3 files (i.e. calling on a netCDF file that was opened with the multi-file interface will fail).

FORTRAN — None

SDsetfillvalue

sfsfill

SDsetfillvalue

```
intn SDsetfillvalue(int32 sds_id, VOIDP fill_val)
```

sds_id	IN:	The data set ID returned from **SDselect**
fill_val	IN:	Pointer to the fill value to be used

Purpose — Sets the fill value for the given data set.

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

Description — This routine sets the fill value to be used for the specified data set. Since data can be written in non-contiguous hyperslabs, it is possible that not all of the locations in a given data set will be written to. The *fill_val* is placed into all unwritten locations. It is assumed that the fill value has the same number type as the data set.

To avoid consistency problems, this routine should be called before any data is written to the data set. In addition, once set, the fill value should not be changed.

FORTRAN

```
integer function sfsfill(sds_id, fill_val)
   integer*4      sds_id
   <valid type>   fill_val
```

sfsrange

SDsetrange

```
intn SDsetrange(int32 sds_id, VOIDP max, VOIDP min)
```

sds_id	IN:	The data set ID returned from **SDselect**
max	IN:	High value in range
min	IN:	Low value in range

Purpose — Sets the "valid" max and min values for the given data set.

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

Description — Since *max* and *min* are supposed to relate to the data itself, it is assumed that the number type for the *max*/*min* values are the same as the number type of the data. One implication of this is that in the C version of **SDsetrange** the arguments are pointers, rather than simple variables, whereas in the FORTRAN version they are simple variables of the same type as the data array.

This routine does not compute the maximum and minimum values; it merely stores the values it is given. As a result, the maximum and minimum values may not always reflect the actual maximum and minimum values in the data array.

FORTRAN

```
integer function sfsrange(sds_id, max, min)
    integer*4       sds_id
    <valid type>    max, min
```

SDstart

sfstart

SDstart

```
int32 SDstart(char *filename, int32 access_mode)
```

filename	IN:	Name of the HDF file
access_mode	IN:	Recognized access code

Purpose Opens the HDF file and initializes the SD interface.

Return value Returns an *sd_id* if successful and FAIL (-1) otherwise.

Description This routine opens a file and returns an *sd_id*. This routine must be called for each file before any other SD calls can be made on that file. The *access_mode* is one of the following:

DFACC_RDONLY
DFACC_RDWR
DFACC_CREATE (destroy existing contents and then rewrite)

The file can be any one of the following:

XDR-based netCDF file
"old-style" DFSD file
"new-style" SD file

If *access_mode* DFACC_CREATE is used "new-style" SD files will be created.

The type of ID returned by **SDstart** is currently not the same as the ID returned by **Hopen**. As a result, *sd_id*s are not understood by other HDF interfaces and *h_id*s are not recognized by the SD interface.

To mix SD calls and other HDF library calls, use **SDstart** and **Hopen** on the same file. **SDstart** must precede all SD calls, and **Hopen** must proceed all other HDF function calls. To terminate access to the file, use both **SDend** and **Hclose** to dispose of the *sd_id* and the *h_id*.

FORTRAN

```
integer function sfstart(filename, access_mode)
    character*(*)  filename
    integer*4      access_mode
```

SDwritedata

```
intn SDwritedata(int32 sds_id, int32 start[], int32 stride[], int32 edge[],
          VOIDP data)
```

sds_id	IN:	The data set ID returned from **SDselect**
start	IN:	Array specifying the starting location
stride	IN:	Array specifying the number of values to skip along each dimension
edge	IN:	Array specifying the number of values to read along each dimension
data	IN:	Values to be written to the data set

Purpose — Writes a hyperslab of data for a data set.

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

Description — The array *start* gives the multi-dimensional index of the starting corner of the hyperslab to write. The values are zero based.

The array *edge* gives the number of values to write along each dimension of the hyperslab.

The array *stride* allows for sub-sampling along each dimension. If a stride value is specified for a dimension, that many values will be skipped over when writing along that dimension. Specifying *stride*=NULL writes contiguous data. No matter what stride value is provided, data is always read contiguously from *data*.

FORTRAN

```
integer function sfwdata(sds_id, start, stride, edge, data)
    integer*4       sds_id
    integer         start(*), stride(*), edge(*)
    <valid type>    data(*)
```

vfadtr

Vaddtagref

```
int32 Vaddtagref(int32 vgroup_id, int32 tag, int32 ref)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
tag	IN:	Tag of object to add
ref	IN:	Reference number of object to add

Purpose This routine inserts the tag and reference number of a data object into the specified Vgroup.

Return value Returns the number of tag and reference numbers in the Vgroup if successful and `FAIL (-1)` otherwise.

Description This routine is primarily used to add an HDF object that is neither a Vgroup nor a Vdata into a Vgroup. The HDF object is specified by the *tag* and *ref* parameters.

FORTRAN

```
integer function vfadtr(vgroup_id, tag, ref)
    integer*4 vgroup_id, tag, ref
```

Vattach

vfatch

Vattach

```
int32 Vattach(int32 file_id, int32 vgroup_ref, char *access)
```

file_id	IN:	File identifier returned by **Hopen**
vgroup_ref	IN:	Reference number for the Vgroup to open
access	IN:	Type of access; "r" for read, "w" for write.

Purpose — This routine attaches to an existing Vgroup or creates a new Vgroup.

Return value — Returns a Vgroup identifier if successful and `FAIL(-1)` otherwise.

Description — The *file_id* is the file identifier of an opened HDF file. The *vgroup_ref* parameter specifies which Vgroup in the HDF file to attach to:

if *vgroup_ref* = -1, a new Vgroup is created.
if *vgroup_ref* is positive, the Vgroup corresponding to the *vgroup_ref* is attached.

Vattach returns a Vgroup identifier for the accessed Vgroup. This identifier or vgroup_id, is used for all further operations on the Vgroup. Once operations are compelte, the vgroup_id must be disposed of via a call to **Vdetach**. Multiple attaches may be made to the same Vgroup simultaneously, giving rise to several Vgroup identifiers for the same Vgroup. Each vgroup_id must be disposed of independently.

FORTRAN

```
integer function vfatch(file_id, vgroup_ref, access)
    integer*4       file_id, vgroup_ref
    character*1     access
```

Vdetach

```
VOID Vdetach(int32 vgroup_id)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**

Purpose — This access routine detaches a Vgroup that is currently attached, terminating further access to that Vgroup.

Return value — None.

Description — All space associated with that Vgroup will be freed. Each attached Vgroup should be detached using this routine before the file is closed. **Vdetach** also updates the Vgroup information in the HDF file if any changes occur. The identifier *vgroup_id* should not be used after that Vgroup is detached.

FORTRAN

```
subroutine vfdtch(vgroup_id)
    integer*4 vgroup_id
```

Vend

vfend

Vend

```
VOID Vend(int32 file_id)
```

file_id	IN:	File identifier returned by **Hopen**

Purpose — Releases the internal data structures for the specified HDF file.

Return value — None.

Description — **Vend** releases all internal data structures allocated by calling **Vstart**. This routine must be called after all Vdata/Vgroup operations on an HDF file are completed. Further attempts to use Vdata or Vgroup routines after calling **Vend** will cause errors.

FORTRAN

```
subroutine vfend(file_id)
    integer*4 file_id
```

vfgcls

Vgetclass

```
VOID Vgetclass(int32 vgroup_id, char *vgroup_class)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
vgroup_class	OUT:	Buffer for storing the Vgroup class name

Purpose — This routine returns the class name (if any) of a Vgroup .

Return value — None.

Description — This routine retrieves the class name of the Vgroup specified by *vgroup_id* and returns it in the buffer pointed to by *vgroup_class*. The maximum length of the name is defined by the macro VGNAMELENMAX.

FORTRAN

```
subroutine vfgcls(vgroup_id, vgroup_class)
    integer*4       vgroup_id
    character*(*)   vgroup_class
```

Vgetid

vfgid

Vgetid

```
int32 Vgetid(int32 file_id, int32 vgroup_ref)
```

file_id	IN:	File identifier returned by **Hopen**
vgroup_ref	IN:	Reference number of the previous Vgroup

Purpose
This search routine searches through the HDF file and returns the reference number of the next Vgroup following the Vgroup that has the reference number *vgroup_ref*. It is generally used to search the file sequentially for Vgroups.

Return value
Returns the reference number of the next Vdata if successful and FAIL(-1) otherwise.

Description
To initiate a search, call this routine with a vgroup_ref value of -1. This will return the reference number of the first Vgroup in the file. Searching past the last Vgroup in the file will return an error.

Example
This sample code prints out the reference numbers of all Vgroups in the HDF file.

```
int32 file_id; /* id of an opened HDF file */
int32 vgroup_ref;

file_id = Hopen("myfile.hdf",DFACC_READ,0);
vgroup_ref = -1;
while(1) {
   vgroup_ref = Vgetid(file_id, vgroup_ref);
   if (vgroup_ref == -1) break;
   printf("found Vgroup with id %d\n", vgroup_ref);
}
Hclose(file_id);
```

FORTRAN

```
integer function vfgid(file_id, vgroup_ref)
   integer*4 file_id, vgroup_ref
```

vfgnam

Vgetname

```
VOID Vgetname(int32 vgroup_id, char *vgroup_name)
```

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_name	OUT:	Buffer to store Vgroup name

Purpose — This inquiry routine returns the name of the Vgroup.

Return value — None.

Description — This routine retrieves the name of the Vgroup specified by *vgroup_id* and returns it in the buffer pointed to by *vgroup_name*. The maximum length of the name is defined by the macro VGNAMELENMAX.

FORTRAN

```
subroutine vfgnam(vgroup_id, vgroup_name)
   integer*4      vgroup_id
   character*(*)  vgroup_name
```

Vgettagref

vfgttr

Vgettagref

```
intn Vgettagref(int32 vgroup_id, int32 index, int32 *tag, int32 *ref)
```

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
index	IN:	Index of the tag and reference number to retrieve from the Vgroup
tag	OUT:	Pointer to location to store the tag found in the indexed position
ref	OUT:	Pointer to location to store the ref number found in the indexed position

Purpose

This inquiry routine returns the tag/ref of an HDF object at a given position with a Vgroup. Do not confuse with **Vgettagrefs**.

Return value

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

Description

The input parameter *index* specifies the position within the Vgroup. This routine provides a means of accessing the tag/ref pairs in a Vgroup by specifying the position within the Vgroup.

Example

This sample code first uses **Vntagrefs** to determine the number of tag/ref pairs in a Vgroup. It then uses **Vgettagref** to extract each tag/ref one at a time.

```
int32 vgroup_id;  /* pointer to an attached Vgroup */
int32 tag, ref;
int32 status, i, npairs;

npairs = Vntagrefs(vgroup_id);
for (i=0; i<npairs; i++) {
   status = Vgettagref (vgroup_id, i, &tag, &ref);
   printf ("found tag=%d ref=%d at position %d\n", tag, ref, i);
   }
```

FORTRAN

```
integer function vfgttr(vgroup_id, tag_array, ref_array, index)
   integer*4      vgroup_id, index
   integer*4      tag, ref
```

Vgettagrefs

```
int32 Vgettagrefs(int32 vgroup_id, int32 tag_array[], int32 ref_array[],
          int32 maxsize)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
tag_array	OUT:	Array to store returned tags
ref_array	OUT:	Array to store returned reference numbers
maxsize	IN:	Maximum number of tag/ref pairs to store in *tag_array* and *ref_array*

Purpose

This inquiry routine returns the tag/ref pairs of the HDF objects belonging to a given Vgroup. Do not confuse with **Vgettagref**.

Return value

Returns the number of tag/ref pairs in a specified Vgroup if successful and FAIL(-1) otherwise.

Description

The input parameter *maxsize* specifies the maximum number of tag/ref pairs to be returned. The tag/ref pairs will be returned in arrays *tag_array* and *ref_array*. Each array must be at least *maxsize* in size.

Example

This sample code uses **Vntagrefs** to determine the number of tag/ref pairs in a Vgroup. It then allocates memory , and uses **Vgettagrefs** to extract the tag/ref values.

```
int32 vgroup_id;  /* pointer to an attached Vgroup */
int32 npairs, status;
int32 *tags, * refs;

npairs = Vntagrefs(vgroup_id);
tags   = (int32*) malloc (sizeof(int32) * npairs);
refs   = (int32*) malloc (sizeof(int32) * npairs);
status = Vgettagrefs(vgroup_id, tags, refs, npairs);
```

FORTRAN

```
integer function vfgttrs(vgroup_id, tag_array, ref_array, maxsize)
    integer*4      vgroup_id, maxsize
    integer*4      tag_array(*), ref_array(*)
```

Vinqtagref

vfinqtr

Vinqtagref

```
intn Vinqtagref(int32 vgroup_id, int32 tag, int32 ref)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
tag	IN:	Tag of item to check for
ref	IN:	Reference number of items to check for

Purpose — This inquiry routine tests whether a given HDF object is linked to a given Vgroup.

Return value — Returns `TRUE(1)` if the tag and reference number are linked to the specified Vgroup, and `FALSE(0)` otherwise.

Description — The HDF object is specified by the *tag* and *ref* parameter. This routine then checks if the tag/ref pair can be found in the Vgroup *vgroup_id*.

FORTRAN

```
integer function vfinqtr(vgroup_id, tag, ref)
    integer*4 vgroup_id, tag, ref
```

vfinq

Vinquire

```
intn Vinquire(int32 vgroup_id, int32 *n_entries, char *vgroup_name)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
n_entries	OUT:	Pointer to variable to store the number of entries in a Vgroup
vgroup_name	OUT:	Buffer to store the name of a Vgroup.

Purpose — This is the general Vgroup inquiry routine.

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

Description — This routine retrieves the number of entries in a Vgroup and the name of the Vgroup. The maximum length of the Vgroup's name is defined by the macro `VGNAMELENMAX`.

FORTRAN

```
integer function vfinq(vgroup_id, n_entries, vgroup_name)
    integer*4       vgroup_id, n_entries
    character*(*)   vgroup_name
```

Vinsert

vfinsrt

Vinsert

```
int32 Vinsert(int32 vgroup_id, int32 v_id)
```

vgroup_id	IN:	The Vgroup identifier returned by **`Vattach`**
v_id	IN:	Identifier for the item (Vdata or Vgroup) to insert into a Vgroup

Purpose
This routine establishes a link from a Vgroup to either another Vgroup or to a Vdata identifier.

Return value
Returns the position of the inserted Vgroup or Vdata within the Vgroup if successful and `FAIL(-1)` otherwise.

Description
Essentially, **`Vinsert`** allows Vgroups or Vdatas to be grouped together. To insert other HDF objects (ie that are not Vgroups or Vdatas) into a Vgroup, use **`Vaddtagref`**. The Vdata or the Vgroup *v_id* will be added to the Vgroup *vgroup_id*.

FORTRAN

```
integer function vfinsrt(vgroup_id, v_id)
    integer*4 vgroup_id, v_id
```

Vlone

```
int32 Vlone(int32 file_id, int32 ref_array[], int32 maxsize)
```

file_id	IN:	File identifier returned by **Hopen**
ref_array	OUT:	Buffer to store the reference numbers of lone Vgroups
maxsize	IN:	Maximum number of Vgroups to store in *ref_array*.

Purpose This inquiry routine returns an array of reference numbers for all Vgroups that are not linked to any Vgroup in the HDF file.

Return value Returns the total number of lone Vgroups in a file if successful and `FAIL(-1)` otherwise.

Definition This routine is provided for applications to locate all lone Vgroups (ie those that are not grouped with other objects) in a HDF file. Another use is to locate all Vgroups at the top of any grouping hierarchy.

The parameter *maxsize* specifies the maximum number of reference numbers to be returned. The reference numbers will be returned in the array *ref_array[]*. The array must be at least *maxsize* elements in size.

The return value from this function will be the total number of Vgroups that are not linked to any Vgroup in the file. But at most *maxsize* reference numbers will be returned in *ref_array*.

An array size of 65000 integers for *ref_array* is more than adequate. The preferred method is to use dynamic memory instead; first call **Vlone** with a value of 0 for *maxsize*, and then use the returned value to allocate memory for *ref_array* before calling **Vlone**.

Example The sample code illustrates the preferred method of using **Vlone**. The second call to **Vlone** returns the required reference numbers.

```
int32 file_id;      /* id of opened HDF file */
int32 maxsize,  status;
int32 *ref_array;

maxsize = Vlone (file_id, NULL, 0);
ref_array  = (int32*) malloc(sizeof(int32)*maxsize);
status   = Vlone (file_id, ref_array, maxsize);
```

FORTRAN

```
integer function vflone(file_id, ref_array, maxsize)
   integer*4      file_id,
   integer*4      ref_array(*), maxsize
```

Vntagrefs

vfntr

Vntagrefs

`int32 Vntagrefs(int32 vgroup_id)`

vgroup_id	IN:	The Vgroup identifier returned by **`Vattach`**

Purpose — This inquiry routine returns a count of the number of HDF objects belonging to a given Vgroup.

Return value — Returns the number of HDF elements linked to the Vgroup and or `FAIL(-1)` otherwise.

Description — The routine returns a count of the number of tag and reference number pairs stored in the given Vgroup. This routine is used together with **`Vgettagrefs`**, or in a loop with **`Vgettagref`** to look at the HDF objects linked to a given Vgroup.

FORTRAN

```
integer function vfntr(vgroup_id)
   integer*4      vgroup_id
```

vfscls

Vsetclass

```
VOID Vsetclass(int32 vgroup_id, char *vgroup_class)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
vgroup_class	IN:	Name of class for Vgroup

Purpose — This specify routine sets the class name for a Vgroup.

Return value — None.

Description — Vgroups initially have a class name of NULL. The class name may be set more than once. Class names, like Vgroup names, can be of any character strings. They exist solely as meaningful labels for user applications.

FORTRAN

```
subroutine vfscls(vgroup_id, vgroup_class)
   integer*4       vgroup_id
   character*(*)   vgroup_class
```

Vsetname

vfsnam

Vsetname

```
VOID Vsetname(int32 vgroup_id, char *vgroup_name)
```

vgroup_id	IN:	The Vgroup identifier returned by **Vattach**
vgroup_name	IN:	Name for the Vgroup

Purpose — This specify routine sets the name for a Vgroup.

Return value — None.

Description — Vgroups each initially have a name of NULL, and may be renamed more than once. Note that the routine does not check for uniqueness of Vgroup names.

Vgroup names are optional, but recommended. They serve as meaningful labels for user applications. If used, they should be unique.

FORTRAN

```
subroutine vfsnam(vgroup_id, vgroup_name)
   integer*4      vgroup_id
   character*(*)  vgroup_name
```

vfstart

Vstart

```
VOID Vstart(int32 file_id)
```

file_id	IN:	File identifier returned by **Hopen**

Purpose — Initialize internal data structures for HDF files containing Vdata or Vgroups.

Return value — None.

Description — **Vstart** initializes the internal Vdata and Vgroup data structures in an HDF file. This routine must be called before any Vdata or Vgroup operation is attempted on an HDF file. **Vstart** must be called once for each file involved in the operation.

FORTRAN

```
subroutine vfstart(file_id)
    integer*4       file_id
```

VFfieldesize

```
int32 VFfieldesize(int32 vdata_ref, int32 field_index)
```

vdata_ref	IN:	Reference number for the Vdata to open
field_index	IN:	Number determined by assigning each field in a record a number starting with `0`

Purpose: Retrieve the size (as stored in an HDF file) of a specified field in a given Vdata.

Return value: Returns the size of a value for the specified field if successful and `FAIL(-1)` otherwise.

FORTRAN: None

VFfieldisize

```
int32 VFfieldisize(int32 vdata_ref, int32 field_index)
```

vdata_ref	IN:	Reference number for the Vdata to open
field_index	IN:	Number determined by assigning each field in a record an index starting with `0`

Purpose: Retrieve the size (as stored in memory) of a specified field in a give Vdata.

Return value: Returns the size of a value for the specified field if successful and `FAIL(-1)` otherwise.

FORTRAN: None

VFfieldname

```
char *VFfieldname(int32 vdata_ref, int32 field_index)
```

vdata_ref	IN:	Reference number for the Vdata to open
field_index	IN:	Number determined by assigning each field in a record an index starting with 0

Purpose — Retrieves the fieldname for a specified field in a given Vdata.

Return value — Returns a pointer to the fieldname if successful and NULL otherwise.

FORTRAN — None

VFfieldorder

```
int32 VFfieldorder(int32 vdata_ref, int32 field_index)
```

vdata_ref	IN:	Reference number for the Vdata to open
field_index	IN:	Number determined by assigning each field in a record an index starting with `0`

Purpose	Retrieves the order of a specified field in a given Vdata.
Return value	Returns the order of the field if successful and `FAIL(-1)` otherwise.
FORTRAN	None

VFfieldtype

```
int32 VFfieldtype(int32 vdata_ref, int32 field_index)
```

vdata_ref	IN:	Reference number for the Vdata to open
field_index	IN:	Number determined by assigning each field in a record an index starting with `0`

Purpose	Retrieves the number type for a specified field in a given Vdata.
Return value	Returns the number type if successful and `FAIL(-1)` otherwise.
FORTRAN	None

VFnfields

```
int32 VFnfields(int32 vdata_ref)
```

vdata_ref	IN:	Reference number for the Vdata to open

Purpose	Retrieve the total number of fields in the specified Vdata.
Return value	Returns the number of fields if successful and `FAIL(-1)` otherwise.
FORTRAN	None

VHmakegroup

```
int32 VHmakegroup(int32 file_id, int32 tag_array[], int32 ref_array[],
          int32 n_objects, char *vgroup_name, char *vgroup_class)
```

file_id	IN:	HDF file identifier returned by **Hopen**
tag_array	IN:	Array of HDF tags to store
ref_array	IN:	Array of HDF reference numbers to store
n_objects	IN:	Number of data objects to store
vgroup_name	IN:	Name of the Vgroup to create
vgroup_class	IN:	Class of the Vgroup to create

Purpose
A high-level method to group a collection of data objects into a Vgroup.

Return Value
Reference numbers of newly created Vgroup if successful, `FAIL(-1)` otherwise.

Description
This routine creates a new Vgroup named *vgroup_name* of class *vgroup_class* in the HDF file *file_id*. The *tag_array* and *ref_array* are matched arrays of *n_objects* containing the tags and reference numbers to be added to the new Vgroup. A matched array in this case means that *tag_array[0]* and *ref_array[0]* refer to one data object, and *tag_array[1]* and *ref_array[1]* to another, etc.

Because **Vstart** initialize the VH interface, it must precede calls to **VHmakegroup**. It is not necessary, however, to call **Vattach** or **Vdetach** in conjunction with **VHmakegroup**.

FORTRAN
None

VHstoredata

```
int32 VHstoredata(int32 file_id, char *fieldname, uint8 buf[], int32 n_records,
          int32 num_type, char *vdata_name, char *vdata_class)
```

file_id	IN:	HDF file identifier returned by **Hopen**
fieldname	IN:	Field name for the new Vdata
buf	IN:	Buffer the records are read from
n_records	IN:	Number of records to store
num_type	IN:	Number type for the field in each record
vdata_name	IN:	Name of the Vdata created
vdata_class	IN:	Class of the Vdata created

Purpose A high-level method for creating Vdatas containing records limited to one field with one component per field.

Return Value Reference number of newly created Vdata if successful, `FAIL(-1)` otherwise

Description This routines provides a high-level method for creating single-order, single-field Vdatas. It creates a Vdata named *vdata_name* of class *vdata_class* in the HDF file referenced by *file_id*. The number type of the field is specified by *num_type* and *n_records* of data from *buf* are stored with a field name specified by *fieldname*.

Because **Vstart** initialize the VH interface, it must preced **VHstoredata**. It is not necessary, however, to call **Vattach** or **Vdetach** in conjunction with **VHstoredata**.

FORTRAN None

VHstoredatam

```
int32 VHstoredatam(int32 file_id, char *fieldname, uint8 buf[], int32 n_records,
          int32 num_type, char *vdata_name, char *vdata_class, int32 order)
```

file_id	IN:	HDF file identifier returned by **Hopen**
fieldname	IN:	Field name for the new Vdata
buf	IN:	Buffer the records are read from
n_records	IN:	Number of records to store
num_type	IN:	Number type of data elements
vdata_name	IN:	Name of the Vdata created
vdata_class	IN:	Class of the Vdata created
order	IN:	The number of components (data values) per field

Purpose

A high-level method for creating Vdatas containing records with one field containing one or more components.

Return Value

Reference number of newly created Vdata if successful, `FAIL(-1)` otherwise

Description

This routines provides a high-level method for creating multi-order, single-field Vdatas. It creates a Vdata named *vdata_name* of class *vdata_class* in the HDF file referenced by *file_id*. The number type of the Vdata and all its components is specified by *num_type* and *n_records* of data from *buf* are stored with a field name specified by *fieldname*. The *order* of the Vdata indicates the number of data values stored per field.

Because **Vstart** initialize the VH interface, it must precede **VHstoredatam**. It is not necessary, however, to call **Vattach** or **Vdetach** in conjunction with **VHstoredatam**.

FORTRAN

None

vsfatch

VSattach

```
int32 VSattach(int32 file_id, int32 vdata_ref, char *access)
```

file_id	IN:	File identifier returned by **Hopen**
vdata_ref	IN:	Reference number for the Vdata to open
access	IN:	Type of access; "r" for read, "w" for write.

Purpose
Attaches to an existing Vdata or creates a new Vdata with the specified reference number and returns an identifier to that Vdata.

Return value
Returns a Vdata identifier if successful and FAIL(-1) otherwise.

Description
The attach returns an identifier to the Vdata, through which all further operations on that Vdata are carried out. See also **VSdetach()**.

Access is either "r" (read) or "w" (write).

If *access* is "r", then *vdata_ref* must be a valid reference number of an existing Vdata returned from any of the Vdata search routines (e.g. **Vgetnext** or **VSgetid**). It is an error to attach to a *vdata_ref* of -1 with "r" access.

If *access* is "w", then *vdata_ref* may be a valid reference number of an existing Vdata. An existing Vdata is generally attached with "w" access to replace part of its data, or to appended new data to it. The parameter *vdata_ref* may also be -1; this is used to create a new (empty) Vdata. The default interlace for a new Vdata is FULL_INTERLACE. (This may be changed using **VSsetinterlace**).

An existing Vdata may be multiply attached for reads, but only one attach with write access to a Vdata is allowed.

FORTRAN

```
integer function vsfatch(file_id, vdata_ref, access)
   integer*4      file_id, vdata_ref
   character*1    access
```

VSdetach

vsfdtch

VSdetach

```
VOID VSdetach(int32 vdata_id)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**

Purpose
Detaches from the current Vdata thereby terminating further access to that Vdata.

Return value
None

Description
VSdetach updates the Vdata information in the HDF file if there are any changes. All memory used for that Vdata is freed. The *vdata_id* should not be used after that Vdata is detached.

FORTRAN

```
subroutine vsfdtch(vdata_id)
    integer*4 vdata_id
```

vsfelts

VSelts

```
int32 VSelts(int32 vdata_id)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**

Purpose — Determines the number of elements (records) in a Vdata.

Return Value — The number of elements in a Vdata if successful and `FAIL(-1)` otherwise.

Description — This routine determines the number of elements (i.e. records) in the Vdata *vdata_id*, and returns that value to the user.

FORTRAN

```
integer function vsfelts(vdata_id)
integer vdata_id
```

VSfdefine

vsffdef

VSfdefine

```
intn VSfdefine(int32 vdata_id, char *fieldname, int32 numbertype, int32 order)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
fieldname	IN:	Name of field to define
numbertype	IN:	Number type for the field values
order	IN:	Order of the new field

Purpose

Defines a new field for in a Vdata.

Return value

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

Description

VSfdefine is only used to define fields in a new Vdata; it does not set the format of a Vdata. Note that defining a field using **VSfdefine** does not prepare the storage format of the Vdata. Once the fields have been defined, the routine **VSsetfields** must be used to set the format. **VSfdefine** may only be used with a new (empty) Vdata. Once there is data in a Vdata, definitions of fields in the Vdata may not be modified or deleted.

A field is defined by its name (fieldname), its type (number type) and its order (order). The name of the field is any sequence of characters. By convention, this is usually mnemonic, e.g. "PRESSURE". The type of a field specifies whether a field is float, integer, etc. Thus, *numbertype* may be one of the following:

32-bit float	DFNT_FLOAT32 5
64-bit float	DFNT_FLOAT64 6
8-bit signed int	DFNT_INT8 20
8-bit unsigned int	DFNT_UINT8 21
16-bit signed int	DFNT_INT16 22
16-bit unsigned int	DFNT_UINT16 23
32-bit signed int	DFNT_INT32 24
32-bit unsigned int	DFNT_UINT32 25

The order of a field is the number of components in that field. Single variables like time or pressure have an order of 1. Compound variables have an order greater than 1. For example, the variable VELOCITY has an order of 3, since velocity has three components.

vsffdef

Example

The following two calls to **VSfdefine** define 2 fields in the newly created Vdata. The first field, named VEL, is a 3-component float-valued field, while the second field, named PRESSURE, is a single-component float-valued field.

```
Vdata = (int32) VSattach(hdf_file, -1, "w");
VSfdefine(Vdata, "VEL", DFNT_FLOAT32, 3);
VSfdefine(Vdata, "PRESSURE", DFNT_FLOAT32, 1);
```

FORTRAN

```
integer function vsffdef(vdata_id, fieldname, numbertype, order)
    integer*4      vdata_id, numbertype, order
    character*(*)  fieldname
```

VSfexist

vsfex

VSfexist

```
intn VSfexist(int32 vdata_id, char *fields)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
fields	IN:	Names of the fields to check for

Purpose: Checks to see if the specified fields exist in the current Vdata.

Return value: A value of `1` is returned if all field(s) exist; otherwise `FAIL(-1)` is returned

Description: **VSfexist** checks if a set of fields exist in a specified Vdata. The *fields* argument is a string of comma-separated fieldnames (e.g. "PX,PY,PZ).

FORTRAN

```
integer function vsfex(vdata_id, fields)
   integer*4      vdata_id
   character*(*)  fields
```

vsffnd

VSfind

```
int32 VSfind(int32 file_id, char *vdata_name)
```

file_id	IN:	File identifier returned by **Hopen**
vdata_name	IN:	Name of the Vdata

Purpose — Looks in the HDF file for a Vdata with a given name.

Return value — Returns the Vdata reference number if successful and `FAIL(-1)` otherwise.

Description — If there are several Vdatas with the same name, **VSfind** will only find the reference number of the first Vdata in the file with that name.

FORTRAN

```
integer function vsffnd(file_id, vdata_name)
   integer*4      file_id
   character*(*)  vdata_name
```

VSgetclass

vsfgcls

VSgetclass

```
VOID VSgetclass(int32 vdata_id, char *vdata_class)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
vdata_class	OUT:	Pointer to space to store the class name

Purpose — Returns the class name (if any) of the Vdata.

Return value — None.

Description — VSgetclass retrieves the class name of the specified Vdata and places it in the memory pointed to by *vdata_class*. The maximum length of the class name is defined by the macro VSNAMELENMAX.

FORTRAN

```
subroutine vsfgcls(vdata_id, vdata_class)
   integer*4      vdata_id
   character*(*)  vdata_class
```

VSgetfields

```
int32 VSgetfields(int32 vdata_id, char *fields)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
fields	OUT:	Storage for field names

Purpose

Retrieves the field names (if any) of all the fields in a Vdata.

Return value

Returns the number of fields in the Vdata if successful and FAIL(-1) otherwise.

Description

This routine retrieves the field names (if any) of all the fields in the specified Vdata. The field names are returned as a comma-separated string (e.g., "PX,PY,PZ").

FORTRAN

```
integer function vsfgfld(vdata_id, fields)
   integer*4      vdata_id
   character*(*)  fields
```

VSgetid

vsfgid

VSgetid

```
int32 VSgetid(int32 file_id, int32 vdata_ref)
```

file_id	IN:	File identifier returned by **Hopen**
vdata_ref	IN:	Reference number for the Vdata preceding the Vdata of interest

Purpose — Sequentially searches through an HDF file for Vdatas.

Return value — Returns the reference number for the next Vdata if successful and `FAIL(-1)` otherwise.

Description — **VSgetid** searches through the specified HDF file and returns the reference number of the next Vdata after the Vdata that has reference number *vdata_ref*. This routines is generally used to search the file sequentially for Vdatas.

To initiate a search, this routine may be called *vdata_ref* equal to -1 for. Doing so returns the reference number of the first Vdata in the file. Searching past the last Vdata in a file will return an error.

FORTRAN

```
integer function vsfgid(file_id, vdata_ref)
    integer*4 file_id, vdata_ref
```

VSgetinterlace

```
int32 VSgetinterlace(int32 vdata_id)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**

Purpose Retrieves the interlace of a Vdata.

Return value Returns FULL_INTERLACE or NO_INTERLACE if successful and FAIL(-1) otherwise.

Description This inquiry routine returns the interlace type of the Vdata *vdata_id*.

FORTRAN

```
integer function vsfgint(vdata_id)
    integer*4 vdata_id
```

VSgetname

vsfgnam

VSgetname

```
VOID VSgetname(int32 vdata_id, char *vdata_name)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
vdata_name	OUT:	Storage space for Vdata name

Purpose
Retrieves the name of a Vdata.

Return value
None.

Description
VSgetname returns in *vdata_name* the name (if any) of the specified Vdata.The maximum length of the class name is defined by the macro VSNAMELENMAX. If there is no name for the Vdata, a null string "" is placed in *vdata_name*.

FORTRAN

```
subroutine vsfgnam(vdata_id, vdata_name)
   integer*4      vdata_id
   character*(*)  vdata_name
```

vsfinq

VSinquire

```
intn VSinquire(int32 vdata_id, int32 *n_records, int32 *interlace, char *fields,
          int32 *vdata_size, char *vdata_name)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
n_records	OUT:	The number of records in the Vdata
interlace	OUT:	Type of interlace: FULL_INTERLACE or NO_INTERLACE
fields	OUT:	A comma-separated string listing all fields in the Vdata (e.g. "PX, PY, PZ")
vdata_size	OUT:	The size in bytes of a record in the Vdata for the local machine.
vdata_name	OUT:	The name (if any) of the Vdata

Purpose: General Vdata inquiry routine.

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

Description: **VSinquire** retrieves the information for each of the variables listed above. If any of the parameters are NULL, then that data is not retrieved. See also: **VSgetfields**, **VSgetinterlace**, **VSsizeof**, and **VSgetname**.

FORTRAN:

```
integer function vsfinq(vdata_id, n_records, interlace, fields,
vdata_size, vdata_name)
   integer*4      vdata_id, n_records, interlace, vdata_size
   character*(*)  fields, vdata_name
```

VSlone

vsflone

VSlone

```
int32 VSlone(int32 file_id, int32 ref_array[], int32 maxsize)
```

file_id	IN:	File identifier returned by **Hopen**
ref_array	OUT:	Array to store a list of Vdata reference numbers
maxsize	IN:	Maximum number of Vdata reference numbers to store in the array

Purpose

Returns an array of reference numbers of all Vdatas that are not linked to any Vgroup in the HDF file.

Return value

Returns the total number of lone Vdata in a file if successful and `FAIL(-1)` otherwise.

Description

This routine is provided for applications to locate all lone Vdatas (i.e. those that are not grouped with other objects) in a HDF file.

You must provide the integer array *ref_array*, and you must specify the maximum size of the array in the argument *maxsize*. The return value from this function will be the total number of Vdatas that are not linked to any Vgroup in the file. But at most *maxsize* reference numbers will be returned in *ref_array*.

An array size of 65000 integers for *ref_array* will be usually more than adequate. The preferred method is to use dynamic memory instead; first call **VSlone** with a value of 0 for *maxsize*, and then use the returned value to allocate memory for *ref_array* to be passed to a subsequent call to **VSlone**. See also **Vlone**.

Example

The sample code illustrates the preferred method of using **VSlone**. The second call to **VSlone** returns the required reference numbers.

```
int32 file_id;     /* id of opened HDF file */
int32 maxsize, status;
int32 *ref_array;

maxsize = VSlone (file_id, NULL, 0);
ref_array = (int32*) malloc(sizeof(int32)*maxsize);
status   = VSlone (file_id, ref_array, maxsize);
```

FORTRAN

```
integer function vsflone(file_id, ref_array, maxsize)
   integer*4      file_id
   integer*4      ref_array(*)
   integer*4      maxsize
```

VSread

```
int32 VSread(int32 vdata_id, uint8 *databuf, int32 n_records,
          int32 interlace)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
databuf	OUT:	Buffer to store the retrieved data
n_records	IN:	Number of records to retrieve
interlace	IN:	Interlacing for the data stored in the buffer

Purpose — Retrieves data from a Vdata.

Return value — Returns the total number of records read if successful and `FAIL(-1)` otherwise.

Description — Given a Vdata *vdata_id* (with read access), and a buffer, *databuf*, this routine returns the data read from that Vdata in the buffer.

You must specify in *n_records* the number of records to be read, You must also specify the buffer interlace in *interlace*. (use `FULL_INTERLACE` (recommended) or `NO_INTERLACE`).

To read data from a Vdata, typically, the Vdata is first attached with read access, and then **VSsetfields** is called to specify which field(s) are to be read from that Vdata. Each subsequent call to **VSread** after that will return records sequentially from that Vdata.

If the call is successful, the data returned in *databuf* is laid out as specified by *interlace*, and the data fields appear in the order specified in the last call to **VSsetfields** for that Vdata. To retrieve any arbitrary record from a Vdata, use **VSseek** to specify the record position before calling **VSread**. See also **VSseek**, and **VSsetfields**.

FORTRAN

```
integer function vsfread(vdata_id, databuf, n_records,
interlace)
    integer*4       vdata_id, n_records, interlace
    character*(*)   databuf
```

VSseek

vsfseek

VSseek

```
int32 VSseek(int32 vdata_id, int32 record)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
record	IN:	Record number to seek to

Purpose
Provides a mechanism for random-access I/O within a Vdata.

Return value
Returns the record position (zero or a positive integer) if successful and `FAIL(-1)` otherwise.

Description
This routine moves the access pointer within the Vdata *vdata_id* to the position of the record specified by *record*. The next **VSread** or **VSwrite** will read the current record. For example, to seek to the third record in the Vdata, set *record* to `2`. The first record position is specified by specifying a record of `0`. Each seek is constrained to a record boundary within the Vdata.

FORTRAN

```
integer function vsfseek(vdata_id, record)
    integer*4 vdata_id, record
```

vsfscls

VSsetclass

```
VOID VSsetclass(int32 vdata_id, char *vdata_class)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
vdata_class	IN:	Name of the Vdata class

Purpose
Sets the class name of the Vdata.

Return value
None.

Description
Vdatas initially have a class name of NULL. The class name may be reset more than once. Class names, like Vdata names, can be of any character strings. They exist solely as meaningful labels to applications and are not used by the HDF library in any way. See also **VSgetclass**.

FORTRAN

```
subroutine vsfscls(vdata_id, vdata_class)
   integer*4      vdata_id
   character*(*)  vdata_class
```

VSsetfields

vsfsfld

VSsetfields

```
intn VSsetfields(int32 vdata_id, char *fields)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
fields	IN:	Name of Vdata fields to access

Purpose

Establishes the fields to be accessed within the Vdata by any subsequent read or write routine (**VSread** or **VSwrite**).

Return value

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

Description

Fields are specified by the string argument *fields*, a comma-separated list of fieldnames (e.g. "PX,PY,PZ"). The combined width of the fields in a Vdata must be less than 32,000 bytes. If an attempt to create a larger record is made, **VSsetfields** will return an `FAIL`.

The fieldnames allowed depends on the access of the Vdata: If access is "`r`", the fieldnames must be one of the names of the fields that existed in the Vdata when that Vdata was created. If access is "`w`", the fieldnames can be any fieldname that has been defined by calling **VSfdefine**. Users may define any fields to be used in a Vdata. Several fieldnames have already been predefined, and may be used without having to define them. See the *HDF User's Guide* for details.

VSsetfields must be called before any called to **VSread** or **VSwrite** is allowed. It sets up the data translation tables for accessing the data in a Vdata.

For reading from a Vdata, a call to **VSsetfields** sets up the fields that are to be retrieved from records in the Vdata.

For writing to a Vdata, **VSsetfields** can only be called once, to set up the fields (i.e. format) of the records in a Vdata. Once the Vdata format is set, it may not be changed.

FORTRAN

```
integer function vsfsfld(vdata_id, fields)
   integer*4      vdata_id
   character*(*)  fields
```

VSsetinterlace

```
intn VSsetinterlace(int32 vdata_id, int32 interlace)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
interlace	IN:	Interlace for storing data in an HDF file

Purpose
Sets the interlace type for a Vdata.

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

Description
VSsetinterlace determines the interlace for the data written to the specified Vdata. The interlace may be specified by either FULL_INTERLACE or NO_INTERLACE. If this routine is not called the default interlace type is FULL_INTERLACE. The FULL_INTERLACE option is more efficient than NO_INTERLACE although both require the same amount of disk space. This routine can only be used when creating new Vdatas with write access.

Specifying FULL_INTERLACE fills the Vdata by record, i.e. all fields in a record are filled before moving to the next record. Specifying NO_INTERLACE fills the Vdata by field, i.e. the first field in each record is filled before moving to next series of fields.

FORTRAN

```
integer function vsfsint(vdata_id, interface)
    integer*4 vdata_id, interlace
```

VSsetname

vsfsnam

VSsetname

```
VOID VSsetname(int32 vdata_id, char *vdata_name)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
vdata_name	IN:	Name for the Vdata

Purpose
Assigns a name to a Vdata.

Return value
None.

Description
Vdatas initially have a Vdata name of NULL. The name may be reset more than once. Vdata names, like class names, can be of any character strings. They exist solely as a meaningful label for other applications and are not used by the HDF library in any way. See also **VSgetname**.

FORTRAN

```
subroutine vsfsnam(vdata_id, vdata_name)
   integer*4      vdata_id
   character*(*)  vdata_name
```

VSsizeof

```
int32 VSsizeof(int32 vdata_id, char *fields)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
fields	IN:	Name(s) of the fields to check

Purpose

Computes the byte size of the given field(s) for the local machine.

Return value

Returns the local machine size in bytes for the field if successful and FAIL(-1) otherwise.

Description

Specify a single field or several fields (comma-separated) in the parameter *fields*. The field(s) should already exist for that Vdata *vdata_id*.

This routine is useful in determining how much memory needs to be allocated to read in the field(s) in question.

Example

The code segment below uses **VSsizeof** to compute the byte size of a data value of the "DENSITY" field, and uses **VSQuerycount** to get the total number of records in the Vdata. The product gives the total memory size needed to read all the "DENSITY" data values from the Vdata.

```
int32 varsize, n_records;
unsigned char *readbuffer;
int32 vdata_id;

VSQuerycount(vdata_id, &n_records);
varsize = VSsizeof(vdata_id,"DENSITY");
readbuffer = (unsigned char*) malloc(varsize * n_records);
```

FORTRAN

```
integer function vsfsiz(vdata_id, fields)
   integer*4      vdata_id
   character*(*)  fields
```

VSwrite

vsfwrit

VSwrite

```
int32 VSwrite(int32 vdata_id, unsigned char *databuf, int32 n_records,
          int32 interlace)
```

vdata_id	IN:	The Vdata access identifier returned from **VSattach**
databuf	IN:	Buffer of records to store in the Vdata
n_records	IN:	Number of records to store
interlace	IN:	Interlace of the buffer in memory

Purpose
Writes data into a Vdata.

Return value
Returns the total number of records written if successful and FAIL(-1) otherwise.

Description
Given a buffer containing data to be written and a Vdata with write access, this routine writes the data from the buffer to the Vdata. The n_records argument specifies the number of record to be written and interlace defines the order in which the Vdata fields are written. Selecting FULL_INTERLACE fills the Vdata by record, i.e. all fields in a record are filled before moving to the next record. FULL_INTERLACE is recommended for speed and efficiency. Specifying NO_INTERLACE fills the Vdata by field, i.e. the first field in each record is filled before moving to next series of fields.

Before writing data to a newly created Vdata, define the format of the Vdata via a call to **VSsetfields**. If the Vdata already exists, **VSsetfields** is unnecessary because data can only be written to a Vdata in compliance with its existing format.

The data in databuf is assumed to be organized in memory as specified by the interlace argument. It is also assumed to contain the exact number and order of data fields defined in the last call to **VSsetfields**. **VSwrite** writes the contents of databuf contiguously to the Vdata, therefore any "padding" or non-data spaces used for record or field alignment must be removed before attempting to write from databuf to the Vdata.

The interlace specifies how the data exists within the buffer, *databuf*. If the data is to be stored in a Vdata with an interlace different from that of the buffer, **VSsetinterlace** must be called prior to **VSwrite**. In most cases this is unnecessary since the default interlace type of FULL_INTERLACE is used. See also **VSsetfields**, **VSseek**, **VHstoredata**.

FORTRAN

```
integer function vsfwrit(vdata_id, databuf, n_records,
interlace)
   integer*4      vdata_id, n_records, interlace
   character*(*)  databuf
```

VSQuerycount

```
intn VSQuerycount(int32 vdata_id, int32 *n_records)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**
n_records	OUT:	Number of records in the Vdata

Purpose: Returns the number of records in the Vdata in *n_records*.

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

FORTRAN: None

VSQueryfields

```
intn VSQueryfields(int32 vdata_id, char *fields)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**
fields	OUT:	Comma-separated string listing the fields in the Vdata

Purpose: Returns the names of the fields in the Vdata in *fields*.

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

FORTRAN: None

VSQueryinterlace

```
intn VSQueryinterlace(int32 vdata_id, int32 *interlace)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**
interlace	OUT:	Interlace type: FULL_INTERLACE or NO_INTERLACE

Purpose: Returns the type of interlace in the Vdata in *interlace*.

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

FORTRAN: None

VSQueryname

```
intn VSQueryname(int32 vdata_id, char *vdata_name)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_name	OUT:	Name (if any) of the Vdata

Purpose: Returns the name of the Vdata in *vdata_name*.

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

FORTRAN: None

VSQueryref

```
int32 VSQueryref(int32 vdata_id)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose	Retrieves the reference number for the specified Vdata.
Return value	Returns the reference number of the Vdata if successful and `FAIL(-1)` otherwise.
FORTRAN	None

VSQuerytag

```
int32 VSQuerytag(int32 vdata_id)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose	Retrieves the tag for the specified Vdata.
Return value	Returns the tag of the Vdata if successful and `FAIL(-1)` otherwise.
FORTRAN	None

VSQueryvsize

```
intn VSQueryvsize(int32 vdata_id, int32 *vdata_size)
```

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_size	OUT:	Size (in bytes) of a record in the Vdata for the local machine.

Purpose	Retrieves the local size in bytes of a record in the Vdata in *vdata_size*.
Return value	Returns SUCCEED(0) if successful and FAIL(-1) otherwise.
FORTRAN	None

Chapter 3

HDF Utility Reference Section

3.0 Overview

This section of the *HDF Reference Manual* contains a listing of each utility contained in the HDF v3.3 library. The pages are organized alphabetically by utility name. Each utility begins on a new page and takes the following form:

Utility_Name

```
utility_name [option1] [option2]
```

Purpose	Section containing a short description of the utility	
Description	Section detailing the specific use of the named utility. It defines any special circumstances surrounding the use of the utility, and describes any known complications affecting its use.	
Options	`option1`	Definition of first option in synopsis
	`option2`	Definition of second option is synopsis
Example	This optional section consists of one or more examples of the use of the utility.	

fp2hdf

```
fp2hdf -help
fp2hdf <infile> [<in_file>...] -o <out_file> [-r [raster_opts...]|-f]
```

Purpose

Converts floating point data to a scientific data set (SDS) and/or 8-bit raster image set (RIS8) format, storing the results in an HDF file. The image data can be scaled about a mean value.

Description

Input file(s), containing a single two-dimensional or three-dimensional floating point array in either ASCII text, native floating point, 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 input format is ASCII text or native floating point, it must have the following input fields:

format	"TEXT" or "FP32" or "FP64"
nplanes	"n" for (n+1)dimensional input
nrows	Number of rows in the image
ncols	Number of columns in the image
max_value	Maximum value in the data set
min_value	Minimum value in the data set
plane1 plane2 ...planen	Scales for depth axis
row1 row2 ...rown	Scales for the vertical axis
col1 col2 ...coln	Scales for the horizontal axis
data1 data2 ...datan	Data ordered left to right, top to bottom, front to back.

For FP32 and FP64 input, *format*, *nplanes*, *nrows*, *ncols*, and *nplanes* are native integers; where *format* is the integer representation of the appropriate 4-character string (0x46503332 for FP32 and 0x46503634 for 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.

Options

-h	Print a helpful summary of usage, and exit.
<*in_file*>	Name(s) of the file(s) containing one 2D or 3D array
-o <*out_file*>	Name of the HDF file to hold the output
-r[raster_opts]|-f	
-r [raster_opts]	Store as a raster image set in the output file
-f	Default to store a SDS in the output file

raster_opts

[-e <*hor*><*vert*>[<*depth*>]|-i <*hor*><*vert*>[<*depth*>] -p <*palfile*> -m <*mean*>

-e <*hor*> <*vert*> [<*depth*>] Expand the data via pixel replication

hor	Number of columns in the new image
vert	Number of rows in the new image
depth	Number of images or depth planes for 3D data only

-i <*hor*> <*vert*> [<*depth*>] Expand the data via interpolation

hor	Number of columns in the new image (> original)
vert	Number of rows in the new image (> original)
depth	Number of images for 3D data only (> original)

-p <*palfile*> Store the palette with the image

palfile	File containing an HDF palette or a raw palette

-m <*mean*> Scale data about a mean floating point value

mean The new extremes (newmax and newmin), as given by:

```
newmax=mean+max(abs(max-mean), abs(mean-min))
newmin=mean-max(abs(max-mean), abs(mean-min))
```

will be equidistant from the mean value. If no mean value is given, then the mean will be:

```
mean=0.5*(max+min)
```

Example Convert floating point data in `f1.txt` to SDS format, and store it in HDF file `o1`:

```
fp2hdf f1.txt -o o1
```

Example Convert floating point data in `f2.hdf` to 8-bit raster format, and store it in HDF file `o2`:

```
fp2hdf f2.hdf -o o2 -r
```

Example Convert floating point data in `f3.bin` to 8-bit raster format and SDS format, and store both in HDF file `o3`:

```
fp2hdf f3.bin -o o3 -r -f
```

Example Convert floating point data in `f4` to a 500x600 raster image, and store the RIS8 in HDF file `o4`. Also store a palette from `palfile` with the image:

```
fp2hdf f4 -o o4 -r -e 500 600 -p palfile
```

Example Convert floating point data in `f5` to 200 planes of 500x600 raster images, and store the RIS8 in HDF file `o5`. Also scale the image data so that it is centered about a mean value of 10.0:

```
fp2hdf f5 -o o5 -r -i 500 600 200 -m 10.0
```

hdf24to8

```
hdf24to8 <in_file> <out_file>
```

Purpose — Converts 24-bit raster image sets into 8-bit raster image sets containing a palette.

Description — The utility **hdf24to8** quantizes a HDF RGB 24-bit image into an 8-bit image with 256-color palette and stores both the palette and image in an HDF file.

Options

in_file	Name of the HDF file containing the 24-bit raster image set
out_file	Name of the HDF file to store the raster image set and palette

Example — All **hdf24to8** operations take the following form:

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

hdf2jpeg

`hdf2jpeg <in_file> <out_file template>`

Purpose — Copies JPEG compressed images from an HDF file into separate JFIF files.

Description — The **`hdf2jpeg`** utility copies compressed JPEG images from HDF files to JFIF files. If multiple JPEG images are to be copied from the HDF file, the "#" character may be used in the JPEG filename template. The "#" character will be replaced with the reference number of the image being copied. The default JPEG name, if none is specified on the command line is `file.#.jpg`

Options

in_file	Name of HDF file containing the compressed images
out_file	Name of JFIF file to store the JPEG data

Example — To copy one image from an HDF file to a JFIF file, execute a command line similar to the following:

```
hdf2jpeg filename.hdf image1.jfif
```

Example — To copy multiple images from an HDF file to separate JFIF files, execute a command line similar to the following:

```
hdf2jpeg filename.hdf image-#.jfif
```

hdf8to24

```
hdf8to24 [-s <scale>] [-j [quality]] <in_file> <out_file>
```

Purpose — Converts 8-bit raster image sets to 24-bit raster image sets.

Description — The **hdf8to24** utility converts an 8-bit raster image stored in an HDF file into 24-bit raster images stored in a new HDF file. The 8-bit raster image can also be scaled in size during the conversion and stored using JPEG compression.

Options

`-s <scale>`	Magnification option
`<scale>`	Floating point number greater than or equal to 0.01
`-j [quality]`	JPEG compression option with default quality factor of 75
`[quality]`	Integer between 1 and 100, with 100 indicating highest quality
in_file	HDF file containing the 8-bit raster image set
out_file	HDF file containing the 24-bit raster image set

Example — To convert an 8-bit image into a 24-bit image, enter the following command line

```
hdf8to24 file1.hdf file2.hdf
```

Example — To convert an 8-bit image into a 24-bit image scaled by a factor of 2, enter the following command line

```
hdf8to24 -s2 file1.hdf file3.hdf
```

Example — To convert an 8-bit image into a 24-bit compressed image using a JPEG quality factor of 45, enter the following command line

```
hdf8to24 -j45 file1.hdf file4.hdf
```

hdfcomp

```
hdfcomp <out_file> <[-c],[-r],[-i], [-j] <quality>> <in_file> [ ...<[-c],[-r],[-i]>
<in_file>]
```

Purpose

Compresses 8-bit raster images by moving them from one HDF file to another.

Description

The **hdfcomp** utility reads uncompressed 8-bit raster images from HDF files and writes compressed 8-bit raster images to HDF files. Compressed images are appended to the output file, if the output file exists.

Options

out_file Name of the HDF file to store the compressed image

-c|-i|-r|-j Compression type:

- -c Run length encoding
- -i IMCOMP compression (requires HDF file with a palette)
- -r No compression
- -j JPEG compression
  - <*quality*> JPEG quality factor valued from 1 - 100

in_file Name of HDF file containing uncompressed image

Example

A directory contains 20 files named `storm001`, `storm002`, `...storm020`. Each file contains a single 8-bit image. To compress these 20 images using run-length encoding and store them in a file called `allcomp.hdf` execute the following command:

```
hdfcomp allcomp.hdf -c storm*.hdf
```

hdfed

`hdfed [filename] [-nobackup] [-batch]`

Purpose — Edits HDF files at the data element level.

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

The **hdfed** utility is designed for users who need to manipulate HDF files at the data element level. It is not designed to provide a high-level view of HDF files. To use **hdfed** effectively requires familiarity with the format of HDF files, therefore it is not recommended for users who are unfamiliar with low level HDF components. The **hdfed** utility also has a small set of special commands, including a conditional statement, a looping statement, an alias command and an unalias command.

The **hdfed** utility will not allow the abitrary modification of binary data or the data element. It does allow modification of tag and reference numbers within strict constraints. Do not attempt to alter individual bytes. It is acceptable, to replace a data element with another of the same type. See the *HDF User's Guide* for more complete instructions.

Options

filename	Name of the HDF file to edit
-nobackup	Cancels default backup file
-batch	Sets input to a series of commands, rather individual commands
-help	Displays usage information, and a listing of the **hdfed** commands

Commands — Once inside the editor, the -help command may be used after any **hdfed** command (i.e. dump -help) to get more information regarding a specific command.

All of the **hdfed** commands can be abbreviated to the extent that their abbreviations are unique. For example -he is an ambiguous abbreviation because it could stand for either the flag -hexadecimal or the flag -help; on the other hand, the flag -hel is not ambiguous. The **hdfed** commands are described as follows:

`annotate [-label|-descriptor] [-editor <editor>]`

Purpose — Edit an annotation

Options

-label	Edit label (default)
-descriptor	Edit descriptor
-editor	Use editor (default EDITOR env value)

`close [-keep]`

Purpose — Close current open file.

Options

-keep	Option to retain backup files.

`delete`

Purpose Delete this object or group.

Options None

`display [-position <xpos> <ypos>] [-expansion <exp>] [-large]`

Purpose Display image on screen.

Options

`-position`	Image position on console screen
`-expansion`	Image expansion factor
`-large`	Make image as large as possible

`dump [-offset <offset>] [-length <len>][-decimal|-short|-byte|-octal|-hexadecimal|-float|-double|-ascii|-udecimal|-ushort]`

Purpose Display the data object contents in the format specified.

Options

`-offset`	Start offset
`-length`	Length to look at
`-decimal`	Decimal format [32 bit integers]
`-short`	Decimal format [16 bit integers]
`-byte`	Decimal format [8 bit integers]
`-octal`	Octal format [Default]
`-hexadecimal`	Hexadecimal format
`-float`	Float format [32 bit floats]
`-double`	Float format [64 bit floats]
`-ascii`	ASCII format
-udecimal	Unsigned decimal format [32-bit floats]
-ushort	Unsigned decimal format [64-bit floats]

`getr8 <`*`filename`*`> <xdim> <ydim> [-pal <palette>] [-raster|-rle|-imcomp]`

Purpose Get an 8-bit raster image set from raw files

Options

`filename`	Name of file containing image
<xdim>	Number of columns in the image
<ydim>	Number of rows in the image
`-pal`	Raw palette file
`-raster`	No compression (default)
`-rle`	Run-length compression
`-imcomp`	IMCOMP compression

`help`

Purpose Print a help screen describing the functionality of **`hdfed`**.

Options None

`info [-all] [-long] [-group] [-label]`

Purpose Show information for one data object.

Options

`-all`	Show info for all objects in file
`-long`	Show more info
`-group`	Organize info in group(s)
`-label`	Show label if any

`next [<predicates>]`

Purpose: Move to the next object that satisfies the predicate

Options: <predicates> are of the general form:
`<item> <comparator> <value>` or `group`

`<item>`	tag, ref, image_size, or label.
`<comparator>`	"=", "!=", ">", "<", ">=", "<="
`<value>`	any number

`open <filename> [-nobackup]`

Purpose: Opens a file *filename*.

`<filename>`	Name of the HDF file to edit
`[-nobackup]`	Cancels default back-up file

`prev [<predicates>]`

Purpose: Move to the next object that satisfies the predicate.

Options: See `next` for examples of `<predicates>`

`put [-file <file>] [-verbose]`

Purpose: Put the raw binary of this object in a file

`-file`	Out file name (default "elt#.@")
`-verbose`	Output diagnostic info

`putr8 [-image <img>] [-palette <pal>] [-verbose]`

Purpose: Put an r8 group into raw image and palette files

`-image`	Image file name (default "img#.@.%")
`-palette`	Palette file name (default "pal#")
`-verbose`	To give output of steps taken

`quit`

Purpose: Exit **`hdfed`**

Options: None

`revert`

Purpose: Discard all changes made in **`hdfed`**.

Options: None

`select [<predicates>] <commands>* end`

Purpose: Step through all elements in the file that satisfy the predicates and execute the commands on them.

Options: if [<predicates>] <commands>* end
Execute commands if predicates are satisfied by element

`wait [<message>]`

Purpose: Print message and then wait for user to hit return

Options: `message`

`write <file> [-attachto <atag> <aref>]`

Purpose: Write an element or group into another HDF file

`-attachto`	What element to write annotation to

hdfls

```
hdfls [-o] [-l] <filename>
```

Purpose

Provides a quick listing of the tags, reference numbers, and lengths for each of the data objects contained in the specified file.

Description

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

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

Options

`-o`	Displays the reference numbers in ascending order
`-l`	Displays default information plus the length of each element
`filename`	Name of the HDF file to examine

Example

A file called example.hdf contains three data elements associated with a raster image set: (1) the dimension element, (2) a palette, and (3) the image array. To display the tag and reference number for each element, enter the following command:

```
hdfls example.hdf
```

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

```
Example.hdf
Image Dimensions-8(Raster-8) : (tag 200) Ref nos: 1
Image Palette-8(Raster-8) : (tag 201) Ref nos: 3
Raster Image-8(Raster-8) : (tag 202) Ref nos: 2
```

hdfpack

```
hdfpack [-i|-b] [-d#] [-t#] <in_file> <out_file>
```

Purpose

Compacts an HDF file by reading in all the data elements in the file and writing them out to a new file.

Description

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

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

Options

`-b|-i`	Linked block option
`-b`	Don't coalesce the linked blocks if they exist
`-i`	Prompt for whether or not to coalesce any existing linked blocks
`-d#`	Number of descriptors for all data descriptor lists in the output file
`-t#`	Set the number of linked blocks per table entry in the output file
`in_file`	Name of HDF file containing unpacked data
`out_file`	Name of HDF file containing the packed data

Example

To compact the file `aa.hdf` and put the result in `aa.pck`, enter the following command:

```
hdfpack aa.hdf aa.pck
```

Example

Suppose the file `bb.hdf` contains elements stored as sequences of linked blocks. To compact this file but leave the linked-block elements intact, and put the result in `bb.blk`, enter the following command:

```
hdfpack -b bb.hdf bb.blk
```

hdfrseq

```
hdfrseq [-s] [-l] [-p xloc yloc] [-e magnify] <filename> [...filename]
```

Purpose

Displays one or more 8-bit images in a specified sequence on a Sun-3 or SGI.

Description

The HDF 8-bit remote image sequencer, otherwise known as the **hdfrseq** utility, will display a sequence of images, one after the other, from files containing 8-bit raster image sets. This sequencer will only display images on a Sun console, an SGI IRIS with access to the gllib library, or a Macintosh. If more than one file is listed, the corresponding images will be displayed in the order in which the files are specified. Each new image will overwrite the preceding image on the screen and only one image will be displayed per file.

The **hdfrseq** utility requires a terminal emulator such as NCSA Telnet 2.3 or Teltool 1.2. When **hdfrseq** executes on a file, it converts the image from the HDF 8-bit raster image format into NCSA's ICR (interactive color raster) format and sends it to the receiving terminal. ICR is a protocol that is easy to transmit to a remote display station. If the receiving software on the display station can decipher the ICR protocol, it displays the image on the console.

In a typical application, the display station would be running a program, such as NCSA Telnet (Version 2.3 or later) or NCSA Teltool (Version 1.2 or later), that understands ICR and immediately converts the incoming stream to a screen display. For more information on **hdfrseq**, consult the HDF User's Guide.

Options

`-s`	Display single image until `RETURN` is pressed
`-l`	Paints the image as large as possible on the screen
`-p xloc yloc`	Set the location of image on the screen
xloc	Column number for upper left pixel
yloc	Row number for upper right pixel
`-e magnify`	Enlarges all the images in the sequence
magnify	Number of times to increase image
filename	Name(s) of the file(s) contributing to the image sequence

Example

Assume four images are stored as raster image sets in the files img1.hdf, img2.hdf, img3.hdf, and img4.hdf. To view all four images three times their actual size, enter the following command:

```
hdfrseq -e 3 img1.hdf img2.hdf img3.hdf img4.hdf
```

The images printed to the screen will appear as a four image animation sequence.

hdftopal

```
hdftopal <in_file< <out_file>
```

Purpose — Converts palettes stored in HDF files to palettes stored in raw data files.

Description — The **hdftopal** utility converts a palette from an HDF file to a raw palette in a raw palette file. The outgoing palette will have 768 bytes: first 256 red values, then 256 greens, then 256 blues. Although HDF stores palettes by interlacing their color components, the raw file will contain a palette stored as 256 red components, followed by 256 green components, and finally 256 blue components.

Options

in_file	HDF file containing a palette
out_file	768-byte output file containing the RGB components

Example — A typical **hdftopal** command line operation appears as follows:

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

hdftor8

```
hdftor8 <in_file> [-i] [-v] [-r out_rfile][-p out_pfile]
```

Purpose

Extracts raster image sets from HDF files, separates them into image data and palette data, and saves them to separate raw data files.

Description

The utility **hdftor8** extracts the images and or palettes from an HDF file and stores them in two sets of files that contain only images and palettes, respectively.

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

The *in_file* parameter is the filename for the file to extract images and palettes from, and the *out_rfile* and *out_pfile* are the basic names of the files that will contain the raw images and palettes. These basic names are extended as follows: For each image file, the filename is given the extension ".#.@.%", where # stands for the image number from the original file, @ is the x dimension of the image, and % is the y dimension of the image. For each palette file, the filename is given the extension ".#", where # stands for the palette number from the original file.

If no names are given for the image file, the default name "img.#.@.%" is used, where #, @ and % are defined as in the preceding paragraph. Similarly the default name for a palette file is "pal.#"

Options

in_file	Name of the HDF file containing the raster image set
-i	Sets mode so filenames can be specified interactively
-v	Sets verbose mode to provide descriptive messages
-r *out_rfile*	Specifies the name of the file to store the raster image
-p *out_pfile*	Specified the name of the file to store the palette

Example

A file called `denm.hdf` contains three 512 x 256 images and three palettes. To extract these images and palettes and put them in separate files, enter the following command:

```
hdftor8 denm.hdf
```

Six files are produced with the names `img.1.512.256`, `img.2.512.256`, `img.3.512.256`, `pal.1`, `pal.2`, and `pal.3`.

jpeg2hdf

```
jpeg2hdf <in_file> <out_file>
```

Purpose — Copies a JPEG compressed image from a JFIF compliant file into an HDF file.

Description — The **jpeg2hdf** utility copies a JPEG compressed image from a JFIF compliant file into an HDF file. The image is not decompressed during the copy. If the HDF file already exists, the image is appended to it without overwriting any data.

Options

in_file	File containing the raw raster image
out_file	The HDF file to store the compressed raster image set

Example — To copy several JPEG files into a single HDF file, execute command lines similar to the following:

```
jpeg2hdf image1.jfif store.hdf
jpeg2hdf image2.jfif store.hdf
jpeg2hdf image3.jfif store.hdf
```

paltohdf

`paltohdf <in_file> <out_file>`

Purpose — Converts raw palette files into the HDF format.

Description — The **paltohdf** utility converts a raw palette to HDF format. The incoming palette is assumed to have 768 bytes organized in the following order: 256 red values, 256 green values, and 256 blue values. The palette in the HDF file will have the RGB values interlaced in an RGB RGB ...RGB sequence. This is the standard HDF format for 8-bit palettes.

Options

in_file	768-byte raw data file containing the RGB components
out_file	The HDF file to store the palette

Example — To convert a raw palette into an HDF palette using **paltohdf**, execute a command line similar to:

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

r24hdf8

`r24hdf8 <cols> <rows> <in_file> <out_file>`

Purpose

Converts 24-bit images into 8-bit raster image sets containing palettes.

Description

The **r24hdf8** utility converts a raw RGB 24-bit pixel image into an 8-bit image with a 256-color palette and stores both the palette and image in an HDF file. The raw raster file is assumed to contain 8-bit color component values ordered in the following manner: RGB, RGB, ...RGB. The order of pixels is left to right and top to bottom.

The *out_file* is an HDF file with one 8-bit raster image set; i.e. 8-bit image, dimensions, and RGB palette.

Options

cols	Number of columns in the image
rows	Number of rows in the image
in_file	File containing the raw 24 bit image
out_file	The HDF file to store the 8-bit raster image set

Example

To convert a 24-bit image to an 8-bit raster image set, execute a command line similar to the following:

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

r8tohdf

r8tohdf <*rows*> <*cols*> <*out_file*> [-p *palfile*] [-c|-i|-r] <*in_file*> [...<*in_file*>]

Purpose — Converts raw 8-bit raster images into HDF raster image sets.

Description — The **r8tohdf** utility converts each named raw 8-bit raster image into an HDF raster image set and stored them in a single HDF file. To convert a series of images in a single command line operation, all the named images must be the same size.

Options

rows	Height of the image(s)
cols	Width of the image(s)
out_file	Name of the HDF file that will store the raster image sets
-p *palfile*	Adds the palette to each raster image group
palfile	Name of the file containing the palette
-c|-i|-r	Compression:
-c	Run length encoding
-i	IMCOMP compression
-r	No compression
in_file	Name of the file(s) that contain the raw 8-bit image data

Example — Consider a 256 row x 512 column raw raster image stored in a file called `raster.raw`. Assume its palette is stored in raw form in a file called `palette.raw`. To create a single raster image set by including data from both files, use the following command line operation:

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

Example — A combination of different types of raster image sets is to be stored in a file called `Example.hdf`. The raster image set created from the first file `Example1.raw` will not contain a palette, however the raster image sets created from `Example2.raw` and `Example3.raw` will include a palette stored in the `pal.raw` file. In addition, `Example1.raw` and `Example2.raw` will be compressed using run length encoding, yet `Example3.hdf` will remain uncompressed. All images are 256 x 512. Performing this conversion will require a command line containing the following information:

```
r8tohdf 256 512 Example.hdf -c Example1.raw -p pal.raw
Example2.raw -r Example3.raw
```

Example — A 256 x 512 byte raw raster image is contained in a file called `rawras`, and the palette with which it is to be used is stored in a file called `mypal`. To convert this information to an RIS8 without compression and store the result in a file called `ras.hdf`, enter:

```
r8tohdf 256 512 ras.hdf -p mypal rawras
```

Example — An 800 x 1000 raw raster is stored in a file called `bigpic`. To convert this image to a run lenght encoded RIS8 stored without a palette in a file called `bigpic.hdf` enter:

```
r8tohdf 800 1000 bigpic.hdf -c bigpic
```

Example

Three 300 x 400 raw images are contained in files `pic1`, `pic2`, and `pic3`. To convert all three files to raster image sets with imcomp compression and store them in `pic.hdf`, enter:

```
r8tohdf 300 400 pic.hdf –i pic1 pic2 pic3
```

Example

A combination of different types of raster image sets is to be stored in a file called `ras.hdf`. The image from file `rawras1` is to be stored without a palette. The images from `rawras2` are to be stored with a palette that is copied from a file called `mypal`. The images from `rawras1` and `rawras2` are to be compressed using run length encoding, and `rawras3` is not to be compressed. All images are 256 x 512:

```
r8tohdf 256 512 ras.hdf –c rawras1 –p mypal rawras2 –r rawras3
```

ristosds

```
ristosds <in_file> [ ...in_file] <-o out_file>
```

Purpose

Converts a series of HDF raster image files into a single HDF scientific data set.

Description

The utility converts a series of HDF raster image files into a single HDF file containing a 3 dimensional scientific data set. Each input file may contains one or more raster images. If there is a palette associated with the images, the palette should be included in the first HDF file. Any subsequent palettes will be ignored.

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

Options

in_file	Name of HDF file(s) containing the 8-bit raster images set(s)
out_file	Name of HDF file that will contain the 3D scientific data set

Example

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

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

vcompat

```
vcompat <filename>
```

Purpose — Converts v1.0 Vdata or Vgroup files to v2.0 Vdata or Vgroup files.

Description — This utility modifies HDF files containing Vdata or Vgroup to be compatible with version 2.0 or later of the HDF Vset library. Note that the file is assumed to be incompatible to begin with. This routine will not check to see if the file is already compatible, but it is harmless to run an already-compatible file through again.

Options

filename	Name of the HDF file to be converted

vmake

```
vmake -help
```

Purpose — The **vmake** utility is a general utility for generating Vdata and Vgroups in HDF files. Invoking **vmake** with no arguments provides help. You may use **vmake** in three different ways:

1. to create new Vgroups,
2. to store data in new Vdatas, and
3. to create links among Vgroups and Vdatas.

```
vmake <filename> <Vgroup_name>
```

Description — The **vmake** utility creates a new Vgroup, named *Vgroup_name* in the HDF file called *filename*. This new Vgroup will not overwrite any information within the file, if that file exists. If it doesn't, **vmake** creates a new file. Upon successful creation, the program prints the reference number of that Vgroup.

Options

filename	Name of the HDF file
Vgroup_name	Name of the Vgroup to be created

```
vmake <filename> <Vdata_name> <field=[#][f|b|s|l|c]> [...<field=[#][f|b|s|l|c]>]
```

Description — The **vmake** utility creates a new Vdata, named *Vdata_name* in the HDF file called *filename*, and stores ASCII data from standard input in the Vdata according to field specifications in *format*.

Options

filename	Name of the HDF file
Vdata_name	Name of the Vdata to be created
field=[#][f|b|s|l|c]	
field	Name of the first field in the Vdata record
#	Optional number of components in the field, default is 1
[f|b|s|l|c]	Number type for all components in the field
f	float32
b	int8 (byte)
s	int16 (short)
l	int32 (long)
c	character

Example — This example demonstrates how to create a Vdata named `coordinates` containing 3 fields `PX`, `PY` and `PZ`, each of type 32-bit float. In this case, a stream of ASCII floating-point numbers is read from the ASCII file `pos.dat`.

```
vmake vgroup1.hdf "coordinates" "PX=f,PY=f,PZ=f" < pos.dat
```

Example

This example will create a Vdata named `simulation5` containing a single field of type float named `DENSITY`. In this case, the output from the program `myprog` (a stream of ASCII floating-point numbers) is piped into **vmake**.

```
myprog | vmake vset.hdf "simulation5" "DENSITY=f"
```

vmake <*filename*> <-l> <Vgroup-ref> [...*ref1 ref2* ...*refn*]

Description

The **vmake** utility in this context allows one or several Vdatas or Vgroups to be inserted (linked) to a Vgroup. All Vgroups and Vdata are identified by their reference numbers. Reference numbers of existing Vgroups and Vdatas can be determined through **vshow**.

The **vmake** utility permits the creation of empty Vgroups, but not empty Vdatas. Vgroups and Vdatas may be created in any order by the user. However, to perform linking, the Vgroups and Vdatas involved must already exist. If **vmake** is successful it prints a positive number; otherwise, a negative number is printed.

The **vmake** utility does not check for cycles, and will happily allow you to insert a Vgroup into itself. It is legal to create multiple Vgroups with identical names.

Options

filename	Name of the HDF file
-l	Indicates use of link option to add objects to a Vgroup
Vgroup_ref	Reference number for the Vgroup into which the other objects are to be inserted
ref1 ref2 ...refn	Reference numbers for objects to be inserted into the Vgroup

vshow

`vshow` *`filename`* `[+[n]]`

Purpose — Lists Vdata and Vgroup information in an HDF file.

Description — For an HDF file containing Vdata or Vgroups, **`vshow`** will display information on all the Vgroups in the file, in the order stored, followed by the Vdata in the file.

Information about Vdata that are not linked to a Vgroup are preceded by an "L".

The **`vshow`** utility is primarily used to examine the contents of a file. But **`vshow`** is also useful for converting Vdata and Vgroups files into ASCII. This utility outputs full data dumps of Vdata through standard output, and outputs general information through standard error.

For full Vdata dumps, specify either one Vdata or all Vdatas.

Options

`n`	dumps data for the Vdata with reference number *`n`*, unformatted.
`+`*`n`*	gives full dump of Vdata with ID n.

Example — The following example will create an ASCII file containing formatted ASCII data from the Vdata with reference number 4.

```
vshow myfile.hdf +4 > text.file
```

Glossary

A

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

annotation interface: collection of routines identified by the prefix DFAN (C) or da (FORTRAN) that are used to annotate HDF files and data objects.

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

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

attribute record: an HDF data object containing a description of the environment from which a file (global attribute) or data set (local attribute) was created, the contents of an attribute record are sometimes referred to as meta data**.**

D

da: the FORTRAN prefix for functions in the single file annotation interface.

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

description: an annotation containing one or more text strings.

desc: abbreviation used in the DFAN interface to identify a description assigned to a data object.

d8: the FORTRAN prefix for functions in the single file 8-bit raster interface.

DF interface: an obsolete collection of low level HDF routines that were replaced by the H routines in version 3.2 of the HDF library.

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

DFAN: the C prefix for functions in the single file annotation interface.

DFP: the C prefix for functions in the single file palette interface.

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

DFSD: the C prefix for functions in the single file scientific data interface.

dp: the FORTRAN prefix for functions in the single file palette interface.

ds: the FORTRAN prefix for functions in the single file scientific data interface.

d2: the FORTRAN prefix for functions in the single file 24-bit raster interface.

E

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

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

8-bit raster interface: collection of routines identified by the prefix DFR8 (C) or d8 (FORTRAN) that are used to manage 8-bit raster image sets.

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

F

fid: abbreviation used in the DFAN interface to identify a file label.

fds: abbreviation used in the DFAN interface to identify a file description.

G

global attribute: an annotation assigned to describe an entire HDF file.

H

H: the C prefix for low level routines in the HDF library.

H interface: a collection of low level HDF routines used for handling file I/O, memory management, and physical storage in HDF files.

High-level Vdata/Vgroup interface: a collection of HDF routines identified by the prefix VH that are used to create simple Vdatas and Vgroups.

HDF: the C prefix for low level multi-file routines in the HDF library.

HDF interface: a collection of low level HDF routines used to support the multi-file application interfaces.

HE: the C prefix for low level error reporting routines in the HDF library.

HE interface: a collection of low level routines used to manage the HDF error stack.

I

interlacing: a method of ordering the 8-bit raster values comprising a 24-bit image; see pixel interlacing, scan-line interlacing, or scan-plane interlacing.

L

label: an annotation containing one text string terminated by `'\0'`.

lab: abbreviation used in the DFAN interface to identify a label.

local attribute: an annotation assigned to a specific data object in a file; may be selected from a predefined set of attributes or user defined.

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

M

meta data: data typically stored in an attribute record that describes the data in an HDF file or data object.

multi-file interface: one of several independent modules containing HDF routines designed to store and access more than one HDF file or data object at the same time.

N

netCDF: an interface developed by the Unidata Program Center for storing multi-dimensional arrays along with their attribute records.

P

palette data: a 768-byte array of 8-bit numbers used to identify which 256 colors appear in an 8-bit raster image; also called a color look-up table.

palette interface: collection of HDF routines identified by the prefix DFP (C) or dp (FORTRAN) that are used to manage 8-bit raster image sets.

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

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

R

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

S

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

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

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

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

scientific data set interface: a collection of single file or multi-file HDF routines that are used to manage multi-dimensional arrays of varying number type.

SD: the C prefix for functions in the multi-file scientific data interface.

sf: the FORTRAN prefix for functions in the multi-file scientific interface.

single file application interface: one of several independent modules containing HDF routines designed to store and access a specific type of data set one file at a time.

T

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

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

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

24-bit raster interface: collection of HDF routines identified by the prefix DF24 (C) or d2 (FORTRAN) that are used to manage 24-bit raster image sets.

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

U

utilities: small application programs executed through a command line interface.

V

V: the C prefix for functions in the multi-file Vgroup interface.

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

Vdata field interface: a collection of HDF routines identified by the prefix VF (C only) that are used to inquire about the names, size, and number types for the fields in a Vdata.

V (cont.)

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

Vdata interface: a collection of HDF routines identified by the prefix VS (C) or vsf (FORTRAN) that are used to manage customized tables in HDF files.

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

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

Vdata Query interface: a collection of HDF routines identified by the prefix VQS (C only) that are used to inquire about the contents of a Vdata.

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

VF: the C prefix for functions in the multi-file Vdata field interface.

VF interface: a collection of HDF routines identified by the prefix VF that are used to query Vdata about their field sizes, types, and field names.

vf: the FORTRAN prefix for functions in the multi-file Vgroup interface.

Vgroup class: a label typically used to describe the members of a data set**.**

Vgroup interface: a collection of HDF routines identified by the prefix V(C) or vf (FORTRAN) that are used to read and write customized data sets.

Vgroup model: a framework for storing customized data sets in an HDf file.

Vgroup name: a label used to identify or describe a Vgroup.

VH: the C prefix for functions in the high level Vdata/ Vgroup interface.

VS: the C prefix for functions in the multi-file Vdata interface.

vsf: the FORTRAN prefix for functions in the multi-file Vdata interface.

VSQ: the C prefix for functions in the multi-file Vdata query interface.

VSQ interface: a collection of HDF routines identified by the prefix VSQ that are used to query Vdata about their records, names, and number types.

Index

Numerics

24-bit raster image set
see: DF24 interface

8-bit raster image set
see: DFR8 interface

A

annotation
see: DFAN interface

D

d2aimg
see: DF24addimage **93**

d2first
see: DF24restart **101**

d2gdims
see: DF24getdims **94**

d2gimg
see: DF24getimage **95**

d2lref
see: DF24lastref **96**

d2nims
see: DF24nimages **97**

d2pimg
see: DF24putimage **98**

d2reqil
see: DF24reqil **100**

d2rref
see: DF24readref **99**

d2sdims
see: DF24setdims **103**

d2setil
see: DF24setil **104**

d2wref
see: DF24writeref **105**

d8aimg
see: DFR8addimage **41**

d8first
see: DFR8restart **48**

d8gdims
see: DFR8getdims **42**

d8gimg
see: DFR8getimage **43**

d8lref
see: DFR8lastref **44**

d8nims
see: DFR8nimages **45**

d8pimg
see: DFR8putimage **46**

d8rref
see: DFR8readref **47**

d8spal
see: DFR8setpalette **50**

d8wref
see: DFR8writeref **51**

daafds
see: DFANaddfds **19**

daafid
see: DFANaddfid **20**

dagdesc
see: DFANgetdesc **21**

dagdlen
see: DFANgetdesclen **22**

dagfds
see: DFANgetfds **23**

dagfdsl
see: DFANgetfdslen **24**

dagfid
see: DFANgetfid **25**

dagfidl
see: DFANgetfidlen **26**

daglab
see: DFANgetlabel **27**

dagllen
see: DFANgetlablen **28**

dallist
see: DFANlablist **29**

dalref
see: DFANlastref **30**

dapdesc
see: DFANputdesc **31**

daplab
see: DFANputlabel **32**

DF24 interface
general interface routines **6**
read routines **5**
write routines **5**

DF24addimage **93, 98, 105**
see also: d2aimg **93**

DF24getdims **94, 95**
see also: d2gdims **94**

DF24getimage **94, 95, 99, 100**
see also: d2gimg **95**
DF24lastref **96**
see also: d2lref **96**
DF24nimages **97**
see also: d2nims **97**
DF24putimage **93, 98, 105**
see also: d2pimg **98**
DF24readref **99**
see also: d2rref **99**
DF24reqil **95, 100**
see also: d2reqil **100**
DF24restart **101**
see also: d2first **101**
DF24setcompress **102**
DF24setdims **103**
see also: d2sdims **103**
DF24setil **104**
see also: d2setil **104**
DF24writeref **105**
see also: d2wref **105**
DFACC_ALL **125, 136**
DFACC_CREATE **125, 136**
DFACC_RDWR **125, 136**
DFACC_READ **125, 136**
DFACC_WRITE **125, 136**
DFAN interface
general interface routines **8**
read routines **8**
write routines **8**
DFANaddfds **19**
see also: daafds **19**
DFANaddfid **19, 20**
see also: daafid **20**
DFANgetdesc **21**
see also: dagdesc **21**
DFANgetdesclen **22**
see also: dagdlen **22**
DFANgetfds **23, 24**
see also: dagfds **23**
DFANgetfdslen **24**
see also: dagfdsl **24**
DFANgetfid **25, 26**
see also: dagfid **25**
DFANgetfidlen **26**
see also: dagfidl **26**
DFANgetlabel **27**
see also: daglab **27**
DFANgetlablen **28**
see also: dagllen **28**
DFANgetlablist **47, 76, 99**
DFANlablist **29**
see also: dallist **29**
DFANlastref **30**
see also: dalref **30**
DFANputdesc **31**
see also: dapdesc **31**
DFANputlabel **32**
see also: daplab **32**
DFP interface
general interface routines **5**
read routines **5**
write routines **5**
DFPaddpal **33, 37**
see also: dpapal **33**
DFPgetpal **34, 38, 39**
see also: dpgpal **34**
DFPlastref **35**
see also: dplref **35**
DFPnpals **36**
see also: dpnpals **36**
DFPputpal **37**
see also: dpppal **37**
DFPreadref **38**
see also: dprref **38**
DFPrestart **39**
see also: dprest **39**
DFPwriteref **40**
see also: dpwref **40**
DFR8 interface
general interface routines **4**
read routines **4**
write routines **4**
DFR8addimage **41, 46, 51**
see also: d8aimg **41**
DFR8getdims **42, 43**
see also: d8gdims **42**
DFR8getimage **42, 43, 47**
see also: d8gimg **43**
DFR8lastref **44**
see also: d8lref **44**
DFR8nimages **45**
see also: d8nims **45**
DFR8putimage **41, 46, 51**
see also: d8pimg **46**
DFR8readref **47**
see also: d8rref **47**
DFR8restart **48**
see also: d8first **48**
DFR8setcompress **49**

DFR8setpalette **50**
see also: d8spal **50**
DFR8writeref **51**
see also: d8wref **51**
DFSD interface
data set attribute routines **6**
dimension attribute routines **6**
general interface routines **7**
read and write routines **6**
slab routines **6**
DFSDadddata **53, 73, 79, 82, 87, 91**
see also: dsadata **53**
DFSDclear **54**
see also: dsclear **54**
DFSDendslab **55, 56, 89**
see also: dseslab **55**
DFSDendslice **56, 90**
see also: dseslc **56**
DFSDgetcal **57**
see also: dsgcal **57**
DFSDgetdata **58, 67, 76, 77**
see also: dsgdata **58**
DFSDgetdatalen **59**
see also: dsgdaln **59**
DFSDgetdatastrs **60, 86**
see also: dsgdast **60**
DFSDgetdimlen **61**
see also: dsgdiln **61**
DFSDgetdims **58, 62, 65, 66, 68, 77**
see also: dsgdims **62**
DFSDgetdimscale **63**
see also: dsgdisc **63**
DFSDgetdimstrs **64, 86**
see also: dsgdist **64**
DFSDgetfillvalue **65**
see also: dsgfill **65**
DFSDgetNT **58, 66**
see also: dsgnt **66**
DFSDgetrange
see also: dsgrang **67**
DFSDgetslice **68**
see also: dsgslc **69**
DFSDlastref **70**
see also: dslref **70**
DFSDndatasets **71**
see also: dsnum **71**
DFSDnumber
see: DFSDndatasets **71**
DFSDpre32sdg **72**
see also: dsp32sd **72**
DFSDputdata **73, 79, 82, 87, 91**
see also: dspdata **73**
DFSDputslice **74, 90**
see also: dspslc **75**
DFSDreadref **76**
see also: dsrref **76**
DFSDreadslab **68, 77**
see also: dsrslab **77**
DFSDrestart **58, 78**
see also: dsfirst **78**
DFSDsetcal **79**
see also: dsscal **80**
DFSDsetdatastrs **81**
see also: dssdast **81**
DFSDsetdims **74, 82, 87, 89, 90**
see also: dssdims **82**
DFSDsetdimscale **82, 83**
see also: dssdisc **83**
DFSDsetdimstrs **82, 84**
see also: dssdist **84**
DFSDsetfillvalue **65, 85, 89**
see also: dssfill **85**
DFSDsetlengths **64, 86**
see also: dsslens **86**
DFSDsetNT **53, 82, 87**
see also: dssnt **87**
DFSDsetrange **67, 88**
see also: dssrang **88**
DFSDstartslab **55, 89, 90**
see also: dssslab **89**
DFSDstartslice **90**
see also: dssslc **90**
DFSDwriteref **91**
see also: dswref **91**
DFSDwriteslab **74, 92**
see also: dswslab **92**
dpapal
see: DFPaddpal **33**
dpgpal
see: DFPgetpal **34**
dplref
see: DFPlastref **35**
dpnpals
see: DFPnpals **36**
dpppal
see: DFPputpal **37**
dprest
see: DFPrestart **39**
dprref
see: DFPreadref **38**

dpwref
see: DFPwriteref **40**
dsadata
see: DFSDadddata **53**
dsclear
see: DFSDclear **54**
dseslab
see: DFSDendslab **55**
dseslc
see: DFSDendslice **56**
dsfirst
see: DFSDrestart **78**
dsgcal
see: DFSDgetcal **57**
dsgdaln
see: DFSDgetdatalen **59**
dsgdast
see: DFSDgetdatastrs **60**
dsgdata
see: DFSDgetdata **58**
dsgdiln
see: DFSDgetdimlen **61**
dsgdims
see: DFSDgetdims **62**
dsgdisc
see: DFSDgetdimscale **63**
dsgdist
see: DFSDgetdimstrs **64**
dsgfill
see: DFSDgetfillvalue **65**
dsgnt
see: DFSDgetNT **66**
dsgrang
see: DFSDgetrange **67**
dsgslc
see: DFSDgetslice **68**
dslref
see: DFSDlastref **70**
dsnum
see: DFSDndatasets **71**
dsp32sd
see: DFSDpre32sdg **72**
dspdata
see: DFSDputdata **73**
dspslc
see: DFSDputslice **74**
dsrref
see: DFSDreadref **76**
dsrslab
see: DFSDreadslab **77**
dsscal
see: DFSDsetcal **79**
dssdast
see: DFSDsetdatastrs **81**
dssdims
see: DFSDsetdims **82**
dssdisc
see: DFSDsetdimscale **83**
dssdist
see: DFSDsetdimstrs **84**
dssfill
see: DFSDsetfillvalue **85**
dsslens
see: DFSDsetlengths **86**
dssnt
see: DFSDsetNT **87**
dssrang
see: DFSDsetrange **88**
dsslab
see: DFSDstartslab **89**
dsslc
see: DFSDstartslice **90**
dswref
see: DFSDwriteref **91**
dswslab
see: DFSDwriteslab **92**

F

fp2hdf **237**

G

global attributes
see: SD Interface, general attribute routines

H

H interface
data descriptor routines **2**
file access routines **2**
file maintenance routines **2**
query routines **2**
read and write routines **2**
special data object routines **2**
Happendable **107**
Hclose **19, 20, 108, 111, 174**
see also: hclose **108**
hclose
see: Hclose **108**
Hdeldd **109**

HDF interface
file access routines **3**
hdf24to8 **239**
hdf2jpeg **240**
hdf8to24 **241**
HDFclose **19, 20, 135**
hdfcomp **242**
hdfed **243**
hdfls **246**
HDFopen **19, 20, 23, 24, 25, 26, 136**
hdfpack **109, 247**
hdfrseq **248**
hdftopal **249**
hdftor8 **250**
Hdupdd **110**
HE interface
error reporting routines **3**
HEclear **137**
Hendaccess **108, 111, 119, 120, 124, 129, 134, 135**
HEprint **111, 138**
see also: heprnt **138**
heprnt
see: HEprint **138**
HEpush **137, 139, 140**
HEreport **138, 139, 140**
HEstring **141**
Hfidinquire **112**
Hfind **113**
Hgetelement **114**
Hgetfileversion **115**
Hgetlibversion **116**
High Level Vdata/Vgroup interface
see: VH interface
Hinquire **117, 120, 124**
Hishdf **118**
HLcreate **119**
Hlength **119, 120**
Hnewref **121**
Hnextread **111, 117, 122, 127, 128, 132**
Hnumber **123**
see also: hnumber **123**
Hoffset **119, 124**
Hopen **19, 20, 23, 24, 25, 26, 108, 109, 110, 112, 113, 114, 115, 117, 118, 119, 120, 121, 123, 124, 125, 126, 129, 130, 131, 134, 135, 174, 178, 180, 182, 189, 193, 201, 202, 203, 205, 211, 214, 218**
see also: hopen **125**
Hputelement **126**
Hread **127, 128**
Hseek **127, 128, 133**
Hstartread **111, 117, 120, 122, 124, 127, 128, 129, 132**
Hstartwrite **107, 111, 117, 127, 128, 130, 132, 133**
Hsync **131**
Htrunc **132**
Hwrite **107, 128, 130, 133, 134**
HXcreate **134**

J

jpeg2hdf **251**

L

local attributes
see: SD Interface, general attribute routines
low level interface **1**
see also: H, HDF, HE interface

M

multi-file application interfaces **9**
see also: SD, V, VF, VH, VS, VSQ interface

N

netCDF interface **10**

P

palette
see: DFP interface
paltohdf **252**

R

r24hdf8 **253**
r8tohdf **254**
ristosds **256**

S

scientific data set
see: DFSD, SD interface
SD interface
access routines **9**
dimension attribute routines **9**
general attribute routines **9**
read and write routines **9**
SDattrinfo **143, 149, 161**
see also: sfgainfo **143**
SDcreate **144, 147**
see also: sfcreate **144**

SDdiminfo **145, 153**
see also: sfgdinfo **145**
SDend **146, 174**
see also: sfend **146**
SDendaccess **147**
see also: sfendacc **147**
SDfileinfo **148, 164**
see also: sffinfo **148**
SDfindattr **149**
see also: sffattr **149**
SDgetcal **150**
see also: sfgcal **150**
SDgetdatastrs **151**
see also: sfgdtstr **151**
SDgetdimid **145, 152, 153, 154, 168, 169, 170**
see also: sfdimid **152**
SDgetdimscale **153**
see also: sfgdscale **153**
SDgetdimstrs **154**
see also: sfgdmstr **154**
SDgetfillvalue **155**
see also: sfgfill **155**
SDgetinfo **156**
see also: sfginfo **156**
SDgetrange **157**
see also: sfgrange **157**
SDidtoref **158**
see also: sfid2ref **158**
SDiscoordvar **159, 164**
see also: sfiscvar **159**
SDnametoindex **160, 164**
see also: sfn2index **160**
SDreadattr **143, 149, 161**
see also: sfrattr **161**
SDreaddata **162, 166**
see also: sfrdata **162**
SDreftoindex **163**
see also: sfref2index **163**
SDselect **144, 147, 150, 151, 152, 155, 156, 157, 158, 159, 160, 162, 163, 164, 166, 167, 171, 172, 173, 175**
see also: sfselect **164**
SDsetattr **148, 165**
see also: sfsattr **165**
SDsetcal **166**
see also: sfscal **166**
SDsetdatastrs **151, 167**
see also: sfsdtstr **167**
SDsetdimname **145, 168**
see: sfsdmname **168**
SDsetdimscale **145, 169**
see also: sfsdscale **169**
SDsetdimstrs **154, 170**
see also: sfsdmstr **170**
SDsetexternalfile **171**
SDsetfillvalue **172**
see also: sfsfill **172**
SDsetrange **157, 173**
see also: sfsrange **173**
SDstart **144, 146, 148, 160, 163, 164, 174**
see also: sfstart **174**
SDwritedata **166, 175**
see also: sfwdata **175**
sfcreate
see: SDcreate **144**
sfdimid
see: SDgetdimid **152**
sfend
see: SDend **146**
sfendacc
see: SDendaccess **147**
sffattr
see: SDfindattr **149**
sffinfo
see: SDfileinfo **148**
sfgainfo
see: SDattrinfo **143**
sfgcal
see: SDgetcal **150**
sfgdinfo
see: SDdiminfo **145**
sfgdmstr
see: SDgetdimstrs **154**
sfgdscale
see: SDgetdimscale **153**
sfgdstr
see: SDgetdatastrs **151**
sfgfill
see: SDgetfillvalue **155**
sfginfo
see: SDgetinfo **156**
sfgrange
see: SDgetrange **157**
sfid2ref
see: SDidtoref **158**
sfiscvar
see: SDiscoordvar **159**
sfn2index
see: SDnametoindex **160**
sfrattr
see: SDreadattr **161**

sfreaddata
see: SDreaddata **162**
sfref2index
see: SDreftoindex **163**
sfsattr
see: SDsetattr **165**
sfscal
see: SDsetcal **166**
sfsdmname
see: SDsetdimname **168**
sfsdmstr
see: SDsetdimstrs **170**
sfsdscale
see: SDsetdimscale **169**
sfsdtstr
see: SDsetdatastrs **167**
sfselect
see: SDselect **164**
sfsfill
see: SDsetfillvalue **172**
sfsrange
see: SDsetrange **173**
sfstart
see: SDstart **174**
sfwdata
see: SDwritedata **175**
single file application interfaces **4**
see also: DFR8, DFP, DF24, DFSD, DFAN interface

U

utilities
compression **15**
conversion **15**
inquiry **15**
Vdata/Vgroup **15**

V

V interface
access routines **14**
inquiry routines **14**
specify routines **14**
Vaddtagref **177, 188**
see also: vfadtr **177**
Vattach **177, 178, 179, 181, 183, 184, 185, 186, 187, 188, 190, 191, 192, 201, 202, 203**
see also: vfatch **178**
vcompat **257**
Vdata
see: VF, VH, VSQ, VS interface, utilities
Vdata Fields interface
see: VF interface
Vdata Query interface
see: VSQ interface
Vdetach **178, 179, 201, 202, 203**
see also: vfdtch **179**
Vend **180**
see also: vfend **180**
VF interface
Vdata field query routines **13**
vfadtr
see: Vaddtagref **177**
vfatch
see: Vattach **178**
vfdtch
see: Vdetach **179**
vfend
see: Vend **180**
VFfieldesize **195**
VFfieldisize **196**
VFfieldname **197**
VFfieldorder **198**
VFfieldtype **199**
vfgcls
see: Vgetclass **181**
vfgid
see: Vgetid **182**
vfgnam
see: Vgetname **183**
vfgttr
see: Vgettagref **184**
vfgttrs
see: Vgettagrefs **185**
vfinq
see: Vinquire **187**
vfinqtr
see: Vinqtagref **186**
vfinsrt
see: Vinsert **188**
vflone
see: Vlone **189**
VFnfields **200**
vfntr
see: Vntagrefs **190**
vfscls
see: Vsetclass **191**
vfsnam
see: Vsetname **192**
vfstart
see: Vstart **193**

Vgetclass **181**
see also: vfglcs **181**
Vgetid **182**
see also: vfgid **182**
Vgetname **183**
see also: vfgnam **183**
Vgetnext **205**
Vgettagref **184, 185, 190**
see also: vfgttr **184**
Vgettagrefs **184, 185, 190**
see also: vfgttrs **185**
Vgroups
see: V, VH interface, utilities
VH Interface **15**
VHmakegroup **201**
VHstoredata **202, 226**
VHstoredatam **203**
Vinqtagref **186**
see also: vfinqtr **186**
Vinquire **187**
see also: vfinq **187**
Vinsert **188**
see also: vfinsrt **188**
Vlone **189, 218**
see also: vflone **189**
vmake **258**
Vntagrefs **184, 185, 190**
see also: vfntr **190**
VS interface
access routines **12**
inquiry routines **12**
read and write routines **12**
specify routines **12**
VSattach **205, 206, 207, 208, 210, 212, 213, 215, 216, 217, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233**
see also: vsfatch **205**
VSdetach **205, 206**
see also: vsfdtch **206**
VSelts **207**
see also: vsfelts **207**
Vsetclass **191**
see also: vfscls **191**
Vsetname **192**
see also: vfsnam **192**
vsfatch
see: VSattach **205**
VSfdefine **208, 209, 222**
see also: vsffdef **209**
vsfdtch
see: VSdetach **206**
vsfelts
see: VSelts **207**
vsfex
see: VSfexist **210**
VSfexist **210**
see also: vsfex **210**
vsffdef
see: VSfdefine **208**
vsffnd
see: VSfind **211**
vsfgcls
see: VSgetclass **212**
vsfgfld
see: VSgetfields **213**
vsfgid
see: VSgetid **214**
vsfgint
see: VSgetinterlace **215**
vsfgnam
see: VSgetname **216**
VSfind **211**
see also: vsffnd **211**
vsfinq
see: VSinquire **217**
vsflone
see: VSlone **218**
vsfread
see: VSread **219**
vsfscls
see: VSsetclass **221**
vsfseek
see: VSseek **220**
vsfsfld
see: VSsetfields **222**
vsfsint
see: VSsetinterlace **223**
vsfsiz
see: VSsizeof **225**
vsfsnam
see: VSsetname **224**
vsfwrit
see: VSwrite **226**
VSgetclass **212, 221**
see also: vsfgcls **212**
VSgetfields **213, 217**
see also: vsfgfld **213**
VSgetid **205, 214**
see also: vsfgid **214**

VSgetinterlace **215, 217**
see also: vsfgint **215**
VSgetname **216, 217, 224**
see also: vsfgnam **216**
vshow **260**
VSinquire **217**
see also: vsfinq **217**
VSlone **218**
see also: vsflone **218**
VSQ interface
Vdata query routines **13**
VSQuerycount **225, 227**
VSQueryfields **228**
VSQueryinterlace **229**
VSQueryname **230**
VSQueryref **231**
VSQuerytag **232**
VSread **219, 220, 222**
see also: vsfread **219**
VSseek **219, 220, 226**
see also: vsfseek **220**
VSsetclass **221**
see also: vsfscls **221**
VSsetfields **208, 219, 222, 226**
see also: vsfsfld **222**
VSsetinterlace **205, 223, 226**
see also: vsfsint **223**
VSsetname **224**
see also: vsfsnam **224**
VSsizeof **217, 225**
see also: vsfsiz **225**
Vstart **180, 193, 201, 202, 203**
see also: vfstart **193**
VSwrite **220, 222, 226**
see also: vsfwrit **226**