HDF Reference Manual

Version 4.1r1 • April 1997

Disclaimer

Trademarks

Authors

William Whitehouse - *whitehou@ncsa.uiuc.edu*

Shiming Xu - *sxu@ncsa.uiuc.edu*

Mike Folk - *mfolk@ncsa.uiuc.edu*

NCSA Contacts

Mail user feedback, bug reports, questions, and software and manual suggestions to:

Software Development Division
NCSA HDF
152 Computing Applications Bldg.
605 E. Springfield Ave.
Champaign, IL 61820-5518

Send electronic correspondence and bug reports to the following:

hdfhelp@ncsa.uiuc.edu

Internet Access

HDF is available without charge from HDF Group’s anonymous FTP server

hdf.ncsa.uiuc.edu

It is also accessible through the HDF Group’s World Wide Web home page at

http://hdf.ncsa.uiuc.edu/

Table of Contents

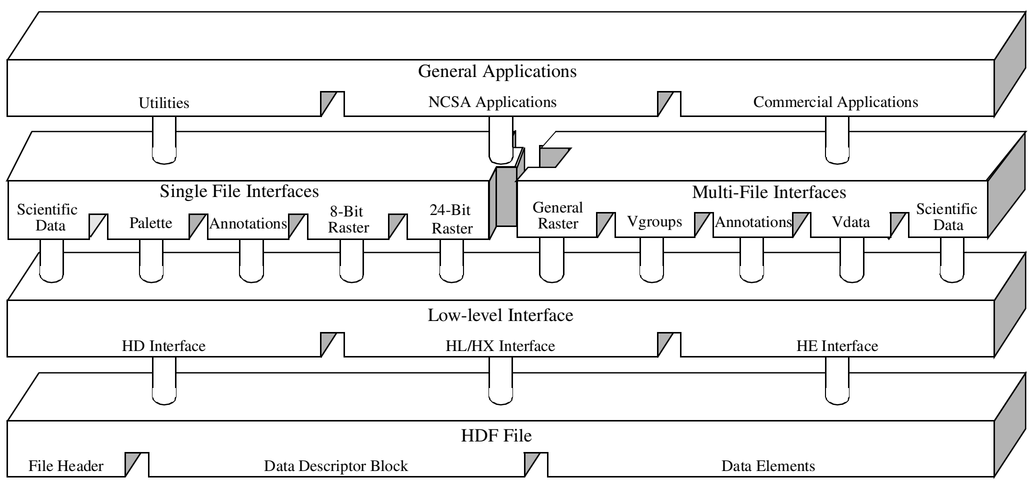
Section 1

Introduction to the HDF APIs

1.1 Overview of the HDF Interfaces

The HDF library structure consists of three interface layers built upon a physical file format. (See Figure 1a.) The first layer, or the *low-level interface*, is generally reserved for software developers because it provides support for low-level details such as file I/O, error handling, and memory management. The second layer, containing the single and multifile *application interfaces*, consists of a set of interfaces designed to simplify the process of storing and accessing data. The single file interfaces are operate on one file at a time, whereas the multifile interfaces can operate on several simultaneously. The highest HDF layer includes a collection of command-line *utilities* that operate on HDF files or the data objects they contain.

FIGURE 1a **The Three Levels of Interaction with the HDF File Format**



1.2 The Low-Level Interface

This is the layer of HDF reserved for software developers and provides routines for error handling, file I/O, memory management, and physical storage. For a more detailed discussion of the low-level interface, consult the *HDF Specification and Developer's Guide.*

1.2.1 The H Interface

The low-level H interface provides a collection of routines, whose names begin with the letter H, for managing HDF files.

Prior to HDF version 3.2, all low-level routines began with the prefix ’DF’. As of HDF version 3.3, the DF interface was no longer recommended for use. It is only supported to maintain backward compatibility with programs and files created under earlier versions of the HDF library.

1.2.2 The HDF Interface

The names of these routines are prefaced by ’HDF’. As HDF begins expanding its interfaces to include mult-file support for each data model, it is anticipated that the some H routines will evolve into an HDF interface. There are only two such routines, **HDFopen** and **HDFclose**. **HDFopen** and **HDFclose** currently operate as macros for **Hopen** and **Hclose** respectively.

1.2.3 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.

1.3 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. 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.

1.3.1 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-77 functions use the prefix ’d8’.

1.3.2 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.

The names of the routines in the palette interface are prefaced by ’DFP’. The equivalent Fortran-77 routine names are prefaced by ’dp’.

1.3.3 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.

The names of the routines in the 24-bit raster interface are prefaced by 'DF24'. The equivalent Fortran-77 routine names are prefaced by 'd2'.

1.3.4 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 multifile SD interface, which permits simultaneous access to more than one file.

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.

The names of the routines in the single-file scientific data set interface are prefaced by 'DFSD'. The equivalent Fortran-77 routine names are prefaced by 'ds'.

1.3.5 Annotations: The DFAN Interface

The single-file 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 null-terminated sequence of characters.

The names of the routines in the single-file annotation interface are prefaced by 'DFAN'. The equivalent Fortran-77 routine names are prefaced by 'da'.

1.4 Multi-File Application Interfaces

The HDF multifile interfaces are designed to allow operations on more than one file and more than one data object at the same time. The multifile interfaces provided are the AN, GR, SD, VS, VSQ, VF, V, and VH interfaces. The AN interface is the multifile version of the DFAN annotation interface. The GR interface is the multifile version of the 8- and 24-bit annotation interfaces. The SD interface is the multifile version of the scientific data set interface. 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 multifile interfaces are built upon the low-level H routines. Unlike single-file operations, operations performed via a multifile 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. This process allows for operations on more than one file at a time.

1.4.1 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. Multidimensional arrays accompanied by a record of their dimension and number type are called scientific data sets. Under the multifile 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.

The names of the routines in the multifile scientific data set interface are prefaced by ’SD’. The equivalent Fortran-77 routine names are prefaced by ’sf’.

1.4.2 Multifile Annotations: The AN Interface

The purpose of the AN multifile annotation interface is to permit concurrent operations on a set of annotations that exist in more than one file. The design of the AN interface is similar to the multifile interfaces for raster image (GR) and scientific data set objects (SD).

The C routine names of the multifile annotation interface are prefaced by the string ’AN’ and the Fortran-77 routine names are prefaced by ’af’.

1.4.3 General Raster Images: The GR Interface

The routines in the GR interface provide for multifile operations on general raster (GR) image data sets.

The C routine names in the general raster interface have the prefix ’GR’ and the equivalent Fortran-77 routine names are prefaced by ’mg’.

1.4.4 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 interface routines and their equivalents in the HDF interface can be found in the User's Guide. Additional information on the netCDF interface can be found in the netCDF User's Guide available by anonymous FTP from `unidata.ucar.edu`.

1.4.5 Vdata: The VS Interface

The VS 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 several vdata field names.

Routines in the VS interface are prefaced by ’VS’. The equivalent Fortran-77 routine names are prefaced by ’vsf’.

1.4.6 Vdata Query: The VSQ Interface

The VSQ 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 routines in the VSQ interface are prefaced by ’VSQ’.

1.4.7 Vdata Fields: The VF Interface

The VF 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 routines in the VF interface are prefaced by 'VF'. There are no equivalent Fortran-77 functions.

1.4.8 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 vdatas, 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 routine name in the vgroup interfaceare prefaced by 'V'. The equivalent Fortran-77 routine names are prefaced by 'vf'.

1.4.9 High-Level Vdata/Vgroups: The VH Interface

The high-level VH interface provides a collection of routines for creating simple vdatas and vgroups with a single function call. All routines in this interface are prefaced by 'VH'.

1.4.10 Vgroup Inquiry: The VQuery Interface

The high-level VQ interface provides one routine that returns tag information from a specified vgroup, and one routine that returns reference number information from a specified vgroup. All C routine names in this interface are prefaced by 'VQuery'.

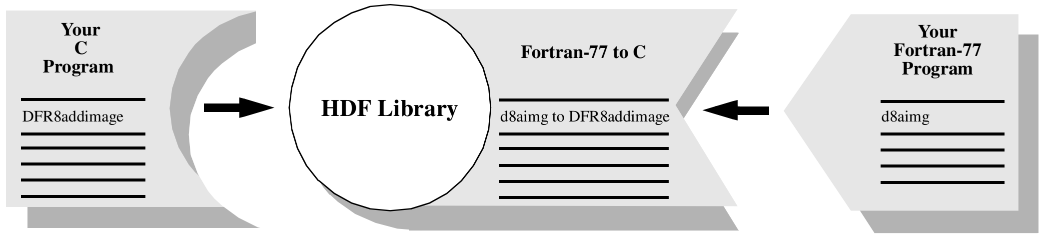
1.5 Fortran-77 and C Language Issues

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

1.5.1 Fortran-77-to-C Translation

Nearly all of the HDF library code is written in C. The Fortran-77 HDF API routines translate all parameter data types to C data types, then call the C routine that performs the main function. For example, **d8aimg** is the Fortran-77 equivalent for **DFR8addimage**. Calls to either routine execute the same C code that adds an 8-bit raster image to an HDF file - see the following figure.

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



1.5.2 Case Sensitivity

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

1.5.3 Name Length

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

1.5.4 Header Files

The inclusion of header files is not generally permitted by Fortran-77 compilers. However, it is sometimes available as an option. On UNIX systems, for example, the macro processors `m4` and `cpp` let your compiler include and preprocess header files. If this capability is not available, you may have to copy whatever declarations, definitions, or values you need from the “constants.f” file into your program code. If it is, include the header file named “hdf.inc” in your Fortran-77 code. The “constants.f” file is included in the “hdf.inc” header file.

1.5.5 Data Type Specifications

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

TABLE 2A **Data Type Definitions**

Data Type	C	Fortran-77
8-bit signed integer	`int8`	Not supported.
8-bit unsigned integer	`uint8`	`character*1`
16-bit signed integer	`int16`	`integer*2`
16-bit unsigned integer	`uint16`	Not supported.
32-bit signed integer	`int32`	`integer*4`
32-bit unsigned integer	`uint32`	Not supported.
32-bit floating point number	`float32`	`real*4`
64-bit floating point number	`float64`	`real*8`

Data Type	C	Fortran-77
Native signed integer	`intn`	`integer`
Native unsigned integer	`uintn`	Not supported.

When using a Fortran-77 data type that is not supported, the general practice is to use another data type of the same size. For example, an 8-bit signed integer can be used to store an 8-bit unsigned integer variable unless the code relies on a sign-specific operation.

1.5.6 Array Specifications

In the declarations contained in the headers of Fortran-77 functions, the following conventions are followed:

- `character* x(*)` means that `x` refers to an array that contains an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold whatever data is stored in the array.
- `real* x(*)` means that `x` refers to an array of reals of indefinite size and of indefinite rank. It is the responsibility of the calling program to allocate an actual array with the correct number of dimensions and dimension sizes.

1.5.7 Fortran-77, ANSI C and K&R C

As much as possible, we have conformed the HDF API routines to those implementations of Fortran and C that are in most common use today, namely Fortran-77, ANSI C and K&R C. Due to the increasing availability of ANSI C, future versions of HDF will no longer support K&R C.

As Fortran-90 is a superset of Fortran-77, HDF programs should compile and run correctly when using a Fortran-90 compiler.

1.6 Error Codes

The error codes defined in the HDF library are defined in the following table.

TABLE 1B **HDF Error Codes**

Error Code	Code Definition
DFE_NONE	No error.
DFE_FNF	File not found.
DFE_DENIED	Access to file denied.
DFE_ALROPEN	File already open.
DFE_TOOMANY	Too many AID's or files open.
DFE_BADNAME	Bad file name on open.
DFE_BADACC	Bad file access mode.
DFE_BADOPEN	Miscellaneous open error.
DFE_NOTOPEN	File can't be closed because it hasn't been opened.
DFE_CANTCLOSE	`fclose` error
DFE_READERROR	Read error.
DFE_WRITEERROR	Write error.
DFE_SEEKERROR	Seek error.

Error Code	Code Definition
DFE_RDONLY	File is read only.
DFE_BADSEEK	Attempt to seek past end of element.
DFE_PUTELEM	`Hputelement` error.
DFE_GETELEM	`Hgetelement` error.
DFE_CANTLINK	Cannot initialize link information.
DFE_CANTSYNC	Cannot syncronize memory with file.
DFE_BADGROUP	Error from `DFdiread` in opening a group.
DFE_GROUPSETUP	Error from `DFdisetup` in opening a group.
DFE_PUTGROUP	Error on putting a tag/reference number pair into a group.
DFE_GROUPWRITE	Error when writing group contents.
DFE_DFNULL	Data file reference is a null pointer.
DFE_ILLTYPE	Data file contains an illegal type: internal error.
DFE_BADDDLIST	The DD list is non-existent: internal error.
DFE_NOTDFFILE	The current file is not an HDF file and it is not zero length.
DFE_SEEDTWICE	The DD list already seeded: internal error.
DFE_NOSUCHTAG	No such tag in the file: search failed.
DFE_NOFREEDD	There are no free DD's left: internal error.
DFE_BADTAG	Illegal WILDCARD tag.
DFE_BADREF	Illegal WILDCARD reference number.
DFE_NOMATCH	No DDs (or no more DDs) that match the specified tag/reference number pair.
DFE_NOTINSET	Warning: Set contained unknown tag. Ignored.
DFE_BADOFFSET	Illegal offset specified.
DFE_CORRUPT	File is corrupted.
DFE_NOREF	No more reference numbers are available.
DFE_DUPDD	The new tag/reference number pair has been allocated.
DFE_CANTMOD	Old element doesn't exisr. Cannot modify.
DFE_DIFFFILES	Attempt to merge objects in different files.
DFE_BADAID	An invalid AID was received.
DFE_OPENAID	Active AIDs still exist.
DFE_CANTFLUSH	Cannot flush DD back to file.
DFE_CANTUPDATE	Cannot update the DD block.
DFE_CANTHASH	Cannot add a DD to the hash table.
DFE_CANTDELDD	Cannot delete a DD in the file.
DFE_CANTDELHASH	Cannot delete a DD from the hash table.
DFE_CANTACCESS	Cannot access specified tag/reference number pair.
DFE_CANTENDACCESS	Cannot end access to data element.
DFE_TABLEFULL	Access table is full.
DFE_NOTINTABLE	Cannot find element in table.
DFE_UNSUPPORTED	Feature not currently supported.
DFE_NOSPACE	`malloc` failed.
DFE_BADCALL	Routine calls were in the wrong order.
DFE_BADPTR	NULL pointer argument was specified.
DFE_BADLEN	Invalid length was specified.
DFE_NOTENOUGH	Not enought space for the data.
DFE_NOVALS	Values were not available.
DFE_ARGS	Invalid arguments passed to the routine.
DFE_INTERNAL	Serious internal error.

Error Code	Code Definition
DFE_NORESET	Too late to modify this value.
DFE_GENAPP	Generic application level error.
DFE_UNINIT	Interface was not initialized correctly.
DFE_CANTINIT	Cannot initialize the interface the operation requires.
DFE_CANTSHUTDOWN	Cannot shut down the interface the operation requires.
DFE_BADDIM	Negative number of dimensions, or zero dimensions, was specified.
DFE_BADFP	File contained an illegal floating point number.
DFE_BADDATATYPE	Unknown or unavailable data type was specified.
DFE_BADMCTYPE	Unknown or unavailable machine type was specified.
DFE_BADNUMTYPE	Unknown or unavailable number type was specified.
DFE_BADORDER	Unknown or illegal array order was specified.
DFE_RANGE	Improper range for attempted access.
DFE_BADCONV	Invalid data type conversion was specified..
DFE_BADTYPE	Incompatible types were specified.
DFE_BADSCHEME	Unknown compression scheme was specified.
DFE_BADMODEL	Invalid compression model was specified.
DFE_BADCODER	Invalid compression encoder was specified.
DFE_MODEL	Error in the modeling layer of the compression operation.
DFE_CODER	Error in the encoding layer of the compression operation.
DFE_CINIT	Error in encoding initialization.
DFE_CDECODE	Error in decoding compressed data.
DFE_CENCODE	Error in encoding compressed data.
DFE_CTERM	Error in encoding termination.
DFE_CSEEK	Error seeking in an encoded dataset.
DFE_MINIT	Error in modeling initialization.
DFE_COMPINFO	Invalid compression header.
DFE_CANTCOMP	Cannot compress an object.
DFE_CANTDECOMP	Cannot decompress an object.
DFE_NODIM	A dimension record was not associated with the image.
DFE_BADRIG	Error processing a RIG.
DFE_RINOTFOUND	Cannot find raster image.
DFE_BADATTR	Invalide attribute.
DFE_BADTABLE	The nsdg table has incorrect information.
DFE_BADSDG	Error in processing an SDG.
DFE_BADNDG	Error in processing an NDG.
DFE_VGSIZE	Too many elements in the vgroup.
DFE_VTAB	Element not in `vtab[]`.
DFE_CANTADDELEM	Cannot add the tag/reference number pair to the vgroup.
DFE_BADVGNAME	Cannot set the vgroup name.
DFE_BADVGCLASS	Cannot set the vgroup class.
DFE_BADFIELDS	Invalid fields string passed to vset routine.
DFE_NOVS	Cannot find the vset in the file.
DFE_SYMSIZE	Too many symbols in the users table.
DFE_BADATTACH	Cannot write to a previously attached vdata.
DFE_BADVSNAME	Cannot set the vdata name.
DFE_BADVSCLASS	Cannot set the vdata class.
DFE_VSWRITE	Error writing to the vdata.

Error Code	Code Definition
DFE_VSREAD	Error reading from the vdata.
DFE_BADVH	Error in the vdata header.
DFE_VSCANTCREATE	Cannot create the vdata.
DFE_VGCANTCREATE	Cannot create the vgroup.
DFE_CANTATTACH	Cannot attach to a vdata or vset.
DFE_CANTDETACH	Cannot detach a vdata or vset with write access.
DFE_BITREAD	A bit read error occurred.
DFE_BITWRITE	A bit write error occurred.
DFE_BITSEEK	A bit seek error occurred.
DFE_TBBTINS	Failed to insert the element into tree.
DFE_BVNEW	Failed to create a bit vector.
DFE_BVSET	Failed when setting a bit in a bit vector.
DFE_BVGET	Failed when getting a bit in a bit vector.
DFE_BVFIND	Failed when finding a bit in a bit vector.

Section 2

HDF Routine Reference

2.1 Reference Section Overview

This section of the Reference Manual contains a listing of every routine contained in the HDF version 4.1r1 library. The pages are organized alphabetically according to the C routine name. Each page addresses one C routine, and the related Fortran-77 routines, and takes the following form:

Routine_Name

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

parameter1	Definition of the first parameter
parameter2	Definition of the second parameter
. . .	
parameterN	Definition of the Nth parameter

Purpose	Section containing a short description of the routine.
Return value	Section describing the return value, if any, and error conditions.
Description	This optional section details any special circumstances surrounding the use of the routine, identifies any routines that must precede it, and describes any known complications related to the use of it.
Example	This optional section consists of one or more examples of the use of the routine. The examples are generally not complete programs are intended for illustrative purposes only. For complete programs, and further descriptions of the routine, refer to the *HDF User's Guide*.
FORTRAN	This section provides a synopsis of the equivalent Fortran-77 routine or routines.

ANannlen/afannlen

int32 ANannlen(int32 *ann_id*)

ann_id IN: Annotation identifier returned by **ANcreate**, **ANcreatef**, or **ANselect**

Purpose Returns the length, in bytes, of the annotation specified by the given annotation identifier.

Return value Returns the length of the target annotation or `FAIL` (or `-1`) otherwise.

Description A common use of **ANannlen** is to determine if a buffer is large enough to contain the entire target annotation.

Example This example illustrates the use of **ANannlen**:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, ann_len, stat;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
ann_id = ANselect(an_id, index, annot_type);
ann_len = ANannlen(ann_id);
  ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afannlen(ann_id)

integer ann_id
```

ANannlist/afannlist

intn ANannlist(int32 *an_id*, ann_type *annot_type,* uint16 *obj_tag,* uint16 *obj_ref,* int32 **ann_list*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
annot_type	IN:	Target annotation type
obj_tag	IN:	Tag of the object the target annotation is attached to
obj_ref	IN:	Reference number of the object the target annotation is attached to
ann_list	OUT:	Buffer for the returned annotation identifiers

Purpose Returns a list of annotation identifiers for the annotations in the file that correspond to the annotation type specified by *annot_type*, the tag specified by *obj_tag* and the reference number specified by *obj_ref*.

Return value Returns the number of qualifying annotations or `FAIL` (or `-1`) otherwise.

Description The identifiers in the list returned by **ANannlist** can refer to either data labels or a data descriptions. The returned identifier list is used in conjunction with **ANnumann** and **HDmalloc** (or **malloc)** to specify the size of a buffer that will be used to store the list of identifiers returned by **ANannlist** - see the example below.

Valid values for *annot_type* are: `AN_DATA_LABEL` for data labels and `AN_DATA_DESC` for data descriptions.

Note that only data labels and data descriptions (identified by the `AN_DATA_LABEL` and `AN_DATA_DESC` definitions) are supported by **ANannlist**. To return file labels or file descriptions use **ANfileinfo** to determine the number of file labels and descriptions, then use **ANselect** with **ANreadann** to read them.

Example This example illustrates the use of **ANannlist** in returning the list of annoations attached to the second numeric data group object in the file:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, *ann_list, stat;
ann_type annot_type = AN_DATA_LABEL;
uint16 obj_tag = DFTAG_NDG;
uint16 obj_ref = 2;
int num_ann;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
num_ann = ANnumann(an_id, annot_type, obj_tag, obj_ref);
ann_list = HDmalloc(num_ann * sizeof(int32));
num_ann = ANannlist(an_id, annot_type, obj_tag, obj_ref,
```

```
                            ann_list);
  ...
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afannlist(an_id, annot_type, obj_tag, obj_ref
    ann_list)

integer ann_list(*)
integer an_id, obj_tag, obj_ref, annot_type
```

ANatype2tag/afatypetag

uint16 ANatype2tag(int32 **ann_type*)

ann_type	IN:	Annotation type

Purpose Returns the tag (prefaced by `DFTAG_`) corresponding to the specified annotation type (prefaced by `AN_`).

Return value A tag: `DFTAG_FID`, `DFTAG_FD`, `DFTAG_DIL`, `DFTAG_DIA`.

FORTRAN

```
integer function afatypetag(ann_type)

integer ann_type
```

ANcreate/afcreate

int32 ANcreate(int32 *an_id,* uint16 *tag,* uint16 *ref,* int32 *annot_type*)

an_id	IN:	Multifile data annotation interface identifier returned by **ANstart**
tag	IN:	Tag of the element the created data annotation will be applied to
ref	IN:	Reference number of the object the created data annotation will be applied to
annot_type	IN:	Type of data annotation

Purpose Creates an data annotation for the object identified by the specified tag and reference number.

Return value Returns a data annotation identifier if successful and `FAIL` (or `-1`) otherwise.

Description The data annotation identifier returned can either point to a data label or a data description.

Valid values for *annot_type* are: `AN_DATA_LABEL` for data labels and `AN_DATA_DESC` for data descriptions.

Example This example illustrates the use of **ANcreate** to create an data label for a numeric data group.

```
#include "hdf.h"

int32 an_id, ann_id, file_id, stat;
uint16 tag = DFTAG_NDG;
uint16 ref = 2;
ann_type type = AN_DATA_LABEL;

file_id = Hopen("myfile", DFACC_WRITE, 0);
an_id = ANstart(file_id);
ann_id = ANcreate(an_id, tag, ref, type);
  ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afcreate(an_id, tag, ref, annot_type)

integer an_id, tag, ref, annot_type
```

ANcreatef/affcreate

int32 ANcreatef(int32 *an_id,* int32 *annot_type*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
annot_type	IN:	Type of file annotation

Purpose Creates an file label or file description annotation .

Return value Returns a file annotation identifier if successful and `FAIL` (or `-1`) otherwise.

Valid values for *annot_type* are: `AN_FILE_LABEL` for file labels and `AN_FILE_DESC` for file descriptions.

FORTRAN

```
integer function affcreate(an_id, type)

integer an_id, type
```

ANdestroy

intn ANdestroy(void)

Purpose	Deallocates all internal data structures used by the multifile annotation interface.
Return value	Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

ANend/afend

intn ANend(int32 *an_id*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**

Purpose Terminates access to the multifile annotation interface.

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

Description This routine is used with the **ANstart** routine to define the extent of a multifile annotation interface session. As in other multifile interfaces, **ANend** disposes of the internal structures used by the remaining AN routines.

Example This example illustrates the use of **ANend**:

```
#include "hdf.h"

int32 an_id, file_id, stat;

file_id = Hopen("myfile", DFACC_WRITE, 0);
an_id = ANstart(file_id);
  ...
Hclose(file_id);
```

FORTRAN

```
integer function afend(an_id)

integer an_id
```

ANendaccess/afendaccess

intn ANendaccess(int32 *ann_id*)

ann_id IN: Annotation identifier returned by **ANcreate**, **ANcreatef** or **ANselect**

Purpose Terminates access to an annotation.

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

Description There should be one call to **ANendaccess** for every call to **ANselect**, **ANcreate** or **ANcreatef**.

Example This example illustrates the use of **ANendaccess**:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, stat;
 uint16 tag = DFTAG_DIL;
 uint16 ref = 0;
 ann_type type = AN_DATA_LABEL;

file_id = Hopen("myfile", DFACC_WRITE, 0);
an_id = ANstart(file_id);
 ann_id = ANcreate(an_id, tag, ref, type);
   ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afendaccess(ann_id)

integer ann_id
```

ANfileinfo/affileinfo

intn ANfileinfo(int32 *an_id*, int32 *n*_*file_label,* int32 **n_file_desc,* int32 **n_data_label,* int32 **n_data_desc*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
n_file_label	OUT:	Returned number of file labels
n_file_desc	OUT:	Returned number of file descriptions
n_data_label	OUT:	Returned total number of data labels
n_data_desc	OUT:	Returned total number of data descriptions

Purpose Returns the number of annotations of each type in the current file.

Return value Returns `SUCCEED` (or `0`) on successful completion or `FAIL` (or `-1`) otherwise.

Description This routine is generally used to find the range of acceptable indices for **ANselect** calls.

Note that the number of data labels and data descriptions (returned in the *n_data_label* and *n_data_desc* parameters) refer to the total number in the file, not the total number for a specific element. Use **ANnumann** to determine this number for a specific element.

Example This example illustrates the use of **ANfileinfo**:

```
#include "hdf.h"

int32 an_id, file_id, stat;
int32 n_data_label, n_data_desc, n_file_label, n_file_desc;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
stat = ANfileinfo(an_id, &n_file_label, &n_file_desc,
      &n_data_label, &n_data_desc);
  ...
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function affileinfo(an_id, n_file_label,
     n_file_desc, n_data_label, n_data_desc)

integer an_id, n_file_label, n_file_desc
integer n_data_label, n_data_desc
```

ANget_tagref/afgettagref

int32 ANget_tagref(int32 *an_id*, int32 **index*, int32 **type*, uint16 **ann_tag*, uint16 **ann_ref*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
index	IN:	Index of the annotation
type	IN:	Annotation type
ann_tag	OUT:	Tag of the annotation
ann_ref	OUT:	Reference number of the annotation

Purpose Returns the tag/reference number pair of the specified annotation.

Return value Returns the tag/reference number pair if successful or `FAIL` (or `-1`) otherwise.

Description The *index* parameter is zero-based.

Valid values for the *type* parameter are:

```
AN_DATA_LABEL - for data labels

AN_DATA_DESC - for data descriptions

AN_FILE_LABEL - for file labels

AN_FILE_DESC - for file descriptions
```

FORTRAN

```
integer function afgettagref(an_id, index, type,
     ann_tag, ann_ref)

integer an_id, index, type
integer ann_tag, ann_ref
```

ANid2tagref/afidtagref

int32 ANid2tagref(int32 *ann_id*, uint16 **ann_tag*, uint16 **obj_ref*)

ann_id	IN:	Annotation identifier returned by **ANselect**, **ANcreate** or **ANcreatef**
ann_tag	OUT:	Tag of the annotation
obj_ref	OUT:	Reference number of the annotation

Purpose Returns the tag/reference number pair of the specified annotation.

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

FORTRAN

```
integer function afidtagref(ann_id, ann_tag, obj_ref)

integer ann_id, ann_tag, obj_ref
```

ANnumann/afnumann

intn ANnumann(int32 *an_id*, int32 *annot_type,* uint16 *obj_tag,* uint16 *obj_ref*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
annot_type	IN:	Target annotation type
obj_tag	IN:	Tag of the object the target annotation is attached to
obj_ref	IN:	Reference number of the object the target annotation is attached to

Purpose Returns the total number of annotations in the file that correspond to the annotation type specified by *annot_type*, the tag specified by *obj_tag* and the reference number specified by *obj_ref*.

Return value Returns the number of qualifying annotations or `FAIL` (or `-1`) otherwise.

Description The annotations referred to by the return value of **ANnumann** can be labels and/or descriptions. The return value is often used in conjunction with **HDmalloc** or **malloc** to specify the size of a buffer that will be used to store information about the target annotations, or in conjunction with **ANannlist**.

Valid values for the *annot_type* parameter are:

`AN_DATA_LABEL - for data labels`

`AN_DATA_DESC - for data descriptions`

Note that only data labels and data descriptions (identified by the `AN_DATA_LABEL` and `AN_DATA_DESC` definitions) are supported by **ANnumann**. Use **ANfileinfo** to set file labels and file descriptions.

Example This example illustrates the use of **ANnumann** in returning the number of annotations attached to the second number data group object in the file:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, stat;
 ann_type annot_type = AN_DATA_LABEL;
 uint16 obj_tag = DFTAG_NDG;
 uint16 obj_ref = 2;
 int num_ann;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
num_ann = ANnumann(an_id, annot_type, obj_tag, obj_ref);
  ...
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN `integer function afnumann(an_id, annot_type, obj_tag, obj_ref)`

```
integer an_id, obj_tag, obj_ref, annot_type
```

ANreadann/afreadann

intn ANreadann(int32 *ann_id*, char* *textbuf*, int32 *textbuf_length*)

ann_id	IN:	Annotation identifier returned by **ANcreate**, **ANcreatef** or **ANselect**
textbuf	OUT:	Buffer for the returned annotation text
textbuf_length	IN:	Length, in bytes, of *textbuf*

Purpose Reads the annotation identified by the annotation identifier *ann_id*.

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

Description **ANreadann** reads the annotation specified by *ann_id* and places it into *textbuf*. The *textbuf_length* parameter is the size of *textbuf*. A null termination is added to the buffer as necessary - therefore, the buffer should be sized to accomodate this null termination.

Example This example illustrates the use of **ANreadann** using a statically-sized buffer:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, stat;
int32 index = 0;
char textbuf[20];
ann_type annot_type = AN_FILE_DESC;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
ann_id = ANselect(an_id, index, annot_type);
   ...
/* Read a 17 character annotation text string, */
/*  plus a null termination cha                */
stat = ANreadann(ann_id, textbuf, 18);
   ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afreadann(ann_id, buf, buf_length)

integer ann_id, buf_length
character* (*) buf
```

ANselect/afselect

int32 ANselect(int32 *an_id*, int32 *index*, int32 *annot_type*)

an_id	IN:	Multifile annotation interface identifier returned by **ANstart**
index	IN:	Location of the annotation in the file
annot_type	IN:	Annotation type

Purpose Selects and returns the identifier for the annotation identified by the index *index* and the annotation type *annot_type*.

Return value Returns the identifier of the selected annotation.

Description The identifier returned by **ANselect** can refer to either a label or a description. The index supplied by the parameter *index* is zero-based.

Valid values for the *annot_type* parameter are:

```
AN_DATA_LABEL - for data labels

AN_DATA_DESC - for data descriptions

AN_DATA_LABEL - for data labels

AN_DATA_DESC - for data descriptions
```

Example This example illustrates the use of **ANselect** in returning the fourth file label in a file:

```
#include "hdf.h"

int32 an_id, ann_id, file_id, stat;
ann_type annot_type = AN_FILE_LABEL;
int32 index = 4;

file_id = Hopen("myfile", DFACC_READ, 0);
an_id = ANstart(file_id);
ann_id = ANselect(an_id, index, annot_type);
  ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afselect(an_id, index, annot_type)

integer an_id, index
integer annot_type
```

ANstart/afstart

int32 ANstart(int32 *file_id*)

file_id IN: File identifier returned by **Hopen**

Purpose Initializes the multifile annotation interface.

Return value Returns an interface identifier if successful and `FAIL` (or `-1`) otherwise.

Description This routine is used with the **ANend** routine to define the extent of a multifile annotation interface session. **ANstart** initializes the internal interface structures needed for the remaining AN routines. Use the general purpose routines **Hopen** and **Hclose** to manage file access as the AN routines will not open and close HDF files.

Example This example illustrates the use of **ANstart** and **ANend** in initializing and terminating an AN interface session.

```
#include "hdf.h"
int32 an_id, file_id, stat;

file_id = Hopen("myfile", DFACC_WRITE, 0);
an_id = ANstart(file_id);
  ...
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afstart(file_id)

integer file_id
```

ANtag2atype/aftagatype

int32 ANtag2atype(uint16 *tag*)

tag	IN:	Object tag

Purpose Returns the annotation type (prefaced by `AN_`) corresponding to the specified tag (prefaced by `DFTAG_`).

Return value Annotation type

Description Valid values for the *annot_type* parameter are:

```
AN_DATA_LABEL - for data labels
AN_DATA_DESC - for data descriptions
AN_DATA_LABEL - for data labels
AN_DATA_DESC - for data descriptions
```

FORTRAN

```
integer function aftagatype(tag)
integer tag
```

ANtagref2id/aftagrefid

int32 ANtagref2id(int32 *an_id*, uint16 **tag*, uint16 *ref*)

an_id	IN:	Annotation interface identifier
tag	IN:	Tag of the annotation
ref	IN:	Reference number of the annotation

Purpose Retrieves an annotation for a specified tag/reference number pair.

Return value Returns the identifier of the annotation if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function aftagrefid(an_id, tag, ref)

integer an_id, tag, ref
```

ANwriteann/afwriteann

intn ANwriteann(int32 *ann_id*, char* *label*, int32 *ann_length*)

ann_id	IN:	Annotation identifier returned by **ANcreate**, **ANcreatef** or **ANselect**
label	IN:	Text to be written to the annotation
ann_length	IN:	Length, in bytes, of the annotation text pointed to by *label*

Purpose Writes an annotation to the current HDF file.

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

Description **ANwriteann** will overwrite any preexisting annotation containing the text identified by *label*.

Example This example illustrates the use of **ANwriteann**:

```
#include "hdf.h"

int32 an_id, stat;
 char *label = "This is annotation text.";

file_id = Hopen("myfile", DFACC_WRITE, 0);
an_id = ANstart(file_id);
 ann_id = ANcreate(an_id, tag, ref, type);
   ...
stat = ANwriteann(ann_id, label, sizeof(label));
   ...
stat = ANendaccess(ann_id);
stat = ANend(an_id);
Hclose(file_id);
```

FORTRAN

```
integer function afwriteann(ann_id, label, ann_length)

integer ann_id, ann_length
character* (*) label
```

d2scomp

integer d2scomp(integer *quality*, integer *baseline*)

quality	IN:	JPEG quality specification
baseline	IN:	JPEG baseline specification

Purpose Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

Description **d8sjpeg** changes the JPEG compression parameter settings set in the **d8scomp** routine.

d2sjpeg

integer d2sjpeg(integer *quality*, integer *baseline*)

quality	IN:	JPEG quality specification
baseline	IN:	JPEG baseline specification

Purpose Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

Description **d2sjpeg** changes the JPEG compression parameter settings set in the **d2scomp** routine.

d8scomp

integer d8scomp(integer *quality*, integer *baseline*)

quality	IN:	JPEG quality specification
baseline	IN:	JPEG baseline specification

Purpose Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

Description **d8sjpeg** changes the JPEG compression parameter settings set in the **d8scomp** routine.

d8sjpeg

integer d8sjpeg(integer *quality*, integer *baseline*)

quality	IN:	JPEG quality specification
baseline	IN:	JPEG baseline specification

Purpose Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

Description **d8sjpeg** changes the JPEG compression parameter settings set in the **d8scomp** routine.

DF24addimage/d2aimg

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

filename	IN:	Name of the 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 Writes a 24-bit image to the specified file.

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

Description **DF24addimage** appends a 24-bit raster image set to the file. Array *image* is assumed to be width *x* height *x* 3 bytes. In Fortran-77, the dimensions of the array *image* must be the same as the dimensions of the image data.

The order in which dimensions are declared is different between C and Fortran-77. Ordering varies because Fortran-77 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 **DF24addimage** writes an image to a file, it assumes row-major order. The Fortran-77 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. In other words, the image must be built "on its side".

FORTRAN

```
integer function d2aimg(filename, image, width, height)

character* (*) filename
<valid numeric data type> image
integer width, height
```

DF24getdims/d2gdims

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

filename	IN:	Name of the file
width	OUT:	Width of the image
height	OUT:	Height of the image
il	OUT:	File interlace mode of the image

Purpose Retrieves dimensions and interlace storage scheme of next image.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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. In this way, images are read in the same order in which they were written to the file.

If the dimensions and interlace of the image are known beforehand, there is no need to call **DF24getdims**. Simply allocate arrays with the proper dimensions for the image and invoke **DF24getimage** to read 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 amount of memory to allocate for the image buffer.

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

The interlace mode codes are: `0` for pixel interlacing, `1` for scan-line interlacing and `2` for scan-plane interlacing.

FORTRAN

```
integer function d2gdims(filename, width, height, il)

character* (*) filename
integer width, height, il
```

DF24getimage/d2gimg

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

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

Purpose Retrieves an image from the next 24-bit raster image set.

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

Description **DF24getimage** retrieves the image and stores it in an array. If **DF24getdims** has not been called, **DF24getimage** finds the next image in the same way that **DF24getdims** 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, first call **DF24reqil**.

FORTRAN

```
integer function d2gimg(filename, image, width, height)

character* (*) filename, image
integer width, height
```

DF24lastref/d2lref

uint16 DF24lastref( )

Purpose Retrieves the last reference number written to or read from a 24-bit raster image set.

Return value Returns the non-zero reference number if successful and `FAIL` (or `-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 the last 24-bit raster image read or written.

FORTRAN `integer function d2lref( )`

DF24nimages/d2nimg

intn DF24nimages(const char **filename*)

filename	IN:	Name of the file

Purpose Counts the number of 24-bit raster images contained in an HDF file.

Return value Returns the number of 24-bit images in the file if successful and `FAIL` (or `-1`) otherwise.

Description **DF24nimages** counts the number of 24-bit images stored in the file.

FORTRAN

```
integer function d2nimg(filename)

character* (*) filename
```

DF24putimage/d2pimg

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

filename	IN:	Name of the 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 Writes a 24-bit image as the first image in the file.

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

Description 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 numeric data type> image
integer width, height
```

DF24readref/d2rref

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

filename	IN:	Name of the file
ref	IN:	Reference number for the next call to **DF24getimage**

Purpose Specifies the reference number of the next image to be read when **DF24getimage** is next called.

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

Description **DF24readref** is commonly used in conjunction with **DFANlablist**, which returns a list of labels for a given tag together with their reference numbers. It provides a means of non-sequentially accessing 24-bit raster images in a file.

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 index of an image.

FORTRAN

```
integer function d2rref(filename, ref)

character* (*) filename
integer ref
```

DF24reqil/d2reqil

intn DF24reqil (intn *il*)

il	IN	Memory interlace of the next image read

Purpose Specifies the interlace mode for the next call to **DF24getimage** will use.

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

Description 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*.

Because a call to **DF24reqil** may require a substantial reordering of the data, slower I/O performance could result than would be achieved if no change in interlace were requested.

The interlace mode codes are: `0` for pixel interlacing,`1` for scan-line interlacing and `2` for scan-plane interlacing.

FORTRAN

```
integer function d2reqil(il)

integer il
```

DF24restart/d2first

intn DF24restart( )

Purpose Specifies that the next 24-bit image read from the file will be the first one rather than the 24-bit image following the one most recently read.

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

FORTRAN `integer function d2first( )`

DF24setcompress/d2scomp

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` (or `0`) if successful and `FAIL` (or `-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 `comp_info` union contains algorithm-specific information for the library routines that perform the compression and is defined in the `hcomp.h` header file as follows:

```
typedef union tag_comp_info
  {
      struct
       {
            intn    quality;
            intn    force_baseline;
        }
      jpeg;
      struct
        {
            int32   nt;
            intn    sign_ext;
            intn    fill_one;
            intn    start_bit;
            intn    bit_len;
        }
      nbit;
      struct
        {
            intn    skp_size;
        }
      skphuff;
      struct
        {
            intn    level;
        }
      deflate;
  }
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.

If the compression type is JPEG, **d2scomp** defines the default JPEG compression parameters to be used. If these parameters must be changed later, the **d2sjpeg** routine must be used. (See the Reference Manual entry for **d2sjpeg**)

FORTRAN

```
integer function d2scomp(type)

integer type
```

DF24setdims/d2sdims

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 of the next image to be written to a file.

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

FORTRAN

```
integer function d2sdims(width, height)

integer width, height
```

DF24setil/d2setil

intn DF24setil(intn *il*)

il	IN:	Interlace mode

Purpose Specifies the interlace mode to be used on subsequent writes.

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

Description **DF24setil** sets the interlace mode to be used when writing out the raster image set for a 24-bit image by determining the interlace mode of the image data in memory. If **DF24setil** is not called, the interlace mode is assumed to be `0`.

The interlace mode codes are: `0` for pixel interlacing, `1` for scan-line interlacing and `2` for scan-plane interlacing.

FORTRAN

```
integer function d2setil(il)

integer il
```

DFANaddfds/daafds

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

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

Purpose Adds a file description to a file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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** and **Hclose** to manage file access as the file annotation routines will not open and close HDF files.

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

```
#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 file_id, desc_len
character* (*) description
```

DFANaddfid/daafid

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

file_id	IN:	The file identifier returned by **Hopen.**
label	IN:	A null-terminated string.

Purpose Writes a file label to a file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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** and **Hclose** to manage file access because the file annotation routines will not open and close HDF files for you.

In the Fortran-77 version, the string length for the label should be close to the actual expected string length, because in Fortran-77 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 file_id
character* (*) label
```

DFANclear/daclear

intn DFANclear( )

Purpose Resets all internal library structures and parameters of the DFAN annotiation interface.

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

Description When a file is regenerated in a single run by a library routine of another interface (such as **DFSDputdata**), **DFANclear** should be called to reset the interface.

FORTRAN `integer function daclear( )`

DFANgetdesc/dagdesc

intn DFANgetdesc(char **filename*, uint16 *tag*, uint16 *ref*, char **desc_buf*, int32 *buf_len*)

filename	IN:	Name of the file
tag	IN:	Tag of the data object assigned the description
ref	IN:	Reference number of the data object assigned the description
desc_buf	OUT:	Buffer allocated to hold the description
buf_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` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

Description The parameter *buf_len* specifies the storage space available for the description. The length of *buf_len* must account for the null termination character appended to the description.

FORTRAN

```
integer function dagdesc(filename, tag, ref, desc_buf,
    buf_len)

character* (*) filename, desc_buf
integer tag, ref
integer buf_len
```

DFANgetdesclen/dagdlen

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

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

Purpose Retrieves 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` (or `-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 tag, ref
```

DFANgetfds/dagfds

int32 DFANgetfds(int32 *file_id*, char **desc_buf*, int32 *buf_len*, intn *isfirst*)

file_id	IN:	File identifier returned by **Hopen**
desc_buf	OUT:	The buffer allocated to hold the description
buf_len	IN:	Size of the buffer allocated to hold the description
isfirst	IN:	Determines the description to be retrieved

Purpose Reads the next file description.

Return value Returns the length of the file description if successful and `FAIL` (or `-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.

V.alid values for *isfirst* are: `1` to read the first description and `0` to read the next description.

FORTRAN

```
integer function dagfds(file_id, desc_buf, buf_len, isfirst)

integer file_id, buf_len, isfirst
character* (*) desc_buf
```

DFANgetfdslen/dagfdsl

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

file_id	IN:	File identifier returned by **Hopen**
isfirst	IN:	Determines the description the retrieved length information applies to

Purpose Returns the length of a file description.

Return value Returns the length of the file description if successful and `FAIL` (or `-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.

V.alid values for *isfirst* are: `1` to read the length of the first description and `0` to read the length of the next description.

FORTRAN

```
integer function dagfdsl(file_id, isfirst)

integer file_id, isfirst
```

DFANgetfid/dagfid

int32 DFANgetfid(int32 *file_id*, char **desc_buf,* int32 *buf_len*, intn *isfirst*)

file_id	IN:	File identifier returned by **Hopen**
label_buf	OUT:	The buffer allocated to hold the label
buf_len	IN:	Size of the buffer allocated to hold the label
isfirst	IN:	Determines the file label to be retrieved

Purpose: Reads a file label from a file.

Return value: Returns the length of the file description if successful and `FAIL` (or `-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 *buf_len* is not large enough, the label is truncated to *buf_len*`-1` characters in the buffer *label_buf*.

Valid values of *isfirst* are: `1` to read the first label, `0` to read the next label

FORTRAN

```
integer function dagfid(file_id, label_buf, buf_len, isfirst)

integer file_id, buf_len, isfirst
character* (*) label_buf
```

DFANgetfidlen/dagfidl

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

file_id	IN:	File identifier returned by **Hopen**
isfirst	IN:	Determines the file label the retrieved length information applies to

Purpose Returns the length of a file label.

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

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

Valid values of *isfirst* are: `1` to read the first label, and `0` to read the next label.

FORTRAN

```
integer function dagfidl(file_id, isfirst)

integer file_id, isfirst
```

DFANgetlabel/daglab

intn DFANgetlabel(char **filename*, uint16 *tag*, uint16 *ref*, char **label_buf*, int32 *buf_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_buf	OUT:	Buffer for the label
buf_len	IN:	Size of the buffer allocated for the label

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

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

Description The parameter *buf_len* specifies the storage space available for the label. The length of *buf_len* must account for the null termination character appended to the annotation.

FORTRAN

```
integer function daglab(filename, tag, ref, label_buf,
                        buf_len)

character* (*) filename, label_buf
integer tag, ref, buf_len
```

DFANgetlablen/dagllen

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

filename	IN:	Name of the 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` (or `-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 tag, ref
```

DFANlablist/dallist

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 file
tag	IN:	Tag to be queried
ref_list	OUT:	Buffer for the returned reference numbers
label_list	OUT:	Buffer for the returned labels
list_len	IN:	Size of the reference number list and the label list
label_len	IN:	Maximum length allowed for a label
start_pos	IN:	Starting position of the search

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` (or `-1`) otherwise.

Description Entries are returned from the *start_pos* entry up to the *list_len* entry.

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, if any. 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 *label_list* parameter can display all of the labels for a given tag. Or it can be searched to find the reference number of a data object with a certain label. Once the reference number for a given label is found, the corresponding data object can be accessed by invoking other HDF routines. Therefore, this routine provides a mechanism for the 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 ref_list(*)
integer list_len, label_len, start_pos
```

DFANlastref/dalref

uint16 DFANlastref( )

Purpose Returns the reference number of the annotation last written or read.

Return value Returns the reference number if successful and `FAIL` (or `-1`) otherwise.

FORTRAN `integer function dalref( )`

DFANputdesc/dapdesc

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

filename	IN:	Name of the 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` or '`\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` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

Description The parameter description can contain any sequence of ASCII characters; it does not have to be a string. If **DFANputdesc** is called more than once for the same tag/reference number pair, only the last description is stored in the file.

FORTRAN

```
integer function dapdesc(filename, tag, ref, description,
                              desc_len)

character* (*) filename, description
integer tag, ref, desc_len
```

DFANputlabel/daplab

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

filename	IN:	Name of the 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:	Null-terminated label string

Purpose Assigns a label to the data object with the given tag/reference number pair.

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

FORTRAN

```
integer function daplab(filename, tag, ref, label)

character* (*) filename, label
integer tag, ref
```

DFKNTsize

int DFKNTsize(int32 *data_type*)

data_type	IN:	Data type

Purpose Determines the size of the specified data type.

Return value Returns the size, in bytes, of the specified data type if successful and `FAIL` (or `-1`) otherwise.

DFPaddpal/dpapal

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

filename	IN:	Name of the HDF file
palette	IN:	Buffer containing the palette to be written

Purpose Appends a palette to a file.

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

Description If the named file does not exist, it is created and the palette written to it. The *palette* buffer should beat least 768 bytes in length.

FORTRAN

```
integer function dpapal(filename, palette)

character* (*) filename, palette
```

DFPgetpal/dpgpal

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

filename	IN:	Name of the HDF file
palette	OUT:	Buffer for the returned palette

Purpose Retrieves the next palette from file and stores it in the buffer *palette*.

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

Description The *palette* buffer is assumed to be at least 768 bytes long. Successive calls to **DFPgetpal** retrieve the palettes in the sequence they are stored in the file.

FORTRAN

```
integer function dpgpal(filename, palette)

character* (*) filename. palette
```

DFPlastref/dplref

uint16 DFPlastref(void)

Purpose Returns 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` (or `-1`) otherwise.

FORTRAN `integer function dplref( )`

DFPnpals/dpnpals

intn DFPnpals(const char **filename*)

filename	IN:	Name of the file

Purpose Indicates the number of palettes in the specified file.

Return value Returns the number of palettes if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function dpnpals(filename)

character* (*) filename
```

DFPputpal/dpppal

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

filename	IN:	Name of the file
palette	IN:	Buffer containing the palette to be written
overwrite	IN:	Flag identifying the palette to be written
filemode	IN:	File access mode

Purpose Writes a palette to the file.

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

Description This routine provides more controlof palette write operations than **DFPaddpal**. Note that the combination *filemode*="w" and *overwrite*=`1` has no meaning and will result in an error condition. To overwrite a palette, *filename* must be the same filename as the last file accessed through the DFP interface.

Valid values for *overwrite* are: `1` to overwrite last palette; `0` to write a new palette.

Valid values for *filemode* are: "a" to append the palette to the file and "w" to create a new file.

The *palette* buffer must be at least 768 bytes in length.

FORTRAN

```
integer function dpppal(filename, palette, overwrite,
                        filemode)

character* (*) filename, palette, filemode
integer overwrite
```

DFPreadref/dprref

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

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

Purpose Retrieves the reference number of the palette to be retrieved next by **DFPgetpal**.

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

Description Used to set the reference number of the next palette to be retrieved.

FORTRAN

```
integer function dprref(filename, ref)

character* (*) filename
integer ref
```

DFPrestart/dprest

intn DFPrestart( )

Purpose Specifies that **DFPgetpal** will read the first palette in the file, rather than the next unread palette.

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

FORTRAN `integer function dprest( )`

DFPwriteref/dpwref

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

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

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

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

Description The file name 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 ref
```

DFR8addimage/d8aimg

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

filename	IN:	Name of the file
image	IN:	Array containing the image data
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` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

Description **DFR8addimage** is functionally equivalent to **DFR8putimage**, except that **DFR8putimage** cannot append image data - it only overwrites.

FORTRAN

```
integer function d8aimg(filename, image, width, height,
    compress)

character* (*) filename, image
integer width, height
integer compress
```

DFR8getdims/d8gdims

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

filename	IN:	Name of the HDF file
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:	Indicator of the existence of a palette

Purpose Opens the file, finds the next image, retrieves the dimensions of the image, and determines whether there is a palette associated with the image.

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

Valid values of *ispalette* are: `1` if there is a palette, or `0` if not.

FORTRAN

```
integer function d8gdims(filename, width, height, ispalette)

character* (*) filename
integer width, height
integer ispalette
```

DFR8getimage/d8gimg

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

filename	IN:	Name of the file
image	OUT:	Buffer for the returned image
width	IN:	Width of the image data buffer
height	IN:	Height of the image data buffer
palette	OUT:	Palette data

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

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

Description In C, if *palette* is `NULL`, no palette is loaded, even if one is stored with the image. In Fortran-77, an array must be allocated to store the palette, even if no palette is expected to be stored. 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 *width* and *height* parameters specify the number of columns and rows, respectively, in the array which you've allocated in memory to store the image. The image may be smaller than the allocated space.

The order in which you declare dimensions is different between C and Fortran-77. Ordering varies because Fortran-77 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-77 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, the image must be built "on its side".

FORTRAN

```
integer function d8gimg(filename, image, width, height,
    palette)

character* (*) filename, image, palette
integer width, height
```

DFR8getpalref

intn DFR8getpalref(uint16 **pal_ref*)

pal_ref	OUT: Reference number of the palette

Purpose Retrieves the reference number of the palette associated with the last image accessed.

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

Description Make certain that **DFR8getdims** is called before **DFR8getpalref**.

DFR8lastref/d8lref

uint16 DFR8lastref( )

Purpose Retrieves the last reference number written to or read froman RIS8.

Return value Returns a non-zero reference number if successful and `FAIL` (or `-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/d8nims

intn DFR8nimages(char **filename*)

filename	IN:	Name of the HDF file

Purpose Retrieves the number of 8-bit raster images stored in the specified file.

Return value Returns the number of raster images in the file if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function d8nims(filename)

character* (*) filename
```

DFR8putimage/d8pimg

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

filename	IN:	Name of the 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 used, if any

Purpose Writes the RIS8 for the image as the first image in the file, overwriting any information previously in the file.

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

Description The *compress* parameter identifies the method 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-77, the dimensions of the *image* array must be the same as the dimensions of the image itself.

The order in which dimensions are declared is different between C and Fortran-77. Ordering varies because Fortran-77 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-77 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-77 program, the image must be built "on its side".

FORTRAN

```
integer function d8pimg(filename, image, width, height,
                        compress)

character* (*) filename, image
integer width, height, compress
```

DFR8readref/d8rref

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

filename	IN:	Name of the file
ref	IN:	Reference number for next **DFR8getimage**

Purpose Specifies the reference number of the image to be read when **DFR8getimage** is next called.

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

Description **DFR8readref** is usually used in conjunction with **DFANlablist**, 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 index of an image.

FORTRAN

```
integer function d8rref(filename, ref)

character* (*) filename
integer ref
```

DFR8restart/d8first

intn DFR8restart( )

Purpose **DFR8restart** causes the next get command to read from the first raster image set in the file.

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

FORTRAN `integer function d8first( )`

DFR8setcompress/d8scomp

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

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

Purpose Sets the compression type to be used when writing the next 8-bit raster image.

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

Description This routine 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 `comp_info` union contains algorithm-specific information for the library routines that perform the compression and is defined in the `hcomp.h` header file as follows (refer to the header file for inline documentation):

```
typedef union tag_comp_info
  {
      struct
       {
           intn    quality;
           intn    force_baseline;
        }
      jpeg;
      struct
        {
           int32   nt;
           intn    sign_ext;
           intn    fill_one;
           intn    start_bit;
           intn    bit_len;
        }
      nbit;
      struct
        {
           intn    skp_size;
        }
      skphuff;
      struct
        {
           intn    level;
        }
      deflate;
  }
```

`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`. It should normally be set to `1` (forcing baseline results), unless special applications require non-baseline images to be used.

If the compression type is JPEG, **d8scomp** defines the default JPEG compression parameters to be used. If these parameters must be changed later, the **d8sjpeg** routine must be used. (Refer to the Reference Manual page on **d8sjpeg**).

FORTRAN

```
integer function d8scomp(type)

integer type
```

DFR8setpalette/d8spal

intn DFR8setpalette(uint8 **palette*)

palette	IN:	Palette data

Purpose Indicate which palette, if any, is to be used for subsequent image sets.

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

Description The specified palette remains the default palette until changed by a subsequent call to **DFR8setpalette**.

FORTRAN

```
integer function d8spal(palette)

character* (*) palette
```

DFR8writeref/d8wref

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

filename	IN:	Name of the HDF file
ref	IN:	Reference number for next call to **DFR8putimage** or **DFR8addimage**

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

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

Description 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 index of 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 ref
```

DFSDadddata/dsadata

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

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

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

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

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

Calling **DFSDadddata** will write the scientific dataset 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 three-dimensional array of type float32 in a scientific dataset. The scientific dataset 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 dimsizes(*), data(*)
```

DFSDclear/dsclear

intn DFSDclear( )

Purpose Clears all values set by **DFSDset*** routines.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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( )`

DFSDendslab/dseslab

intn DFSDendslab( )

Purpose Terminates a sequence of slab calls started by **DFSDstartslab** by closing the file.

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

FORTRAN `integer function dseslab( )`

DFSDendslice/dseslc

intn DFSDendslice( )

Purpose	Terminates the write operation after storing a slice of data in a scientific dataset.
Return value	Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise.
Description	**DFSDendslice** must be called after all the slices are written. It checks to ensure that the entire dataset 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. HDF will continue to support **DFSDendslice** only to maintain backward compatibility with earlier versions of the library.
FORTRAN	`integer function dseslc( )`

DFSDgetcal/dsgcal

int32 DFSDgetcal(float64 **cal*, float64 **cal_err*, float64 **offset*, float64 **offset_err*, int32 **data_type*)

cal	OUT: Calibration factor
cal_err	OUT: Calibration error
offset	OUT: Uncalibrated offset
offset_err	OUT: Uncalibrated offset error
data_type	OUT: Data type of uncalibrated data

Purpose Retrieves the calibration record, if there is one, attached to a scientific dataset.

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

Description A calibration record contains four 64-bit floating point values followed by a 32-bit integer.

The relationship between a value `iy` stored in a dataset 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 dataset `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 dataset `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,
    data_type)

real cal, cal_err, offset, offset_err
integer data_type
```

DFSDgetdata/dsgdata

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

filename	IN:	Name of the file
rank	IN:	Number of dimensions
dimsizes	IN:	Dimensions of the *data* buffer
data	OUT:	Buffer for the data

Purpose Reads the next dataset in the file.

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

Description If the values of *rank* or *dimsizes* aren't known, **DFSDgetdims** must be called to retrieve them and then use them to determine the buffer space needed for the array data. If the data type of the data in a scientific dataset isn't know, **DFSDgetNT** must be called to retrieve it. Subsequent calls to **DFSDgetdata** (or to **DFSDgetdims** and **DFSDgetdata**)will sequentially read scientific datasets from the file. For example, if **DFSDgetdata** is called three times in succession, the third call reads data from the third scientific dataset in the file.

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

Example The following code reads an array whose dimensions are known to be 100 *x* 200, and whose data type is known to be `int16`.

```
#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 dimsizes(*), data(*)
```

DFSDgetdatalen/dsgdaln

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

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

Purpose Retrieves the lengths of the label, unit, format, and coordinate system strings.

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

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

FORTRAN

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

integer label_len, unit_len, format_len, coords_len
```

DFSDgetdatastrs/dsgdast

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

label	OUT: Label describing 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 the label, unit, and format attribute strings associated with the data.

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

Description 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/dsgdiln

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

dim	IN:	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 Retrieves the length of the label, unit, and format attribute strings associated with the specified dimension.

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

Description The space allocated to hold the label, unit, and format strings must be at least one byte larger than the actual length of the string, to account for the null termination.

FORTRAN

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

integer dim, label_len, unit_len, format_len
```

DFSDgetdims/dsgdims

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

filename	IN:	Name of the HDF file
rank	OUT:	Number of dimensions
dimsizes	OUT:	Buffer for the returned dimensions
maxrank	IN:	Size of the storage buffer *dimsizes*

Purpose Retrieves the number of dimensions (*rank*) of the dataset and the sizes of the dimensions (*dimsizes*) for the next scientific dataset in the file.

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

Description The *maxrank* parameter tells **DFSDgetdims** the size of the array that is allocated for storing the *dimsizes* array. The value of *rank* must not exceed the value of *maxrank*.

The allocation of a buffer for the scientific dataset data should correspond to the values retrieved by **DFSDgetdims**. The first value in the array *dimsizes* should equal the first dimension of the array that is allocated to hold the dataset; the second value in *dimsizes* should equal the second dimension of the dataset, and so forth.

FORTRAN

```
integer function dsgdims(filename, rank, dimsizes, maxrank)

character* (*) filename
integer rank, maxrank
integer dimsizes(*)
```

DFSDgetdimscale/dsgdisc

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

dim	IN:	Dimension this scale corresponds to
size	IN:	Size of the *scale* buffer
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` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

Description The DFSD interface requires the dimension scales to be of the same data type as the corresponding data. To store dimension scales of a different data type than the corresponding data, use the multifile 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, size
integer scale(*)
```

DFSDgetdimstrs/dsgdist

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 Retrieves the label, unit, and format attribute strings corresponding to the specified dimension.

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

Description The space allocated for the label, unit, and format string must be at least one byte larger than the length of the string to accomodate the null termination . If the length is unknown when the program is written, declare the array size as `1+maxlen_label`, `maxlen_unit`, or `maxlen_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/dsgfill

intn DFSDgetfillvalue(VOIDP *fill_value*)

fill_value	OUT: Fill value

Purpose Retrieves the fill value of a DFSD scientific dataset.

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

Description 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)

character* (*) fill_value
```

DFSDgetNT/dsgnt

intn DFSDgetNT(int32 **data_type*)

data_type	OUT: Data type of data in the scientific dataset

Purpose Retrieves the data type of the next dataset to be read.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 *data_type* are of the general form DFNT_. The following are valid symbolic names and their data 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
8-bit character	DFNT_CHAR8	4

FORTRAN

```
integer function dsgnt(num_type)

integer num_type
```

DFSDgetrange/dsgrang

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

max	OUT: Maximum value stored with the scientific dataset
min	OUT: Maximum value stored with the scientific dataset

Purpose Retrieves the maximum and minimum values stored with the scientific dataset.

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

Description The *max* and *min* values are set via a call to **DFSDsetrange**. They are not automatically stored when a dataset is written to a file. The data type of these values is the data type of the dataset array. One implication of this is that in the C version of **DFSDgetrange** the arguments are pointers, rather than simple variables, whereas in the Fortran-77 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 dataset to the actual values in the dataset; 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 dataset. In some cases the *max* and *min* values may actually lie outside the range of values in the dataset.

Example In this example 16-bit data is read from an HDF file.

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

FORTRAN

```
integer function dsgrang(max, min)

character* (*) max, min
```

DFSDgetslice/dsgslc

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
windim	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 dataset from a file.

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

Description **DFSDgetslice** accesses the dataset last accessed by **DFSDgetdims**. If **DFSDgetdims** has not been called for the named file, **DFSDgetslice** gets a slice from the next dataset 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 dataset. For example, if the file contains a three-dimensional dataset, *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 dataset.

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-77-style one-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). HDF will continue to support **DFSDgetslice** only to maintain backward compatibility with HDF applications built on earlier versions of the library.

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, data
integer winst(*), windims(*), dims(*)
```

DFSDlastref/dslref

intn DFSDlastref( )

Purpose Retrieves the most recent reference number used in writing or reading a scientific dataset.

Return value Returns the reference number for the last accessed scientific dataset if successful and `FAIL` (or `-1`) otherwise.

Description **DFSDlastref** returns the value of the last reference number of a scientific dataset read from or written to the file.

FORTRAN `integer function dslref( )`

DFSDndatasets/dsnum

intn DFSDndatasets(char **filename*)

filename	IN:	Name of the HDF file

Purpose Returns the number of scientific datasets in the file.

Return value Returns the number of datasets if successful and `FAIL` (or `-1`) otherwise.

Description In HDF version 3.3, **DFSDndatasets** replaced **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

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

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

Purpose Tests if the scientific dataset with the specified reference number was created by an HDF library earlier than version 3.2.

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

Description If the scientific dataset was created with a version of HDF prior to version 3.2, *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 ref, ispre32
```

DFSDputdata/dspdata

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

filename	IN:	Name of the HDF file
rank	IN:	Number of dimensions of data array to be stored
dimsizes	IN:	Buffer for the dimension sizes
data	IN:	Buffer for the data to be stored

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

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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
<valid numeric data type> data
integer rank
integer dimsizes(*)
```

DFSDputslice/dspslc

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

windims	IN:	Window dimensions specifying the size of the slice to be written
source	IN:	Buffer for the slice
dims	IN:	Dimensions of the *source* array

Purpose Writes part of a scientific dataset to a file.

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

Description **DFSDputslice** read a subset of an array in memory and stores it as part of the scientific dataset 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 dataset 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-77-style one-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). HDF will continue to support **DFSDputslice** only to maintain backward compatibility withearlier versions of the library.

Example Suppose we want to create a 7 *x* 12 scientific dataset array, and we want to write it using slices. Suppose also that the array from which we get our data is a 10 **x** 12 array in memory called source, and we want to write the 7 *x* 12 "window" from top of that array. The following example will do this.

```
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 dataset 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 windims(*), dims(*), source(*)
```

DFSDreadref/dsrref

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

filename	IN:	Name of the HDF file
ref	IN:	Reference number for next **DFSDgetdata** call

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

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

Description This routine is commonly 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 datasets.

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 an index number of a scientific dataset.

FORTRAN

```
integer function dsrref(filename, ref)

character* (*) filename
integer ref
```

DFSDreadslab/dsrslab

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

filename	IN:	Name of the HDF file
start	IN:	Buffer of size *rank* containing the coordinates for the start of the slab
slab_size	IN:	Buffer of size *rank* containing the size of each dimension in the slab
stride	IN:	Subsampling (not yet implemented)
buffer	OUT:	\Buffer for the returned slab
buffer_size	OUT:	Dimensions of the *buffer* buffer

Purpose Reads a slab of data from any scientific dataset.

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

Description **DFSDreadslab** will access to the scientific dataset following the current one if **DFSDgetdims** or **DFSDgetdata** are not called earlier. The *start* array indices are one-based. The rank of *start* must be the same as the number of dimensions of the specified variable. The elements of *slab_size*must be no larger than the dimensions of the scientific dataset 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 dataset, enter `1` in the *slab_size* array for each omitted dimension. For example, to extract a two-dimensional slab from a three-dimensional dataset, specify the beginning coordinates in three 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, buffer
integer start(*), slab_size(*),
integer stride(*), buffer_size(*)
```

DFSDrestart/dsfirst

intn DFSDrestart( )

Purpose Causes the next read command to be read from the first scientific dataset in the file, rather than the scientific dataset following the one that was most recently read.

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

FORTRAN `integer function dsfirst( )`

DFSDsetcal/dsscal

intn DFSDsetcal(float64 *cal*, float64 *cal_err*, float64 *offset*, float64 *offset_err*, int32 *data_type*)

cal	IN:	Calibration factor
cal_err	IN:	Calibration error
offset	IN:	Uncalibrated offset
offset_err	IN:	Uncalibrated offset error
data_type	IN:	Data type of uncalibrated data

Purpose Sets the calibration information associated with data

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

Description This routine sets the calibration record associated with a dataset. 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

data_type   data type of uncalibrated data
```

The relationship between a value `iy` stored in a dataset 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 again for each dataset that is to be written.

Example Suppose the values in a dataset `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 dataset `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,
    data_type)

real*8 cal, cal_err, offset, offset_err
integer data_type
```

DFSDsetdatastrs/dssdast

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

label	IN:	Label describing 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 of the data

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

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

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

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

rank	IN:	Number of dimensions
dimsizes	IN:	Dimensions of the scientific dataset

Purpose Sets the rank and dimension sizes for all subsequent scientific datasets written to the file.

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

Description 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 data type, which is set by calling **DFSDsetNT**.

FORTRAN

```
integer function dssdims(rank, dimsizes)

integer rank
integer dimsizes(*)
```

DFSDsetdimscale/dssdisc

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

dim	IN:	Dimension this scale corresponds to
dimsize	IN:	Size of the *scale* buffer
scale	IN:	Buffer for the scale values

Purpose Defines the scale for a dimension.

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

Description A scale is a one-dimensional array whose values describe reference points along one dimension of the dataset. For example, a two-dimensional dataset 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 dimsize(*), scale(*)
```

DFSDsetdimstrs/dssdist

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

Description In both Fortran-77 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 200 *x* 300 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[0], 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
```

DFSDsetfillvalue/dssfill

intn DFSDsetfillvalue(VOIDP *fill_value*)

fill_value	IN:	Fill value

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

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

Description It is assumed that the fill value has the same data type as the dataset. Once the fill value is set for a particular SDS, it cannot be changed.

If **DFSDsetfillvalue** is called before the first call to **DFSDstartslab**, **DFSDstartslab** will set the fill value tag attribute to the value specified in the **DFSDsetfillvalue** call, but will not actually write out the fill value when **DFSDwriteslab** is called. However, if **DFSDsetfillvalue** is called after the first call the **DFSDstartslab**, the fill value tag attribute will be set by **DFSDsetfillvalue** and the fill value will be written to the slab during the **DFSDwriteslab** call. This is shown in the following C example.

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 each dimension.

```
int16 data[5] = {0, 1, 2, 3, 4};
int dims[1] = {5};
int32 start[1] = {1};
int 32 count[1] = {2};
int16 fillv = -9999;

main( ) {
    int stat;

    /* Set the fill value only; don't write. */
    stat = DFSDsetdims(1, dims);
    stat = DFSDsetNT(DFNT_INT16);
    stat = DFSDwriteslab("File.hdf");
    stat = DFSDsetfillvalue((VOIDP)&fillv);
    stat = DFSDwriteslab(start, start, count, (VOIDP)data);
    stat = DFSDendslab( );

    /* Set the fill value and write it. */
    stat = DFSDsetdims(1, dims);
    stat = DFSDsetNT(DFNT_INT16);
    stat = DFSDsetfillvalue((VOIDP)&fillv);
    stat = DFSDstartslab("File.hdf");
    stat = DFSDwriteslab(start, start, count, (VOIDP)data);
    stat = DFSDendslab( );
}
```

FORTRAN `integer function dssfill(fill_value)`

```
character* (*) fill_value
```

DFSDsetlengths/dsslens

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

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

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

Description The lengths set by this routine 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/dssnt

intn DFSDsetNT(int32 *data_type*)

data_type	IN:	Data type

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

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

Description **DFSDsetNT** must be called if a data 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**.

The following symbolic names can be used as the value of *data_type*:

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
8-bit character	DFNT_CHAR8	4

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 dataset.

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

FORTRAN

```
integer function dssnt(num_type)
integer num_type
```

DFSDsetrange/dssrang

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

max	IN:	Highest value in the range
min	IN:	Lowest value in the range

Purpose Stores the specified maximum and minimum data values.

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

Description It is assumed that the data 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-77 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 datasets 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 dataset array data.

```
int16 max, min, data[100][100];

 ...

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

FORTRAN

```
integer function dssrang(max, min)
character* (*) max, min
```

DFSDstartslab/dssslab

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 dataset.

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

Description **DFSDsetdims** must be called before calling **DFSDstartslab**. No call which involves 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

intn DFSDstartslice(char **filename*)

filename	IN:	Name of the HDF file

Purpose Prepares the interface to write a data slice to the specified file.

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

Description Before calling **DFSDstartslice**, **DFSDsetdims** must be called to specify the dimensions of the dataset to be written to the file. **DFSDstartslice** always appends a new dataset to an existing file.

Also, **DFSDstartslice** must be called before **DFSDputslice** or **DFSDendslice**.

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

FORTRAN

```
integer function dssslc(filename)

character* (*) filename
```

DFSDwriteref/dswref

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

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

Purpose **DFSDwriteref** determines the reference number of the dataset to overwritten next by **DFSDputdata** or **DFSDadddata**, after checking for its existance.

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

Description If a non-existent reference number is specified, an error code will be returned.

As this routine alters data in a destructive manner, **DFSDwriteref** should be used with caution.

FORTRAN

```
integer function dswref(filename, ref)

character* (*) filename
integer ref
```

DFSDwriteslab/dswslab

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

start	IN:	Array containing the starting coordinates of the slab
stride	IN:	Array containing the dimensions for subsampling
count	IN:	Array containing the size of the slab
data	IN:	Array to hold the floating point data to be written

Purpose Writes a slab of data to a scientific dataset.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 dataset's dimensions in order. The stride feature is not currently implemented. For now just pass the *start* array as the argument for the *stride*paramater, 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 dataset'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 dataset. 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 start(*), stride(*), count(*)
character* (*) data
```

DFUfptoimage/duf2im

int DFUfptoimage(int32 *hdim*, int32 *vdim*, float32 *max*, float32 *min*, float32 **hscale*, float32 **vscale*, float32 **data*, uint8 **palette*, char **outfile*, int *ct_method*, int32 *hres*, int32 *vres*, int *compress*)

hdim	IN:	Horizontal dimension of the input data
vdim	IN:	Vertical dimension of the input data
max	IN:	Maximum value of the input data
min	IN:	Minimum value of the input data
hscale	IN:	Horizontal scale of the input data (optional)
vscale	IN:	Vertical scale of the input data (optional)
data	IN:	Buffer containing the input data
palette	IN:	Pointer to the palette data
outfile	IN:	Name of the file the image data will be stored in
ct_method	IN:	Color tranformation method
hres	IN:	Horizontal resolution to be applied to the output image
vres	IN:	Vertical resolution to be applied to the output image
compress	IN:	Compression flag

Purpose Converts floating point data to 8-bit raster image format and stores the converted image data in the specified file.

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

Description This routine is very similar to the utility `fptohdf`, which takes its input from one or more files, rather than from internal memory. Another difference is that this routine allows compression (run-length encoding), whereas `fptohdf` does not at present.

As this routine is meant to mimic many of the features of NCSA DataScope, much of the code has been taken directly from the DataScope source.

Valid values for *ct_method* are: `1` (or `EXPAND)` for expansion and `2` (or `INTERP)` for interpolation.

Valid values for *compress* are: `0` for no compression and `1` for compression enabled.

FORTRAN

```
integer function duf2im(hdim, vdim, max, min, hscale,
        vscale, data, palette, outfile, ct_method, hres,
        vres, compress)

integer hdim, vdim
real  max, min, hscale, vscale, data
character* (*) palette, outfile
integer ctmethod, hres, vres, compress
```

GRattrinfo/mgatinf

int32 GRattrinfo(int32 *[ri, gr]_id*, int32 *attr_index*, char **name*, int32 **data_type*, int32 **length*)

[ri, gr]_id	IN:	Raster image or GR dataset identifier of the target object
attr_index	IN:	Index of the attribute
name	OUT:	Buffer for the name of the attribute
data_type	OUT:	Data type of the attribute
length	OUT:	Length of the attribute

Purpose Reads the specified attribute's data type and length.

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

Example This example illustrates the use of **GRattrinfo** in reading the attributes of a general raster image:

```
#include "hdf.h"

int32 ri_id, gr_id, stat, attr_index;
int32 *data_type, length;
int32 attr_values[2];
char name[MAX_GR_NAME];
  ...
ri_id = GRselect(gr_id, 0);
stat = GRattrinfo(gr_id, attr_index, name, data_type,
                  length);
stat = GRgetattr(ri_id, attr_index, attr_values)
  ...
```

FORTRAN

```
integer function mgatinf([ri, gr]_id,  attr_index,
        name, data_type, length)

integer [ri, gr]_id, data_type, num_attrs, length
character* (*) name
```

GRcreate/mgcreat

int32 GRcreate(int32 *gr_id*, const char **name*, int32 *ncomp*, int32 *data_type*, int32 *interlace_mode*, int32 *dim_sizes*[2])

gr_id	IN:	General raster interface identifier returned by **GRstart**
name	IN:	Name of the image to be created
ncomp	IN:	Number of components in each element of the image
data_type	IN:	Type of the image data
interlace_mode	IN:	Interlace mode of the image data
dim_sizes	IN:	Size of each dimension of the image

Purpose Creates a raster image via the general raster image interface.

Return value Returns a general raster imge identifier if successful and `FAIL` (or `-1`) otherwise.

Example This example illustrates the use of **GRcreate**:

```
#include "hdf.h"

int32 gr_id, ri_id, file_id, stat;
char *name = "Image name";
int32 ncomp = 2;
int32 interlace_mode = MFGR_INTERLACE_PIXEL;
int32 data_type = DFNT_UINT16;

file_id = Hopen("myfile", DFACC_WRITE, 0);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, data_type,
        interlace_mode, dim_sizes);
   ...
stat = GRendaccess(ri_id);
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgcreat(gr_id, name, ncomp, data_type
    interlace_mode, dim_sizes)

integer gr_id, data_type, interlace_mode, dim_sizes(2)
character* (*) name
```

GRend/mgend

intn GRend(int32 *gr_id*)

gr_id	IN:	General raster interface identifier returned by **GRstart**.

Purpose Terminates the general raster interface session.

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

Description This routine is used with the **GRstart** routine to define the extent of a general raster interface session. As with the start routines in the other interfaces, **GRend** disposes of the internal structures used in the remaining GR routines. Use the general purpose routines **Hopen** and **Hclose** to manage file access because the GR routines will not open and close HDF files.

Example This example illustrates the use of **GRstart** and **GRend** in initializing and terminating a GR interface session.

```
#include "hdf.h"
int32 gr_id, file_id, stat;

file_id = Hopen("myfile", DFACC_WRITE, 0);
gr_id = GRstart(file_id);
  ...
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgend(gr_id)

integer gr_id
```

GRendaccess/mgendac

intn GRendaccess(int32 *ri_id*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**

Purpose Terminates access to an general raster image.

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

Description There should be one call to **GRendaccess** for every *ri_id* returned from **GRselect** and/or **GRcreate**. Attempts to access *ri_id* after the call to **GRendaccess** will result in an error condition.

Example This example illustrates the use of **GRendaccess**:

```
#include "hdf.h"

int32 gr_id, ri_id, file_id, stat;
char *name = "Image name";
int32 ncomp = 2;
int32 interlace_mode = MFGR_INTERLACE_PIXEL;
int32 data_type = DFNT_UINT16;
int32 dim_sizes[2];

file_id = Hopen("myfile", DFACC_WRITE, 0);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, data_type,
        interlace_mode, dim_sizes);

   ...
stat = GRendaccess(ri_id);
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgendac(ri_id)

integer ri_id
```

GRfileinfo/mgfinfo

intn GRfileinfo(int32 *gr_id*, int32 **n_datasets*, int32 **n_file_attrs*)

gr_id	IN:	General raster interface identifier returned by **GRstart**
n_datasets	OUT:	Number of datasets in the file
n_file_attrs	OUT:	Number of global attributes in the file

Purpose Reports general information about the number of datasets and global attributes for the GR interface.

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

Description This routine is generally used to find the range of acceptable indices for **GRselect** calls.

Example This example illustrates the use of **GRfileinfo** to search through the general raster image datasets in a file:

```
#include "hdf.h"

int32 gr_id, n_datasets, n_file_attrs, gr_index;

   ...
 stat = GRgetfileinfo(gr_id, &n_datasets, &n_file_attrs);
 for(gr_index = 0; gr_index < n_datasets; gr_index++) {
     ri_id = GRselect(gr_id, gr_index);
   ...
 }
```

FORTRAN

```
integer function mgfinfo(gr_id, n_datasets, n_file_attrs)

integer gr_id, n_datasets, n_file_attrs
```

GRfindattr/mgfndat

int32 GRfindattr(int32 *[ri, gr]_id*, const char **attr_name*)

[ri, gr]_id	IN:	Identifier of the general raster image or the GR interface identifier returned by **GRstart**
attr_name	IN:	Name of the attribute to be searched for

Purpose Returns the index of the attribute with the specified name.

Return value Returns the index of the attribute if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function mgfndat([ri, gr]_id,  attr_name)

integer [ri, gr]_id
character* (*) attr_name
```

GRgetattr/mggnatt/mggcatt

intn GRgetattr(int32 *[ri, gr]_id*, int32 *attr_index*, VOIDP *values*)

[ri, gr]_id	IN:	Identifier of the general raster image or the GR interface identifier returned by **GRstart**
attr_index	IN:	Index of the attribute(s)
values	OUT:	Buffer for the attribute values

Purpose Reads an attribute of a raster image, or all raster images, into a buffer.

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

Description **GRgetattr** is often used in conjunction with **GRselect** and **GRnametoindex**, or **GRselect** and **GRreftoindex**, to retrieve the attributes of a specific general raster image. If a *gr_id* is specified as the first parameter, the global attributes (applied to all general raster images in the file) are retrieved.

It is not possible to read a subset of the attribute values assigned to the target object with **GRgetattr** - all values will be read.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mggnatt**) and the other for buffered character data (**mggcatt**).

Example This example illustrates the use of **GRgetattr** to read the attributes of a general raster image:

```
#include "hdf.h"

int32 ri_id, gr_id, stat, attr_index;
int32 attr_values[2];
   ...
ri_id = GRselect(gr_id, 0);
attr_index = GRfindattr(ri_id, "Target attribute");
stat = GRgetattr(ri_id, attr_index, attr_values);
   ...
```

FORTRAN

```
integer function mggnatt([ri, gr]_id,  attr_index,
                         values)

integer [ri, gr]_id, attr_index
<valid numeric data type> values(*)

integer function mggcatt([ri, gr]_id,  attr_index,
                         values)

integer [ri, gr]_id, attr_index
character* (*) values
```

GRgetiminfo/mggiinf

intn GRgetiminfo(int32 *ri_id*, char **gr_name*, int32 **ncomp*, int32 **data_type*, int32 **interlace*, int32 *dim_sizes*[2], int32 **num_attrs*)

ri_id	IN:	General raster interface identifier returned by **GRcreate** or **GRselect**
gr_name	OUT:	Buffer for the returned name of the general raster image
ncomp	OUT:	Buffer for the returned number of components in the image
data_type	OUT:	Buffer for the returned data type of the image data
interlace	OUT:	Buffer for the returned interlace mode of the stored image data
dim_sizes	OUT:	Buffer for the returned sizes of each image dimension
num_attrs	OUT:	Buffer for the returned number of attributes assigned to the image

Purpose Reports general information about the specified raster image.

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

Description Each of the arguments can be set to `NULL`, in which case the corresponding item of information will not be retrieved.

FORTRAN

```
integer function mggiinf(ri_id, gr_name, ncomp, data_type,
        interlace, dim_sizes, num_attrs)

integer ri_id, ncomp, data_type, interlace, num_attrs
integer dim_sizes[2]
character* (*) gr_name
```

GRgetlutid/mggltid

int32 GRgetlutid(int32 *ri_id*, int32 *palette_index*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
palette_index	IN:	Index of the palette

Purpose Assigns a palette identifier to a general raster image dataset.

Return value Returns a palette identifier if successful and `FAIL` (or `-1`) otherwise.

Description This routine establishes the connection between a palette and a general raster image. Often used in conjunction with **GRwritelut**. Currently, only one palette can be assigned to a general raster image, which means that *palette_index* should always be set to `0`.

Example This example illustrates the use of **GRgetlutid** in creating a palette:

```
#include "hdf.h"

int32 pal_id, ri_id, gr_id, stat, image_index;
char pal_data[PALETTE_SIZE][3];
int32 ncomp = 3;
int32 data_type = DFNT_INT8;
int32 interlace = MFGR_INTERLACE_PIXEL;

   ...
image_index = GRnametoindex(gr_id, "Target image");
ri_id = GRselect(gr_id, image_index);
pal_id = GRgetlutid(ri_id, 0);
stat = GRwritelut(pal_id, ncomp, data_type, interlace,
        PALETTE_SIZE, pal_data);
   ...
```

FORTRAN

```
integer function mggltid(ri_id, palette_index)

integer ri_id, palette_index
```

GRgetlutinfo/mgglinf

intn GRgetlutinfo(int32 *pal_id*, int32 **ncomp*, int32 **data_type*, int32 **interlace*, int32 **num_entries*)

pal_id	IN:	Palette identifier returned by **GRgetlutid**
ncomp	OUT:	Number of components in the palette
data_type	OUT:	Data type of the palette data
interlace	OUT:	Interlace mode of the stored palette data
num_entries	OUT:	Number of color lookup table entries in the palette

Purpose Reports the data type, interlace mode and number of color lookup table entries of the specified palette.

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

Description Each of the arguments can be set to `NULL`, in which case the corresponding item of information will not be retrieved.

FORTRAN

```
integer function mgglinf(pal_id, ncomp, data_type, interlace,
                         num_entries)

integer pal_id, ncomp, data_type, interlace, num_entries
```

GRidtoref/mgid2ref

uint16 GRidtoref(int32 *ri_id*)

ri_id	IN:	General raster image identifier returned by **GRselect** or **GRcreate**

Purpose Maps a general raster image identifier to a reference number.

Return value Returns a general raster image identifier if successful and `FAIL` (or `-1`) otherwise.

Description This routine is commonly used for the purpose of annotating the image or including the image within a vgroup.

Example This example illustrates the use of **GRidtoref** in attaching an annotation to an image:

```
#include "hdf.h"

uint16 ref;
int32 an_id, ann_id, file_id;
ref = GRidtoref(ri_id);
an_id = ANstart(file_id);
ann_id = ANcreate(an_id, DFTAG_RI, ref, AN_DATA_LABEL);
```

FORTRAN

```
integer function mgid2ref(ri_id)

integer ri_id
```

GRluttoref

uint16 GRluttoref(int32 *pal_id*)

pal_id	IN:	Palette identifier returned from **GRgetlutid**

Purpose Returns the reference number of the specified palette.

Return value Returns the reference number of the palette if successful or `DFTAG_WILDCARD` (or `0`) otherwise.

Description This routine is commonly used for the purpose of annotating the palette or including the palette within a vgroup.

GRnametoindex/mgn2idx

int32 GRnametoindex(int32 *gr_id*, const char *gr_*name*)

gr_id	IN:	General raster interface identifier returned by **GRstart**
gr_name	IN:	Name of the target general raster image

Purpose Maps the name of a general raster image to an index.

Return value Returns the index of the image if successful and `FAIL` (or `-1`) otherwise.

Description In a manner similar to **GRreftoindex**, this routine is commonly used for the purpose of accessing an image in conjunction with a call to **GRselect**.

Example This example illustrates the use of **GRnametoindex**:

```
#include "hdf.h"

int32 gr_id, ri_id, index;
char *gr_name;
   ...
 index = GRnametoindex(gr_id, gr_name);
 ri_id = GRselect(gr_id, index);
   ...
```

FORTRAN

```
integer function mgn2idx(gr_id, gr_name)

integer gr_id
character* (*) gr_name
```

GRreadimage/mgrdimg/mgrcimg

intn GRreadimage(int32 *ri_id*, int32 *start*[2], int32 *stride*[2], int32 *edge*[2], VOIDP *data*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
start	IN:	Array containing the two-dimensional coordinate of the initial location for the read
stride	IN:	Array containing the number of data locations the current location is to be moved forward before each read
edge	IN:	Array containing the number of data elements that will be read along each dimension
data	OUT:	Buffer for the image data to be read

Purpose Reads a general raster image to the current HDF file.

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

Description By setting the *start*, *stride* and *edge* parameters appropriately , **GRreadimage** will perform subsampling and image slab writes. Setting *stride* to `NULL` assumes a *stride* value of `1`.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mgrdimg**) and the other for buffered character data (**mgrcimg**).

Example This example illustrates the use of **GRreadimage**:

```
#include "hdf.h"

int32 gr_id, ri_id, stat;
char *name = "Image name";
int32 ncomp = 0;
int32 interlace_mode = MFGR_INTERLACE_PIXEL;
int32 data_type = DFNT_UINT16;
int32 start[2], stride[2], edge[2];
VOIDP data[100];
int32 dim_sizes[2] = {50, 60};

file_id = Hopen("myfile", DFACC_READ, 0);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, data_type,
        interlace_mode, dim_sizes);
  ...
start[0] = start[1] = 0;
stride[0] = stride[1] = 1;
edge[0] = edge[1] = 10;

stat = GRreadimage(ri_id, start, stride, edge, data);
```

```
   ...
stat = GRendaccess(ri_id);
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgrdimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
<valid numeric data type> data(*)

integer function mgrcimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
character* (*) data
```

GRreadlut/mgrdlut/mgrclut

intn GRreadlut(int32 *pal_id*, VOIDP *pal_data*)

pal_id	IN:	Palette identifier of the target general raster image
pal_data	OUT:	Buffer for the palette data to be read

Purpose Reads the palette referred to by the given palette identifier.

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

Description This routine is commonly used in conjunction with a call to **GRgetlutid**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mgrdlut**) and the other for buffered character data (**mgrclut**).

Example This example illustrates the use of **GRreadlut**:

```
#include "hdf.h"

int32 pal_id, ri_id, gr_id, stat;
intn index;
char pal_data[PALETTE_SIZE];
   ...
index = GRnametoindex(gr_id, "Target image");
ri_id = GRselect(gr_id, index);
pal_id = GRgetlutid(ri_id, 0);
stat = GRreadlut(pal_id, pal_data);
   ...
```

FORTRAN

```
integer function mgrclut(pal_id, pal_data)

integer pal_id
<valid numeric data type> pal_data(*)

integer function mgrdlut(pal_id, pal_data)

integer pal_id
character* (*) pal_data
```

GRreftoindex/mgr2idx

int32 GRreftoindex(int32 *gr_id*, uint16 *ref*)

gr_id	IN:	General raster interface identifier returned by **GRstart**
ref	IN:	Reference number to be mapped

Purpose Maps the reference number of an image to an index.

Return value Returns the index of the image if successful and `FAIL` (or `-1`) otherwise.

Description In the same manner as **GRnametoindex**, this routine is commonly used for the purpose of accessing an image in conjunction with a call to **GRselect**.

Example This example illustrates the use of **GRreftoindex**:

```
#include "hdf.h"

int32 gr_id, ri_id, index;
uint16 ref;

   ...
index = GRreftoindex(gr_id, ref);
ri_id = GRselect(gr_id, index);
   ...
```

FORTRAN

```
integer function mgr2idx(gr_id, ref)

integer gr_id, ref
```

GRreqimageil/mgrimil

intn GRreqimageil(int32 *ri_id*, intn *interlace_mode*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
interlace_mode	IN:	Interlace mode to be in effect during the next image operation: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`), or `MFGR_INTERLACE_COMPONENT` (or `2`)

Purpose Sets or resets the interlace mode to be in effect during the next image read operation.

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

Description **GRreqimageil** can to called anytime before the image read operation.

Example This example illustrates the use of **GRreqimageil**:

```
#include "hdf.h"

int32 ri_id, stat;
intn interlace_mode = MFGR_INTERLACE_PIXEL;
int32 start[2], stride[2], edge[2];
uint16 data[100];

start[0] = start[1] = 0;
stride[0] = stride[1] = 1;
edge[0] = edge[1] = 10;
  ...
stat = GRreqimageil(ri_id, interlace_mode);
stat = GRreadimage(ri_id, start, stride, edge, data);
  ...
```

FORTRAN

```
integer function mgrimil(ri_id, interlace_mode)

integer ri_id, interlace_mode
```

GRreqlutil/mgrltil

intn GRreqlutil(int32 *ri_id*, intn *interlace_mode*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
interlace_mode	IN:	Interlace mode of the next palette read operation: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`), or `MFGR_INTERLACE_COMPONENT` (or `2`)

Purpose Sets or resets the interlace mode that will be in effect during the next palette read operation.

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

Description **GRreqlutil** can to called anytime before the palette read operation.

Example This example illustrates the use of **GRreqlutil**:

```
#include "hdf.h"

int32 ri_id, pal_id, stat;
VOIDP palette_data;
intn interlace_mode = MFGR_INTERLACE_PIXEL;
  ...
stat = GRreqlutil(ri_id, interlace_mode);
stat = GRreadlut(pal_id, palette_data);
  ...
```

FORTRAN

```
integer function mgrltil(ri_id, interlace_mode)

integer ri_id, interlace_mode
```

GRselect/mgselct

int32 GRselect(int32 *gr_id*, int32 *index*)

gr_id	IN:	General raster interface identifier returned by **GRstart**
index	IN:	Index of the general raster image in the file

Purpose Selects and returns the identifier for the general raster image identified by the index *index*.

Return value Returns the identifier of the selected general raster image if successful or `FAIL` (or `-1`) otherwise.

Description The index supplied by the parameter *index* is zero-based.

Example This example illustrates the use of **GRselect**:

```
#include "hdf.h"

int32 gr_id, ri_id, file_id, index, stat;

file_id = Hopen("myfile", DFACC_READ, 0);
gr_id = GRstart(file_id);
ri_id = GRselect(gr_id, index);
  ...
stat = GRendaccess(ri_id);
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgselct(gr_id, index)

integer gr_id, index
```

GRsetaccesstype/mgsactp

intn GRsetaccesstype(int32 *ri_id*, uintn *access_mode*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
access_mode	IN:	Access mode for the image data: `DFACC_SERIAL` or `DFACC_PARALLEL`

Purpose Specifies the access mode to be either parallel or serial I/O.

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

Description **GRsetaccesstype** is designed specifically for use on computers that support parallel I/O. On all other computers, the standard serial mode I/O setting is recommended.

FORTRAN

```
integer function mgsactp(ri_id, access_mode)

integer ri_id, access_mode
```

GRsetattr/mgsnatt/mgscatt

intn GRsetattr(int32 *[ri, gr]_id*, const char **attr_name*, int32 *data_type*, int32 *count*, const VOIDP *values*)

[ri, gr]_id	IN:	Identifier of the raster image or the GR interface identifier returned by **GRstart**
attr_name	IN:	Name of the attribute(s)
data_type	IN:	Data type of the attribute(s)
count	IN:	Number of values in the attribute(s)
values	IN:	Buffer for attribute values

Purpose Assigns an attribute to one raster image, or all general raster images, in a file.

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

Description Currently, the only predefined attribute is the fill value, identified by the `FILL_ATTR` definition. If a *gr_id* is specified as the first parameter, the global attributes (applied to all general raster images in the file) are set.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mgsnatt**) and the other for buffered character data (**mgscatt**).

Example This example illustrates the use of **GRsetattr** to assign atrributes to a general raster image:

```
#include "hdf.h"

int32 ri_id, gr_id, stat;
int32 attr_values[2];
   ...
attr_values[0] = 5;
attr_values[1] = 50;
ri_id = GRselect(gr_id, 0);
stat = GRsetattr(ri_id, "Value range", DFNT_INT32, 2
                 attr_values);
   ...
```

FORTRAN

```
integer function mgsnatt([ri, gr]_id,
        attr_name, data_type, count, values)

integer [ri, gr]_id, attr_name, data_type
integer count
<valid numeric data type> values(*)

integer function mgscatt([ri, gr]_id,
        attr_name, data_type, count, values)

integer [ri, gr]_id, attr_name, data_type
```

```
integer count
character* (*) values
```

GRsetcompress

intn GRsetcompress(int32 *ri_id*, int32 *comp_type*, comp_info **c_info*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
comp_type	IN:	Compression method for the image data: `COMP_CODE_RLE`, `COMP_CODE_DEFLATE` or `COMP_CODE_SKPHUFF`
c_info	IN:	Pointer to the `comp_info` union

Purpose Specifies that the image data of a general raster dataset is a compressed special element.

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

Description The `COMP_CODE_RLE` definition specifies run-length encoding, `COMP_CODE_DEFLATE` specifies Gnu ZIP (or GZIP) compression and `COMP_CODE_SKPHUFF` specifies skipping Huffman. The `comp_info` union contains algorithm-specific information for the library routines that perform the compression and is defined in the `hcomp.h` header file.

GRsetexternalfile/mgsxfil

int32 GRsetexternalfile(int32 *ri_id*, const char **filename*, int32 *offset*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
filename	IN:	Name of the file the external dataset will be stored in
offset	IN:	Offset, in bytes, from the beginning of the external file to the image data

Purpose Specifies that the image data of a general raster image dataset is a special element of an external element. Creates an external image array for a general raster dataset.

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

Description An external image array is one that is stored in a file that is not the file containing the metadata for the image. **GRsetexternalfile** marks the image identified by *ri_id* as one whose data is to be written to the external image array. It can only be called once for each general raster dataset created.

FORTRAN

```
integer function mgsxfil(ri_id, filename, offset)

integer ri_id, offset
character* (*) filename
```

GRstart/mgstart

intn GRstart(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**.

Purpose Initializes the general raster interface.

Return value Returns a general raster interface identifier if successful and `FAIL` (or `-1`) otherwise.

Description This routine is used with the **GRend** routine to define the extent of a general raster interface session. As with the start routines in the other interfaces, **GRstart** initializes the internal interface structures needed for the remaining GR routines. Use the general purpose routines **Hopen** and **Hclose** to manage file access. The GR routines will not open and close HDF files.

Example This example illustrates the use of **GRstart** and **GRend** in initializing and terminating a GR interface session.

```
#include "hdf.h"
int32 gr_id, file_id, stat;

file_id = Hopen("myfile", DFACC_WRITE, 0);
gr_id = GRstart(file_id);
  ...
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgstart(file_id)

integer file_id
```

GRwriteimage/mgwrimg/mgwcimg

intn GRwriteimage(int32 *ri_id*, int32 *start*[2], int32 *stride*[2], int32 *edge*[2], VOIDP *data*)

ri_id	IN:	General raster image identifier returned by **GRcreate** or **GRselect**
start	IN:	Array containing the two-dimensional coordinate of the initial location for the write
stride	IN:	Array containing the number of data locations the current location is to be moved forward before each write
edge	IN:	Array containing the number of data elements that will be written along each dimension
data	IN:	Buffer containing the image data to be written

Purpose Writes a general raster image to the current HDF file.

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

Description By setting the *start*, *stride* and *edge* parameters appropriately, **GRwriteimage** will perform subsampling and image slab writes. Setting *stride* to `NULL` assumes a stride value of `1`.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mgwrimg**) and the other for buffered character data (**mgwcimg**).

Example This example illustrates the use of **GRwriteimage**:

```
#include "hdf.h"

int32 file_id, gr_id, ri_id, stat;
char *name = "Image name";
int32 ncomp = 0;
int32 interlace_mode = MFGR_INTERLACE_PIXEL;
int32 data_type = DFNT_UINT16;
int32 dim_sizes[2];
uint16 *data;

file_id = Hopen("myfile", DFACC_WRITE, 0);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, data_type,
        interlace_mode, dim_sizes);
   ...
stat = GRwriteimage(ri_id, start, stride, edge, data);
  ...
stat = GRendaccess(ri_id);
stat = GRend(gr_id);
Hclose(file_id);
```

FORTRAN

```
integer function mgwrimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
<valid numeric data type> data(*)

integer function mgwcimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
character* (*) data
```

GRwritelut/mgwrlut/mgwclut

intn GRwritetlut(int32 *pal_id*, int32 *ncomp*, int32 *data_type*, int32 *interlace*, int32 *num_entries*, VOIDP *pal_data*)

pal_id	IN:	Palette identifier to be assigned to the written data
ncomp	IN:	Number of color components in the palette
data_type	IN:	Data type of the palette data
interlace	IN:	Interlace mode of the stored palette data
num_entries	IN:	Number of entries in the palette
pal_data	IN:	Buffer for the palette data to be written

Purpose Writes palette data to a general raster image dataset.

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

Description This routine is commonly used in conjunction with a call to **GRgetlutid**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**mgwrlut**) and the other for buffered character data (**mgwclut**).

Example This example illustrates the use of **GRwritelut**:

```
#include "hdf.h"

int32 pal_id, ri_id, gr_id, stat;
intn lut_index;
char pal_data[PALETTE_SIZE][3];
int32 ncomp = 3;
int32 data_type = DFNT_INT8;
int32 interlace = MFGR_INTERLACE_PIXEL;
   ...
index = GRnametoindex(gr_id, "Target image");
ri_id = GRselect(gr_id, index);
pal_id = GRgetlutid(ri_id, lut_index);
stat = GRwritelut(pal_id, ncomp, data_type, interlace,
        PALETTE_SIZE, pal_data);
   ...
```

FORTRAN

```
integer function mgwrlut(pal_id, ncomp, data_type,
    interlace, num_entries, pal_data)

integer pal_id, ncomp, data_type, interlace, num_entries
<valid numeric data type> pal_data(*)

integer function mgwclut(pal_id, ncomp, data_type,
    interlace, num_entries, pal_data)
```

```
integer pal_id, ncomp, data_type, interlace, num_entries
character* (*) pal_data
```

Happendable

intn Happendable(int32 *h_id*)

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

Purpose Specifies that the specified element can be appended to

Return value Returns `SUCCEED` (or `0`) if data element can be appended and `FAIL` (or `-1`) otherwise.

Description If a data element is at the end of a file **Happendable** allows **Hwrite** to append data to it, converting it to linked-block element only when necessary.

Hcache

intn Hcache(int32 *file_id*, intn *cache_switch*)

file_id	IN:	File identifier returned by **Hopen**
cache_switch	IN:	Flag to enable or disable caching

Purpose Enables low-level caching for the specified file.

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

Description If *file_id* is set to `CACHE_ALL_FILES`, then the value of *cache_switch* is used to modify the default file cache setting.

Valid values for *cache_switch* are: `TRUE` (or `1`) to enable caching and `FALSE` (or `0`) to disable caching.

Hclose/hclose

intn Hclose(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**

Purpose Closes the access path to the file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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` (or `-1`) is returned, and the file remains open. This is a common error when developing new interfaces. Refer to the Reference Manual page on **Hendaccess** for a discussion of this problem.

FORTRAN

```
integer function hclose(file_id)

integer file_id
```

Hdeldd

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

file_id	IN:	File identifier returned by **Hopen**
tag	IN:	Tag of data descriptor to be deleted
ref	IN:	Reference number of data descriptor to be deleted

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

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

Description Once the data descriptor is removed, the data in the data object becomes inaccessible and is marked as such. 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.

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

Description The number of active access identifiers is limited to `MAX_ACC` as defined in the `hlimits.h` 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 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 the number of active access identifiers. Refer to the Reference Manual page on **HEprint**.

This is a 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 method of determining which access identifiers have yet to be released.

Hendbitaccess

intn Hendbitaccess(int32 *h_id*, intn *flushbit*)

h_id	IN:	Identifier of the bit-access element to be disposed of
flushbit	IN:	Specifies how the leftover bits are to be flushed

Purpose Disposes of the specified bit-access file element.

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

Description If called after a bit-write operation, **Hendbitaccess** flushes all buffered bits to the dataset, then calls **Hendaccess**..

"Leftover bits" are bits that have been buffered, but are fewer than the number of bits defined by `BITNUM`, which is usually set to `8`.

Valid codes for *flushbit* are:

```
0 - flush with zeros

1 - flush with ones

-1 - dispose of leftover bits
```

Hexist

intn Hexist(int32 *h_id*, uint16 *search_tag*, uint16 *search_ref*)

h_id	IN:	Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
search_tag	IN:	Tag of the object to be searched for
search_ref	IN:	Reference number of the object to be searched for

Purpose Locates an object in an HDF file.

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

Description Simple interface to **Hfind** that determines if a given tag/reference number pair exists in a file. Wildcards apply..

Hfind performs all validity checking; this is just a *very* simple wrapper around it.

Hfidinquire

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

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

Purpose Returns file information through a reference of its file identifier.

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

Description Gets the complete path name, access mode, and number of access identifiers associated with a file. 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.

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:	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/OUT:	If (**find_tag* == 0) and (**find_ref* == 0) then start the search from either the beginning or the end of the file. If the object is found, the tags of the object will be returned here.
find_ref	IN/OUT:	If (**find_tag* == 0) and (**find_ref* == 0) then start the search from either the beginning or the end of the file. If the object is found, the reference numbers of the object will be returned here.
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 to be searched for in an HDF file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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`, searching is forward from the current position in the file, otherwise `DF_BACKWARD` specifies backward searches 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`.

Hflushdd

intn Hflushdd(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**

Purpose Makes the disk image of the specified HDF file conform to its memory representation.

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

Hgetbit

intn Hgetbit(int32 *h_id*)

h_id	IN:	Bit-access element identifier

Purpose Reads one bit from the specified bit-access element.

Return value Returns the bit read (or `0` or `1`) if successful and `FAIL` (or `-1`) otherwise.

Description This function is a wrapper for **Hbitread**.

Hgetelement

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

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

Purpose Reads the data element for the specified tag and reference number and writes it to the *data* buffer.

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

Description It is assumed that the space allocated for the buffer is large enough to hold the data.

Hgetfileversion

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

file_id	IN:	File identifier returned by **Hopen**
major_v	OUT:	Major version number
minor_v	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Version number text string

Purpose Retrieves version information for an HDF file.

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

Description 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* argument is limited to a length of `LIBVSTR_LEN` characters as defined in `hfile.h`.

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: Version number text string

Purpose Retrieves the version information of the current HDF library.

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

Description The version information is 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`.

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:	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` (or `0`) if the access identifier points to a valid data element and `FAIL` (or `-1`) otherwise.

Description 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` (or `-1`). If *h_id* is invalid, the function returns `FAIL` (or `-1`) and the access type is set to zero. To avoid excess information, pass `NULL` for any unnecessary pointer.

Hishdf

intn Hishdf(const char **filename*)

filename	IN:	Complete path and filename of the file to be checked

Purpose Determines if a file is an HDF file.

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

Description The first four bytes of a file identify it as 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.

Hlength

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

file_id	IN:	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` (or `-1`) otherwise.

Description **Hlength** calls **Hstartread**, **HQuerylength**, and **Hendaccess** to determine the length of a data element. **Hlength** uses **Hstartread** to obtain an access identifier for the specified data object.

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.

Hnewref

uint16 Hnewref(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**

Purpose Returns a reference number that can be used with any tag to produce a unique tag /reference number pair.

Return value Returns the reference number if successful and `0` otherwise.

Description Successive calls to **Hnewref** will generate reference number values that increase by one each time until the highest possible reference number has been returned. At this point, additional calls to **Hnewref** will return an increasing sequence of unused reference number values starting from `1`.

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 Searches for the next data descriptor that matches the specified tag and reference number.

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

Description 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 backwards from the end of a data descriptor list is not yet implemented.

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

Hnumber/hnumber

int32 Hnumber(int32 file_id, uint16 *tag*)

file_id	IN:	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` (or `-1`) otherwise.

Description **Hnumber** determines how many objects with the specified tag are in a file. To determine 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 file_id, tag
```

Hoffset

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

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

Purpose Returns the offset of a data element in the file.

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

Description **Hoffset** calls **Hstartread**, **HQueryoffset**, and **Hendaccess** to determine the length of a data element. **Hoffset** uses **Hstartread** to obtain an access identifier for the specified data object.

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 should not be used to determine the offset of an external element. In this case, **Hoffset** returns zero, an invalid offset for HDF files.

Hopen/hopen

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

filename	IN:	Complete path and filename for the file to be opened
access	IN:	Access code definition (preceded by `DFACC_`)
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` (or `-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.

If *n_dds* is set to `0`, the number of data descriptors will be defined as the machine default.

HDF provides several access code definitions:

`DFACC_READ` - Open for read only. If file does not exist, error

`DFACC_CREATE` - If file exists, delete it, then open a new file for read/write.

`DFACC_WRITE` - Open for read/write. If file does not exist, create it.

If a file is opened and an attempt is made to reopen the file using `DFACC_CREATE`, HDF will issue the error code `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` or `DFACC_WRITE`, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the specified file is opened with the relevent 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, n_dds
```

Hputbit

intn Hputbit(int32 *h_id*, intn *bit*)

h_id	IN:	Bit-access element identifier
bit	IN:	Bit to be written

Purpose Writes one bit to the specified bit-access element.

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

Description This function is a wrapper for **Hbitwrite**.

Hputelement

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

file_id	IN:	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 Writes a data element or replaces an existing data element in a HDF file.

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

Hread

int32 Hread(int32 *h_id*, int32 *length*, VOIDP *data*)

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

Purpose Reads the next segment in a data element.

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

Description **Hread** begins reading at the current file position, reads the specified number of bytes, and increments the current file position by one. 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.

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 of the offset origin

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

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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* and the *offset* arguments determine the byte location for the access identifier. If *origin* is set to `DF_START`, the offset is added to the beginning of the data element. If *origin* is set to `DF_CURRENT`, the offset is added to the current position of the access identifier.

Valid values for *origin* are: `DF_START` (the beginning of the file) or `DF_CURRENT` (the current position in the file).

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.

Hsetlength

int32 Hsetlength(int32 *file_id*, int32 *length*)

file_id	IN:	File identifier returned by **Hopen**
length	IN:	Length of the new element

Purpose Specifies the length of a new HDF element.

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

Description This function can only be used when called after **Hstartaccess** on a new data element and before any data is written to that element.

Hshutdown

int32 Hshutdown( )

Purpose	Deallocates buffers previously allocated in other H routines.
Return value	Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise..
Description	Should only be called by the function **HDFend**.

Htagnewref

int32 Htagnewref(int32 *file_id*, uint16 *tag*)

file_id	IN:	Access identifier returned by **Hstartread** or **Hnextread**
tag	IN:	Tag to be identified with the returned reference number

Purpose Returns a reference number that is unique for the specified file that will correspond to the specified tag. Creates a new tag/refernce number pair.

Return value Returns the reference number if successful and `0` otherwise.

Description Successive calls to **Hnewref** will generate a increasing sequence of reference number values until the highest possible reference number value has been returned. It will then return unused reference number values starting from `1` in increasing order.

Htrunc

int32 Htrunc(int32 *h_id*, int32 *trunc_len*)

h_id	IN:	Access identifier returned by **Hstartread** or **Hnextread**
trunc_len	IN:	Length to truncate element

Purpose Truncates the data object specified by the *h_id* to the length *trunc_len*.

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

Description **Htrunc** does not handle special elements.

Hwrite

int32 Hwrite(int32 *h_id*, int32 *length*, VOIDP *data*)

h_id	IN:	Access identifier returned by **Hstartwrite**
len	IN:	Length of segment to be written
data	IN:	Pointer to the data to be written

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` (or `-1`) otherwise.

Description **Hwrite** begins writing at the current position of the access identifier, writes the specified number of bytes, then moves the access identifier to the position immediately following the last accessed byte. Calling **Hwrite** with *length* = `0` results iu an error condition. 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 more than one data element in a file.

HDFclose/hdfclose

intn HDFclose(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**

Purpose Closes the access path to the file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 code `DFE_OPENAID` is returned and the file is not closed. This is a common occurrence when developing new interfaces. See **Hendaccess** for further discussion of this problem.

FORTRAN

```
integer function hdfclose(file_id)

integer file_id
```

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:	File access code
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` (or `-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 file baccess code definitions:

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

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, n_dds
```

HEclear

VOID HEclear(void)

Purpose Clears all information on reported errors from 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.

HEprint/heprnt

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 to the error stack.

Return value None.

Description If *level* is `0`, 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: the ASCII description of the error, the reporting routine, the reporting routine as source file name, and the line at which the error was reported. If the programmer has supplied extra information by means of **HEreport**, this information is printed as well.

The Fortran-77 routine uses one less parameter than the C routine because it doesn't allow the user to specify the print stream. Instead, it always prints to `stdout`.

FORTRAN

```
integer heprnt(level)

integer level
```

HEpush

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

error_code	IN:	HDF error code corresponding to 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.

HEreport

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

format	IN:	Output string specification

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

Return value None

Description **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 format argument must conform to the string specification requirements 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;
}
```

HEstring

char *HEstring(int16 *error_code*)

error_code	IN:	HDF error code

Purpose Returns the error message associated with specified error code.

Return value Returns a pointer to a string associated with the error code if successful.

Description Returns a text description of the given error code. These strings are statically declared and should not be deallocated from memory (using the `free` routine) by the user. If a defined text description cannot be found a generic default message is returned.

HEvalue

int16 HEvalue(int32 *level*)

level	IN:	Level of the error stack to be returned

Purpose Returns an error from the specified level of the error stack.

Return value The error code if successful for `DFE_NONE` otherwise.

HXsetcreatedir/hxiscdir

intn HXsetcreatedir(const char **dir*)

dir	IN:	Target directory of the external file to be written

Purpose Initializes the directory environment variable, identifying the location of the external file to be written.

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

Description The contents of *dir* is copied into the private memory of the HDF library. If *dir* is NULL, the directory variable is unset. If **HXsetcreatedir** encounters an error condition, the directory variable is not changed. When a new external element is created (via the **HXcreate** or **SDsetexternal** routines), the HDF library accesses the external file just like the **open** call by default. Refer to the Reference Manula page on **HXcreate** for a description of when a new or an old file should be opened.

Users may override the default action by calling **HXsetcreatedir** or by defining the environment variable $HDFEXTCREATEDIR. The HDF library will access the external file in the directory according to the environment variable setting. The precedence is **HXsetcreatedir**, then $HDXEXTDIR, in the manner of **open**.

Note that the above override does not apply to absolute pathnames - i.e., filenames starting with a forward slash. HDF will access the absolute pathname without change. Also note that **HXsetcreatedir** and $HDFEXTCREATEDIR are not symmetrical to **HXsetdir** and $HDFEXTDIR. The former pair permits only single directory values and is used to compose the filename for access. The later pair permits multiple directory values which are used for searching an existing file.

The *dir_len* parameter in the Fortran-77 routine specifies the length of the *dir* character string.

FORTRAN

```
integer function hxiscdir(dir, dir_len)

character* (*) dir
integer dir_len
```

HXsetdir/hxisdir

intn HXsetdir(const char **dir*)

dir	IN:	Target directory of the external file to be located

Purpose Initializes the directory environment variable, identifying the location of the external file to be located.

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

Description **HXsetdir** sets the directory variable for locating an external file according to *dir* which may contain multiple directories separated by vertical bars (e.g., "dir1|dir2"). The content of *dir* is copied into the private memory of the HDF library. If *dir* is NULL, the directory variable is unset.

If **HXsetdir** encounters any error, the directory variable is not changed. By default, the HDF library locates the external file just like the **open** call. It also searches for the external file in the directories specified by the user environment variable $HDFEXTDIR, if defined, and the directory variable set by **HXsetdir**. The searching precedence is directory variable, if set, then $HDXEXTDIR, then in the manner of **open**.

The searching differs if the external filename is an absolute pathname - i.e., starting with a forward slash. HDF will try **open** first. If **open** fails and if $HDFEXTDIR is defined or the directory variable is set via **HXsetdir**, HDF will remove all directory components of the absolute pathname (e.g., "/usr/groupA/projectB/Data001" becomes "Data001") and search for that filename with the strategy described in the previous paragraph.

The *dir_len* parameter in the Fortran-77 routine specifies the length of the *dir* character string.

FORTRAN

```
integer function hxisdir(dir, dir_len)

character* (*) dir
integer dir_len
```

SDattrinfo/sfgainfo

intn SDattrinfo(int32 [*file*, *sds*, *dim*]_*id*, int32 *attr_index*, char **attr_name*, int32 **data_type*, int32 **count*)

[*file*, *sds*, *dim*]_*id*	IN:	Identifier of the object the attribute is to be attached to: an *file_id* for a file, an *sds_id* for an SDS or a *dim_id* for a dimension
attr_index	IN:	Index of the attribute to read
attr_name	OUT:	Name assigned to the attribute/dataset/dimension
data_type	OUT:	Data type of the attribute values
count	OUT:	Total number of values in the specified attribute

Purpose Retrieves information about a global or local attribute.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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**(*data_type*) bytes long.

FORTRAN

```
integer function sfgainfo([file, sds, dim]_id, attr_index,
     attr_name, data_type, count)

character* (*) attr_name
integer [file, sds, dim]_id, attr_index, data_type, count
```

SDcreate/sfcreate

int32 SDcreate(int32 *sd_id*, char **name*, int32 *data_type*, int32 *rank*, int32 *dimsizes*[])

sd_id	IN:	SD interface identifier returned from **SDstart**
name	IN:	ASCII string defining a variable name
data_type	IN:	Data type for the values in the dataset
rank	IN:	Number of dimensions in the dataset
dimsizes	IN:	Size of each dimension

Purpose Creates a new dataset.

Return value Returns the *sds_id* if successful and `FAIL` (or `-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 dataset has been created, it is impossible to change its name, data type, or rank. However, it is possible to create a dataset 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 dataset, 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 to this dimension at will. In Fortran-77, *dimsizes*(*rank*) is the only dimension that can be unlimited. This is useful when the eventual size of a dataset is not known at creation time.

Valid values for *data_type* are prefaced by DFNT_. The following are valid symbolic names and their data 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
```

```
8-bit unsigned character DFNT_CHAR84
```

MAX_VAR_DIMS is the maximum rank a dataset can have.

FORTRAN

```
integer function sfcreate(sd_id, name, data_type, rank,
    dimsizes)

character* (*) name
integer sd_id, data_type, rank, dimsizes(*)
```

SDdiminfo/sfgdinfo

intn SDdiminfo(int32 *dim_id*, char **name*, int32 **count*, int32 **data_type*, int32 **nattrs*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**
name	OUT:	Array to retrieve dimension name
count	OUT:	Size of the dimension
data_type	OUT:	Data type of the data stored in the dataset
nattrs	OUT:	Attribute count assigned to the dimension's coordinate variable

Purpose Retrieves information about a dimension.

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

Description If scale information has been stored for this dimension via **SDsetdimscale**, *data_type* will contain the data type - otherwise, *data_type* will be `0`. If the user has not named the dimension via **SDsetdimname** a place holder will be generated by the library. If the name is unimportant, `NULL` can be passed in for the *name* parameter.

The *count* argument is set to `SD_UNLIMITED` for unlimited dimensions. To get the number of records of an unlimited dimension, use **SDgetinfo**.

If "label, unit, format" metadata (or "LUF" metadata) is assigned to a dimenstion, it will be "promoted" by the SD interface to be a variable. If the target dimension specfied by *dim_id* is a varaible, the value returned in the *data_type* argument will be `0`.

FORTRAN

```
integer function sfgdinfo(dim_id, name, count, data_type,
                              nattrs)

character* (*) name
integer dim_id, count, data_type, nattrs
```

SDend/sfend

intn SDend(int32 *sd_id*)

sd_id	IN:	SD interface identifier returned from **SDstart**

Purpose Closes the file and frees memory allocated by the library.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 data in-core are likely not to be flushed to the file.

FORTRAN

```
integer function sfend(sd_id)

integer sd_id
```

SDendaccess/sfendacc

intn SDendaccess(int32 *sds_id*)

sds_id	IN:	Dataset identifier returned from **SDselect**

Purpose Disposes of a dataset identifier.

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

Description When done interacting with a specific dataset, 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 sds_id
```

SDfileinfo/sffinfo

intn SDfileinfo(int32 *sd_id*, int32 **ndatasets*, int32 **nglobal_attr*)

sd_id	IN:	SD interface identifier returned from **SDstart**
ndatasets	OUT:	Number of datasets in the file
nglobal_attr	OUT:	Number of global attributes in the file

Purpose Determines the number of datasets and global attributes in a file.

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

Description **SDfileinfo** returns the number of currently defined datasets 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 datasets. For example, "file_creator_date" would be a global attribute, while "units" would typically be dataset-specific. Global attributes are created by **SDsetattr** using the *sd_id* rather than an *sds_id*.

FORTRAN

```
integer function sffinfo(sd_id, ndatasets, nglobal_attr)

integer sd_id, ndatasets, nglobal_attr
```

SDfindattr/sffattr

int32 SDfindattr(int32 [*file*, *sds*, *dim*]_*id*, char **attr_name*)

[*file*, *sds*, *dim*]_*id*	IN:	Identifier of the object the attribute is to be attached to: an *file_id* for a file, an *sds_id* for an SDS or a *dim_id* for a dimension
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` (or `-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 *attr_name* parameter.

FORTRAN

```
integer function sffattr([file, sds, dim]_id, attr_name)

integer [file, sds, dim]_id
character* (*) attr_name
```

SDgetcal/sfgcal

intn SDgetcal(int32 *sds_id*, float64 **cal*, float64 **cal_err*, float64 **offset*, float64 **offset_err*, int32 **data_type*)

sds_id	IN:	Dataset identifier returned from **SDselect**
cal	OUT:	Calibration factor
cal_err	OUT:	Calibration error
offset	OUT:	Uncalibrated offset
offset_err	OUT:	Uncalibrated offset error
data_type	OUT:	Data type of uncalibrated data

Purpose Retrieves the calibration information associated with the specified dataset.

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

Description **SDgetcal** reads the calibration record attached to a dataset. 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
data_type	data type of uncalibrated data

The relationship between a value `iy` stored in a dataset 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, data_type)

integer sds_id, data_type
real*8 cal, cal_err, offset, offset_err
```

SDgetchunkinfo

intn SDgetchunkinfo(int32 *sds_id*, HDF_CHUNK_DEF **cdef*, int32 **flags*)

sds_id	IN:	Dataset identifier returned from **SDselect**
cdef	OUT:	Union structure containing information about the chunks in the SDS
flags	OUT:	Flags determining routine behavior

Purpose Obtains chunking information about a scientific dataset.

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

Description A pointer to a `chunk_lengths` array (defined in the `comp_info` union pointed to by *cdef*) containing chunk dimension size information will be returned if the value of `flags` is `HDF_CHUNK`, and a pointer to a `comp_info` structure containing the dimension size of the chunks and compression information will be returned if the value of `flags` is `HDF_CHUNK` bitwise-OR'ed with `HDF_COMP`.

For the definition of the `HDF_CHUNK_DEF` union, refer to the Reference Manual page for **SDsetchunk**.

SDgetchunkinfo can also be used to determine if the target SDS is not chunked. In this case, the value of `flags` is set to `HDF_NONE` and the *cdef* parameter is ignored

A `NULL` value can also be passed in as the `cdef` parameter if chunking information is not desired.

SDgetdatastrs/sfgdtstr

intn SDgetdatastrs(int32 *sds_id*, char **label*, char **unit*, char **format*, char **coordsys*, intn *len*)

sds_id	IN:	Dataset identifier returned from **SDselect**
label	OUT:	Label describing 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 dataset.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 sds_id, len
character* (*) label, unit, format, coordsys
```

SDgetdimid/sfdimid

int32 SDgetdimid(int32 *sds_id*, intn *dim_number*)

sds_id	IN:	Dataset identifier returned from **SDselect**
dim_number	IN:	Number of dimensions

Purpose	Retrieves the dimension identifier associated with the specified dimension in the dataset.
Return valu	Returns a *dim_id* if successful and `FAIL` (or `-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 sds_id, dim_number`

SDgetdimscale/sfgdscale

intn SDgetdimscale(int32 *dim_id*, VOIDP *data*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**
data	OUT:	Buffer for the scale values for the specified dimension

Purpose Retrieves the scale values for the specified dimension.

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

Description It is assumed that the user has already called **SDdiminfo** and has thus allocated sufficient space to hold the values. **SDdiminfo** is used to check if a scale has been set for the target dimension. If the data type parameter value returned by **SDdiminfo** is `0`, no scale has been set and **SDgetdimscale** should not be called.

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

FORTRAN

```
integer function sfgdscale(dim_id, data)

integer dim_id
<valid numeric data type> data
```

SDgetdimstrs/sfgdmstr

intn SDgetdimstrs(int32 *dim_id*, char **label*, char **unit*, char **format*, intn *len*)

dim_id	IN:	Dimension identifier 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 Retrieves the label, unit, and format strings for a given dimension.

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

Description This routine returns the dimension 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 returned, `NULL` can be passed for any of the string values.

FORTRAN

```
integer function sfgdmstr(dim_id, label, unit, format, len)

integer dim_id, len
character* (*) label, unit, format
```

SDgetfillvalue/sfgfill/sfgcfill

intn SDgetfillvalue(int32 *sds_id*, VOIDP *fill_val*)

sds_id	IN:	Dataset identifier returned from **SDselect**
fill_val	OUT:	Buffer for the returned fill value

Purpose Reads the fill value for the given dataset if it exists.

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

Description It is assumed that the data type of the fill value is the same as for the dataset.

Note that there are two Fortran-77 versions of this routine: **sfgfill** and **sfgcfill**. The **sfgfill** routine reads numeric fill value data and **sfgcfill** reads character fill value data.

FORTRAN

```
integer function sfgfill(sds_id, fill_val)

integer sds_id
<valid numeric data type> fill_val


integer function sfgcfill(sds_id, fill_val)

integer sds_id
character* (*) fill_val
```

SDgetinfo/sfginfo

intn SDgetinfo(int32 *sds_id*, char **sds_name*, int32 **rank*, int32 *dimsizes*[], int32 **data_type*, int32 **nattrs*)

sds_id	IN:	Dataset identifier returned from **SDselect**
sds_name	OUT:	Buffer for the name, if any, of the dataset
rank	OUT:	Buffer for the number of dimensions in the dataset
dimsizes	OUT:	Buffer for the size of each dimension in the dataset
data_type	OUT:	Buffer for the data type for the data stored in the dataset
nattrs	OUT:	Buffer for the number of "netCDF-style" attributes for this dataset

Purpose Retrieves the name, rank, dimension sizes, data type and attribute count for the specified dataset.

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

Description Returns basic information about a given dataset. All fields must be provided. The dataset name can be at most `MAX_NC_NAME` characters long and the rank of the dataset is limited to `MAX_VAR_DIMS`.

In the case of unlimited dimensions, the *dimsizes*[0] argument returns the number of records in the dimension.

FORTRAN

```
integer function sfginfo(sds_id, sds_name, rank, dimsizes,
    data_type, nattrs)

character* (*) sds_name
integer sds_id, rank, dimsizes(*)
integer data_type, nattrs
```

SDgetrange/sfgrange

intn SDgetrange(int32 *sds_id*, VOIDP *max*, VOIDP *min*)

sds_id	IN:	Dataset identifier returned from **SDselect**
max	OUT:	Highest value in the range
min	OUT:	Lowest value in the range

Purpose Retrieves the maximum and minimum values as they are stored with the scientific dataset.

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

Description The maximum and minimum values must be set via a call to **SDsetrange**. They are not automatically stored when a dataset is written to a file. As the max imumand minimum values are supposed to relate to the data itself, it is assumed that they are of the same data 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-77 version they are simple variables of the same data type as the data array.

FORTRAN

```
integer function sfgrange(sds_id, max, min)

integer sds_id
<valid numeric data type>  max, min
```

SDidtoref/sfid2ref

int32 SDidtoref(int32 *sds_id*)

sds_id	IN:	Dataset identifier returned from **SDselect**

Purpose Retrieves the reference number assigned to the specified dataset.

Return value Returns sds_ref if successful and `FAIL` (or `-1`) otherwise.

Description The specified reference number can be used to add the dataset to a vgroup as well as a means of using the HDF annotations interface to annotate the dataset.

FORTRAN

```
integer function sfid2ref(sds_id)

integer sds_id
```

SDiscoordvar/sfiscvar

intn SDiscoordvar(int32 *sds_id*)

sds_id	IN:	Dataset identifier returned from **SDselect**

Purpose Determines if a dataset is a coordinate variable

Return value Returns `TRUE` if the dataset is a coordinate variable, and `FALSE` otherwise.

Description Coordinate variables are created to store metadata associated with dimensions. Due to netCDF compatibility, coordinate variables are also considered datasets which sometimes is a source of confusion.

FORTRAN

```
integer function sfiscvar(sds_id)

integer sds_id
```

SDisdimval_bwcomp/sfisdmvc

intn SDisdimval_bwcomp(int32 *dim_id*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**

Purpose Determines whether the specified dimension *will have* the old and new representations or the new representation only.

Return value Returns SD_DIMVAL_BW_COMP (or 1) if backward compatible, SD_DIMVAL_BW_INCOMP (or 0) if incompatible, FAIL (or -1) if error.

Description **SDstart** reads dimension records into memory. The compatiblity mode of each dimesion is decided by the existance of the dimension vdata of class "DimVal0.0" in the dimension vgroup for that dimension.

If "DimVal0.0" vdata doesn't exist in that dimension vgroup, it will be flagged as backward-incompatible. The compatibility mode can be changed by calls to **SDsetdimval_comp** at any time between the calls to **SDstart** and **SDend**.

FORTRAN

```
integer function sfisdmvc(dim_id)

integer dim_id
```

SDisrecord

int32 SDisrecord(int32 *sds_id*)

sds_id	IN:	SDS identifier returned from **SDselect**

Purpose Determines if a dataset is a record variable

Return value Returns `TRUE` if the dataset is a record variable, and `FALSE` otherwise.

SDnametoindex/sfn2index

int32 SDnametoindex(int32 *sd_id*, char **sds_name*)

sd_id	IN:	SD interface identifier returned from **SDstart**
sds_name	IN:	Name of the dataset to index

Purpose Determines the index assigned to the scientific dataset defined by *sds_name*.

Return value Returns an *sds_index* if successful and `FAIL` (or `-1`) otherwise.

Description The *sds_index* can be passed to **SDselect** to return an *sds_id* for the named dataset. This routine is case sensitive and will not accept wild cards. In addition, this routine will only return the *sds_index* for the first dataset with the given name.

FORTRAN

```
integer function sfn2index(sd_id, sds_name)

integer sd_id
character* (*) sds_name
```

SDreadattr/sfrnatt/sfrcatt

intn SDreadattr(int32 [*file*, *sds*, *dim*]*_id*, int32 *attr_index*, VOIDP *data*)

[*file*, *sds*, *dim*]*_id*	IN:	Identifier of the object the attribute is to be attached to: an *file_id* for a file, an *sds_id* for an SDS or a *dim_id* for a dimension
attr_index	IN:	Index of the attribute to be read
data	OUT:	Buffer for the attribute values

Purpose Reads the values of an attribute.

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

Description It's assumed that the user has called **SDattrinfo** and that the buffer is large enough to store 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.

Note that this routine has two Fortran-77 versions: **sfrnatt** and **sfrcatt**. The **sfrnatt** routine reads numeric attribute data and **sfrcatt** reads character attribute data.

The index returned as the *attr_index* argument is one-based.

FORTRAN

```
integer function sfrnatt([file, sds, dim]_id, attr_index, data)

integer [file, sds, dim]_id, attr_index
<valid numeric data> data


integer function sfrcatt([file, sds, dim]_id, attr_index, data)

integer [file, sds, dim]_id, attr_index
character* (*) data
```

SDreadchunk

intn SDreadchunk(int32 *sds_id*, int32 **origin*, VOIDP *datap*)

sds_id	IN:	SD interface identifier returned from **SDstart**
origin	IN:	Origin of the chunk to be read
datap	OUT:	Buffer for the returned chunk data

Purpose Reads data from a chunked scientific dataset.

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

Description **SDreadchunk** is used when an entire chunk of data is to be read. **SDreaddata** is used when the read operation is to be done regardless of the chunking scheme used in the SDS.

Also, **SDreadchunk** is written specifically for chunked SDSs and and doesn't have the overhead of the additional functionality supported by the **SDreaddata** routine - therefore, it is much faster than **SDreaddata**.

SDreadchunk will return `FAIL` when an attempt is made to use it to read from a non-chunked SDS.

SDreaddata/sfrdata/sfrcdata

intn SDreaddata(int32 *sds_id*, int32 *start*[], int32 *stride*[], int32 *edge*[], VOIDP *buffer*)

sds_id	IN:	Dataset identifier 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 for the data

Purpose Reads a hyperslab of data from a dataset.

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

Description The *start* array specifies the multi-dimensional index of the starting corner of the hyperslab to read. The values are zero-based.

The *edge* array specifies the number of values to read along each dimension of the hyperslab.

The *stride* array 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` in the C interface or *stride* = `1` in either interface specifies contiguous reading of data. If the *stride* values are set to `0`, **SDreaddata** returns `FAIL` (or `-1`). No matter what stride value is provided, data is always placed contiguously in buffer.

If the SDS specified by *sds_id* contains no data, **SDreaddata** returns `0`.

When reading data from a "chunked" SDS using **SDreaddata**, consideration should be given to be issues presented in the section on chunking in Chapter 3 of the HDF User's Manual, tited *Scientific Data Sets (SD API)* and Chapter 13 of the HDF User's Manual, titled *HDF Performance Issues*.

Note that there are two Fortran-77 versions of this routine: **sfrdata** and **sfrcdata**. The **sfrdata** routine reads numeric scientific data and **sfrcdata** reads character scientific data.

FORTRAN

```
integer function sfrdata(sds_id, start, stride, edge, buffer)

integer sds_id, start(*), stride(*), edge(*)
<valid numeric data> buffer(*)

integer function sfrcdata(sds_id, start, stride, edge, buffer)

integer sds_id, start(*), stride(*), edge(*)
```

```
character* (*) buffer
```

SDreftoindex/sfref2index

int32 SDreftoindex(int32 *sd_id*, int32 *sds_ref*)

sd_id	IN:	SD interface identifier returned from **SDstart**
sds_ref	IN:	Reference number for the specified dataset

Purpose Determines the index assigned to a scientific dataset given the specified reference number.

Return value Returns an *sds_index* if successful and `FAIL` (or `-1`) otherwise.

Description The value of *sds_index* can be passed to **SDselect** to return a dataset identifier (*sds_id*).

FORTRAN

```
integer function sfref2index(sd_id, sds_ref)

integer sd_id, sds_ref
```

SDselect/sfselect

int32 SDselect(int32 *sd_id*, int32 *sds_index*)

sd_id	IN:	SD interface identifier returned from **SDstart**
sds_index	IN:	Index of the dataset

Purpose Retrieves the *sds_id* for the given dataset.

Return value Returns the *sds_id* if successful and `FAIL` (or `-1`) otherwise.

Description To get an *sds_id* for the Nth dataset, use `N-1` as the index. `N` must be a number greater than or equal to `0` and less than the total number of datasets in the file. The total number of datasets in a file may be obtained from a call to **SDfileinfo**. The function **SDnametoindex** can be used to find the index of a dataset if its name is known.

The integration with netCDF has required that dimension metadata be stored as "coordinate variables." This sometimes causes problems when calling **SDselect** as the coordinate variables can lead to different dataset ordering than is expected. In situations such as these, users should use the routine **SDiscoordvar** to determine if a given dataset is a coordinate variable or not.

FORTRAN

```
integer function sfselect(sd_id, sds_index)

integer sd_id, sds_index
```

SDsetaccesstype/sfsacct

intn SDsetaccesstype(int32 *sd_id*, int32 *access_type*)

sd_id	IN:	SD interface identifier returned from **SDstart**
access_type	IN:	I/O access mode of the dataset

Purpose Determines the I/O access mode to be used for the specified dataset.

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

Description The dataset must be an external element and the I/O access defined via this routine applies to the specified external file only.

The *access_type* parameter can be defined as DFACC_SERIAL for sequential I/O or DFACC_PARALLEL for parallel I/O. At present, HDF has only implemented the CM5 CMFS parallel I/O.

Though the access mode is set for dataset *sds_id*, it applies to the entire external file in which the dataset resides. If different access modes are defined for datasets within the same HDF external file, unpredictable effects will result. It is also the users' responsibility to make sure the access mode is appropriate for the files involved. For example, setting DFACC_PARALLEL on a dataset contained in a serial file may result in slower I/O rates and sometimes failures.

FORTRAN

```
integer function sfsacct(sd_id, access_type)

integer sd_id, access_type
```

SDsetattr/sfsnatt/sfscatt

intn SDsetattr(int32 [*file*, *sds*, *dim*]_*id*, char **attr_name*, int32 *data_type*, int32 *count*, VOIDP *values*)

[*file*, *sds*, *dim*]_*id*	IN:	Identifier of the object the attribute is to be attached to: an *file_id* for a file, an *sds_id* for an SDS or a *dim_id* for a dimension
attr_name	IN:	Name to be assigned to the attribute
data_type	IN:	Data type of the values in the attribute
count	IN:	Total number of values to be stored in the attribute
values	IN:	Data values to be storde in the attribute

Purpose Defines a new type of attribute for the given variable.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 data 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 datasets. For example, "file_creator_date" is a global attribute, whereas "units" is typically considered to be dataset-specific.

Valid values for *data_type* are prefaced by DFNT_. The following are valid symbolic names and their data 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

```
8-bit signed character DFNT_CHAR   4
```

Note that there are two Fortran-77 versions of this routine: **sfsnatt** and **sfscatt**. The **sfsnatt** routine writes numeric attribute data and **sfscatt** writes character attribute data.

Example Store a "valid_range" attribute for a dataset.

```
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 sfsnatt([file, sds, dim]_id, attr_name,
                         data_type, count, values)

character* (*) attr_name
integer [file, sds, dim]_id, data_type, count
<valid numeric data type> values(*)

integer function sfscatt([file, sds, dim]_id, attr_name,
                         data_type, count, values)

character* (*) attr_name, values
integer  [file, sds, dim]_id, data_type, count
```

SDsetblocksize

intn SDsetblocksize(int32 *sd_id*, int32 *block_size*)

sd_id	IN:	SD interface identifier returned from **SDstart**
block_size	IN:	Size of the block in bytes

Purpose Sets the block size used for storing data datasets with unlimited dimensions.

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

Description **SDsetblocksize** is used when creating new datasets only - it has no affect on pre-existing datasets. The *block_size* parameter should be set to a multiple of the "slice" size.

SDsetcal/sfscal

intn SDsetcal(int32 *sds_id*, float64 *cal*, float64 *cal_err*, float64 *offset*, float64 *offset_err*, int32 *data_type*)

sds_id	IN:	Dataset identifier returned from **SDselect**
cal	IN:	Calibration factor
cal_err	IN:	Calibration error
offset	IN:	Uncalibrated offset
offset_err	IN:	Uncalibrated offset error
data_type	IN:	Data type of uncalibrated data

Purpose Sets the calibration information.

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

Description **SDsetcal** stores the calibration record associated with a dataset. A calibration record contains the following information:

cal	Calibration factor
cal_err	Calibration error
offset	Uncalibrated offset
offset_err	Uncalibrated offset error
data_type	Data type of uncalibrated data

The relationship between a value `iy` stored in a dataset 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 dataset that is to be written.

FORTRAN

```
integer function sfscal(sds_id, cal, cal_err, offset,
    offset_err, data_type)

integer sds_id, data_type
real*8 cal, cal_err, offset, offset_err
```

SDsetchunk

intn SDsetchunk(int32 *sds_id*, HDF_CHUNK_DEF *cdef*, int32 *flags*)

sds_id	IN:	SD interface identifier returned from **SDstart**
cdef	IN:	Union containing information on how the chunks are to be defined
flags	IN:	Flags determining the behavior of the routine

Purpose Determines the chunk size and the compression method, if any, to be applied when partitioning the array into chunks.

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

Description **SDsetchunk** receives it's information on how to partition the array into chunks, as well as compression information, from a `HDF_CHUNK_DEF` union passed in as its second argument. This union structure is defined in the HDF library as follows:

```
typedef union hdf_chunk_def_u {
    int32 chunk_lengths[MAX_VAR_DIMS];
    struct {
        int32 chunk_lengths[MAX_VAR_DIMS];
        int32 comp_type;
        comp_info cinfo;
    } comp;
} HDF_CHUNK_DEF
```

The `flags` parameter can either be set to `HDF_CHUNK` if the SDS is to be uncompressed, or to the bitwise-OR'ed values of `HDF_CHUNK` and `HDF_COMP` (`HDF_CHUNK | HDF_COMP`) if a compression method is to be applied to the array while it is being partitioned into chunks.

SDsetchunkcache

intn SDsetchunkcache(int32 *sds_id*, int32 *maxcache*, int32 *flags*)

sds_id	IN:	SD interface identifier returned from **SDstart**
maxcache	IN:	Maximum number of chunks in the cache
flags	IN:	Flags determining the behavior of the routine

Purpose Determines the size of the chunk cache.

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

Description By default, when a generic SDS is promoted to be a chunked SDS, the `max-cache` parameter is set to the number of chunks along the last dimension and a cache for the chunks is created.

If the chunk cache is full and the value of the `maxcache` parameter is larger than the currently allowed maximum number of cached chunks, then the maximum number of cached chunks is reset to the value of `maxcache`. If the chunk cache is not full, then the size of the chunk cache is reset to the value of `maxcache` only if it is greater than current number of chunks in the cache.

Never set the value of `maxcache` to be less than the number of chunks along the last dimension of the biggest slab to be written or read via **SDreaddata** or **SDwritedata**. Doing this will cause internal thrashing. See the section on chunking in Chapter 13 of the HDF User's Guide, titled *HDF Performance Issues*, for more information on this.

Currently the only allowed value of the `flags` parameter is `0`, which designates default operation. In the near future, the value `HDF_CACHEALL` will be supported to be used to specifiy that the whole SDS object is to be cached.

SDsetcompress

intn SDsetcompress(int32 *sd_id*, char **comp_type*, comp_info **cinfo*)

sd_id	IN:	SD interface identifier returned from **SDstart**
comp_type	IN:	Compression method
cinfo	IN:	Pointer to compression information structure

Purpose Sets the compression method for the specified dataset.

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

Description The **SDsetcompress** routine is a simplified interface to the **HCcreate** routine, and should be used instead of **HCcreate** unless the user is familiar with working with the lower-level routines..

The `comp_type` parameter is the compression type definition and is set to `COMP_CODE_RLE` for run-length encoding, `COMP_CODE_DEFLATE` for Gnu ZIP (or GZIP) compression, `COMP_CODE_SKPHUFF` for skipping Huffman or `COMP_CODE_NONE` for no compression. The `c_info` parameter is a union structure of type `tag_comp_info`.

SDsetcompress compresses the dataset data at the time it is called, not during the next call to **SDwritedata**.

SDsetdatastrs/sfsdtstr

intn SDsetdatastrs(int32 *sds_id*, char **label*, char **unit*, char **format*, char **coordsys*)

sds_id	IN:	Dataset identifier 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 dataset.

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

Description This routine can be used to set the label, unit, format, and coordinate system strings for a given dataset. `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 sds_id
character* (*) label, unit, format, coordsys
```

SDsetdimname/sfsdmname

intn SDsetdimname(int32 *dim_id*, char **dim_name*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**
dim_name	IN:	ASCII string to name dimension

Purpose Assigns a name to the specified dimension.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 default names generated by the HDF library.

FORTRAN

```
integer function sfsdmname(dim_id, dim_name)

integer dim_id
character* (*) dim_name
```

SDsetdimscale/sfsdscale

intn SDsetdimscale(int32 *dim_id*, int32 *count*, int32 *data_type*, VOIDP *data*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**
count	IN:	Total number of values along the specified dimension
data_type	IN:	Data type of the values along the specified dimension
data	IN:	Value of each increment along the specified dimension

Purpose Stores the values of the specified dimension.

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

Description Users should note that it is possible to store values for a dimension without naming it. Even though its redundant, the *count* argument has been included for backward compatibility. Note that there is now a *data_type* parameter so the dimension scales are no longer required to be the same data type as the dataset.

FORTRAN

```
integer function sfsdscale(dim_id, count, data_type, data)

integer dim_id, count, data_type
character* (*) data
```

SDsetdimstrs/sfsdmstr

intn SDsetdimstrs(int32 *dim_id*, char **label*, char **unit*, char **format*)

dim_id	IN:	Dimension identifier 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` (or `0`) if successful and `FAIL` (or `-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 dim_id
character* (*) label, unit, format
```

SDsetdimval_comp/sfsdmvc

intn SDsetdimval_comp(int32 *dim_id*, intn *comp_mode*)

dim_id	IN:	Dimension identifier returned from **SDgetdimid**
comp_mode	IN:	Compatibility mode to be set

Purpose Determines whether the specified dimension *will have* the old and new representations or the new representation only.

Return value Returns either SUCCEED (or 0) on successful completion, or FAIL (or -1) otherwise.

Description The `comp_mode` parameter is set to either `SD_DIMVAL_BW_COMP`, which specifies compatible mode and that the old and new dimension representations will be written to file , or `SD_DIMVAL_BW_INCOMP`, which specifies backward incompatible mode and that only the new dimension representation will be written to file.

Unlimited dimensions are always backward-compatable, therefore **SDsetdimval_comp** takes no action on unlimited dimensions.

As of HDF4.1r1, the default mode is backward-incompatable. Subsequent calls to **SDsetdimval_comp** will override the settings established in previous calls to the routine.‘

FORTRAN

```
integer function sfsdmvc(dim_id, comp_mode)

integer dim_id, comp_mode
```

SDsetexternalfile/sfsextf

intn SDsetexternalfile(int32 *sds_id*, char **filename*, int32 *offset*)

sds_id	IN:	Dataset identifier 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` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

Description **SDsetexternalfile** allows users to move the data values for a dataset into an external data file. Only the dataset values can be stored externally, all other data must stay in the original file.

Data can only be moved once for any given dataset and it is the user's responsibility to make sure the external data file is kept with the "original" file. The *offset* parameter 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 post-version 3.2 files (i.e. calling on a netCDF file that was opened with the multi-file interface will fail).

If the SDS specified by *sds_id* already exists, its data will be moved to the external file and the connection between the tag and the data in the primary file will be broken, effectively making that data inaccessible to the interface routines. However, the data itself must be explicitly deleted by the `hdfpack` command-line utility, as **SDsetexternalfile** doesn't do this.

See the Reference Manual entries for **HXsetcreatedir** and **HXsetdir** for more information on the options available for accessing external files.

FORTRAN

```
integer function sfsextf(sds_id, file_name, offset)

integer sds_id, offset
character* (*) file_name
```

SDsetfillmode/sfsflmd

intn SDsetfillmode(int32 *file_id*, intn *fill_mode*)

file_id	IN:	Dataset identifier returned from **SDselect**
fill_mode	IN:	Fill mode

Purpose Sets the fill mode for the specified file.

Return value Returns the previous fill mode if successful and `FAIL` (or `-1`) otherwise.

Description **SDsetfillmode** sets the fill mode to be applied to all SDSs contained in the specified file.

When the specified file is first opened, or when the file is closed then re-opened, its fill mode will be the default SD_FILL mode, which indicates that fill values will be written when the SDS is created.

Possible values for `fill_mode` are: SD_FILL (or `0`) or SD_NOFILL (or 256), defined in "hdf.inc". SD_NOFILL indicates that fill values will not be written.

When an SDS without unlimited dimensions is created, by default the first **SDwritedata** call will fill the entire dataset with the user-defined fill value, or the default fill value if there is no user-defined fill value, during the call to **SDsetfillvalue**. In SDSs with an unlimited dimension defined as the slowest-changing dimension, if a new write operation takes place along the unlimited dimension beyond the last location of the previous write operation, the array locations between these written areas will be initialized to the user-defined fill value, or the default fill value if a user-defined fill value hasn't been specified.

If it is certain that all dataset values will be written before any read operation takes place, there is no need to write the fill values. Simply call **SDsetfillmode** with a `fill_mode` value of SD_NOFILL which will eliminate all fill value write operations to the dataset. For large datasets, this can improve the speed by almost 50%.

FORTRAN

```
integer function sfsflmd(file_id, fill_mode)

integer file_id, fill_mode
```

SDsetfillvalue/sfsfill

intn SDsetfillvalue(int32 *sds_id*, VOIDP *fill_value*)

sds_id	IN:	Dataset identifier returned from **SDselect**
fill_value	IN:	Fill value

Purpose Sets the fill value for the specified dataset.

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

Description The fill value is assumed to have the same data type as the dataset.

FORTRAN

```
integer function sfsfill(sds_id, fill_value)

integer sds_id, fill_value
```

SDsetnbitdataset/sfsnbit

intn SDsetnbitdataset(int32 *sds_id*, intn *start_bit*, intn *bit_len*, intn *sign_ext*, intn *fill_one*)

sds_id	IN:	Dataset identifier returned from **SDselect**
start_bit	IN:	Leftmost bit of the field to bewritten
bit_len	IN:	Length of the bit field to bewritten
sign_ext	IN:	Sign extend specifier
fill_one	IN:	Background bit specifier

Purpose Specifies a non-standard bit length for SDS data.

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

Description **SDnbitdataset** allows the HDF user to specify that a particular SDS array contains data of a non-standard length. Any length between 1 and 32 bits can be specified. After **SDnbitdataset** has been called for the SDS array, any read or write operations will involve a conversion between the new data length of the SDS array and the data length of the read or write buffer.

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

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

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

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

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

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

This bit operation is performed before the sign-extend bit-filling. For example, using the `sign_ext` example above, bits 0 to 16 and 26 to 31 will first be set to the "background" bit value, and then bits 26 to 31 will be set to `1` or `0` based on the value of the 25th bit.

FORTRAN

```
integer function sfsnbit(sds_id, start_bit, bit_len, sign_ext,
        fill_one)

integer sds_id, start_bit, bit_len, sign_ext, fill_one
```

SDsetrange/sfsrange

intn SDsetrange(int32 *sds_id*, VOIDP *max*, VOIDP *min*)

sds_id	IN:	Dataset identifier returned from **SDselect**
max	IN:	Highest value in the range
min	IN:	Lowest value in the range

Purpose Sets the "valid" maximum and minimum values for the given dataset.

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

Description It is assumed that the data type for the maximum and minimum values are the same as the data 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-77 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 sds_id
<valid numeric data type>  max, min
```

SDstart/sfstart

int32 SDstart(char **filename*, int32 *access_mode*)

filename	IN:	Name of the HDF file
access_mode	IN:	The SDS access mode in effect during the current session

Purpose Opens the HDF file and initializes the SD interface.

Return value Returns an *sd_id* if successful and `FAIL` (or `-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* parameter is one of the following:

`DFACC_READ` - Open existing file for read-only access. If the file doesn't exist, specifying this mode will result in an error condition.

`DFACC_WRITE` - Open existing file for read and write access. If the file doesn't exist, specifying this mode will result in an error condition.

`DFACC_CREATE` - Create a new file with read and write access. If the file does exist, the contents of this file will be deleted before any new writes occur (the file contents will be replaced).

The file can be any one of the following:an XDR-based netCDF file, "old-style" DFSD file or a "new-style" SD file

If *access_mode* is set to `DFACC_CREATE` "new-style" SD files will be created. If *access_mode* is set to `DFACC_RDONLY`, the specified file will not be created if it doesn't exist.

The type of identifier returned by **SDstart** is currently not the same as the identifier 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 access_mode
```

SDwritechunk

intn SDwritechunk(int32 *sds_id*, int32 **origin*, VOIDP *datap*)

sds_id	IN:	SD interface identifier returned from **SDstart**
origin	IN:	Origin of the chunk to be written
datap	IN:	Buffer for the chunk data to be written

Purpose Writes data to a chunked scientific dataset.

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

Description **SDwritechunk** is used when an entire chunk of data is to be written. **SDwritedata** is used when the write operation is to be done regardless of the chunking scheme used in the SDS.

Also, **SDwritechunk** is written specifically for chunked SDSs and doesn't have the overhead of the additional functionality supported by the **SDwritedata** routine - therefore, it is much faster than **SDwritedata**.

SDwritechunk will return `FAIL` when an attempt is made to use it to write to a non-chunked SDS.

SDwritedata/sfwdata/sfwcdata

intn SDwritedata(int32 *sds_id*, int32 *start*[], int32 *stride*[], int32 *edge*[], VOIDP *data*)

sds_id	IN:	Dataset identifier 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 be written along each dimension
data	IN:	Values to be written to the dataset

Purpose Writes a hyperslab of data for a dataset.

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

Description The array *start* specifies 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 *stride* array 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` in the C interface or *stride* = `1` in either interface specifies the contiguous reading of data. If the *stride* values are set to `0`, **SDreaddata** returns `FAIL` (or `-1`). No matter what stride value is provided, data is always placed contiguously in buffer.

When writing data to a "chunked" SDS using **SDwritedata**, consideration should be given to be issues presented in the section on chunking in Chapter 3 of the HDF User's Manual, tited *Scientific Data Sets (SD API)* and Chapter 13 of the HDF User's Manual, titled *HDF Performance Issues*.

Note that there are two Fortran-77 versions of this routine: **sfwdata** and **sfwcdata**. The **sfwdata** routine writes numeric data and **sfwcdata** writes character scientific data.

FORTRAN

```
integer function sfwdata(sds_id, start, stride, edge, data)

integer sds_id
integer start(*), stride(*), edge(*)
<valid numeric data type> data

integer function sfwcdata(sds_id, start, stride, edge, data)

integer sds_id
integer start(*), stride(*), edge(*)
character* (*) data
```

Vaddtagref/vfadtr

int32 Vaddtagref(int32 *vgroup_id*, int32 *tag*, int32 *ref*)

vgroup_id	IN:	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/reference number pairs 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` (or `-1`) otherwise.

Description This routine is primarily used to add an HDF object that may be 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 vgroup_id, tag, ref
```

Vattach/vfatch

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 Attaches to an existing vgroup or creates a new vgroup and returns an identifier for it.

Return value Returns a vgroup identifier if successful and `FAIL` (or `-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 complete, 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 file_id, vgroup_ref
character access
```

Vattrinfo/vfainfo

intn Vattrinfo(int32 *vgroup_id*, intn *attr_index,* char **attr_name,* char **data_type,* intn **count,* intn **size*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_index	IN:	Index of the target attribute
attr_name	OUT:	Name of the target attribute
data_type	OUT:	Data type of the target attribute
count	OUT:	Number of values in the target attribute
size	OUT:	Size, in bytes, of the values of the target attribute.

Purpose Returns the name, data type, number of values, and the size of the values of the specified attribute of the specified vgroup identified by *vgroup_id* and *attr_index*.

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

Description The values of the *attr_name*, *data_type*, *count* and *size* parameters can be set to `NULL`, if the information returned by these parameters are not needed.

The value of *attr_index* parameter is used as the index of the target vgroup attribute, and is zero-based. For example, a *attr_index* value of `4` would refer to the fifth attribute of the vgroup.

FORTRAN

```
integer function vfainfo(vgroup_id, attr_index, attr_name,
        data_type, count, size)

integer vgroup_id, attr_index, data_type, count, size
character* (*) attr_name
```

Vdetach/vfdtch

int32 Vdetach(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Detaches a currently-attached vgroup, terminating further access to that vgroup.

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

Description All space associated with the specified vgroup will be freed. Each attached vgroup should be detached by calling 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

```
integer function vfdtch(vgroup_id)

integer vgroup_id
```

Vend/vfend

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

Description **Vend** releases all internal data structures allocated by calling **Vstart**. This routine must be called after all vdata and vgroup operations on an HDF file are completed. Further attempts to use vdata or vgroup routines after calling **Vend** will result in an error condition.

FORTRAN

```
integer function vfend(file_id)

integer file_id
```

Vfind/vfind

int32 Vfind(int32 *file_id*, const char **vgname*)

file_id	IN:	File identifier returned by **Hopen**
vgname	IN:	Name of the vgroup

Purpose Returns the reference number of the vgroup with the specified name.

Return value Returns the reference number of the vgroup if successful and `0` otherwise.

FORTRAN

```
integer function vfind(file_id, vgname)

integer file_id
character* (*) vgname
```

Vfindattr/vffdatt

intn Vfindattr(int32 *vgroup_id*, char **attr_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_name	OUT:	Name of the target attribute

Purpose Returns the index of an attribute with the given name, specified by the value of the *attr_name* parameter, of a vgroup.

Return value Returns the index of the target attribute if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vffdatt(vgroup_id, attr_name)

integer group_id
character* (*) attr_name
```

Vfindclass/vfndcls

int32 Vfindclass(int32 *file_id*, const char **vgclass*)

file_id	IN:	File identifier returned by **Hopen**
vgclass	IN:	Class of the vgroup

Purpose Returns the reference number of the vgroup with the specified class.

Return value Returns the reference number of the vgroup if successful and `0` otherwise.

FORTRAN

```
integer function vfndcls(file_id, vgclass)

integer file_id
character* (*) vgclass
```

Vflocate/vffloc

int32 Vflocate(int32 *vgroup_id*, char **field*)

vgroup_id	IN:	Vgroup identifier
field	IN:	Target field name

Purpose Determines if the specified field name exists in the vgroup referred to by *vgroup_id*.

Return value Returns the reference number of the vdata containing the field name if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vffloc(vgroup_id, field)

integer vgroup_id
character* (*) field
```

Vgetattr/vfgnatt/vfgcatt

intn Vgetattr(int32 *vgroup_id*, intn *attr_index,* char **values*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_index	IN:	Index of the target attribute
values	OUT:	Name of the target attribute

Purpose Returns all of the values of the specified attribute of the specified vgroup.

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

Description The *attr_index* parameter is the ordinal, zero-based index of the target attribute.

FORTRAN

```
integer function vfgnatt(vgroup_id, attr_index, values)

integer vgroup_id, attr_index, values


integer function vfgcatt(vgroup_id, attr_index, values)

integer vgroup_id, attr_index
character* (*) values
```

Vgetclass/vfgcls

int32 Vgetclass(int32 *vgroup_id*, char **vgroup_class*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_class	OUT:	Buffer for storing the vgroup class name

Purpose Retrieves the class name of the vgroup specified by *vgroup_id* and returns it in the buffer pointed to by *vgroup_class*.

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

Description The maximum length of the name is defined by the macro `VGNAMELENMAX`.

FORTRAN

```
integer function vfgcls(vgroup_id, vgroup_class)

integer vgroup_id
character* (*) vgroup_class
```

Vgetid/vfgid

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 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 used to sequentially search the file for vgroups.

Return value Returns the reference number of the next vdata if successful and `FAIL` (or `-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);
Vstart(file_id);
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);
}

Vend(file_id);
Hclose(file_id);
```

FORTRAN

```
integer function vfgid(file_id, vgroup_ref)

integer file_id, vgroup_ref
```

Vgetname/vfgnam

int32 Vgetname(int32 *vgroup_id*, char **vgroup_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_name	OUT:	Buffer for the vgroup name

Purpose Returns the name of the vgroup.

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

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

```
integer function vfgnam(vgroup_id, vgroup_name)

integer vgroup_id
character* (*) vgroup_name
```

Vgetnext/vfgnxt

int32 Vgetnext(int32 *vgroup_id*, int32 *elem_ref*)

vgroup_id	IN:	Vgroup identifier
elem_ref	IN:	Reference number of the vgroup or vdata

Purpose Searches in the vgroup specified by *vgroup_id* for thevgroup or vdata following the vgroup or vdata referred to by *elem_ref*.

Return value Returns the reference number of the target vgroup or vdata if successful and `FAIL` (or `-1`) otherwise.

Description If *elem_ref* is set to `-1` the reference number of the first element in the vgroup is returned.

Vgetnext will only examine vset elements. To examine all element links in a vgroup, use **Vgettagrefs**.

FORTRAN

```
integer function vfgnxt(vgroup_id, elem_ref)

integer vgroup_id, elem_ref
```

Vgettagref/vfgttr

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/reference number pair to be retrieved from the vgroup
tag	OUT:	Buffer for the returned tag
ref	OUT:	Buffer for the returned reference number

Purpose Returns the tag/reference number pair of an HDF object at a given position within a vgroup. Do not confuse this routine with **Vgettagrefs**.

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

Description The input parameter index specifies the position within the vgroup. This routine provides a means of accessing the tag/reference number pairs in a vgroup by specifying the position within the vgroup.

Example This sample code first uses **Vntagrefs** to determine the number of tag/reference number pairs in a vgroup. It then uses **Vgettagref** to extract each tag/ref erence number pair 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, index, tag, ref)

integer vgroup_id, index
integer tag, ref
```

Vgettagrefs/vfgttrs

int32 Vgettagrefs(int32 *vgroup_id*, int32 *tag_array*[], int32 *ref_array*[], int32 *maxsize*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag_array	OUT:	Buffer for the returned tags
ref_array	OUT:	Buffer for the returned reference numbers
maxsize	IN:	Maximum number of tag/reference number pairs to store in *tag_array* and *ref_array*

Purpose Returns the tag/reference number pairs of the HDF objects belonging to a given vgroup. Do not confuse this routine with **Vgettagref**.

Return value Returns the number of tag/reference number pairs in a specified vgroup if successful and `FAIL` (or `-1`) otherwise.

Description The input parameter maxsize specifies the maximum number of tag/reference number pairs to be returned. The tag/reference number 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/reference number pairs in a vgroup. It then allocates memory, and uses **Vgettagrefs** to extract the tag/reference number pair 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 vgroup_id, maxsize
integer tag_array(*), ref_array(*)
```

Vgetversion/vfgver

int32 Vgetversion(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose **Vgetversion** returns a value of `VSET_NEW_VERSION` when encountering a vgroup of this vgroup version.

Return value Returns the vset version number if successful, and `FAIL` otherwise.

Description Version 3 is the vset version corresponding to all versions of the HDF library between 3.2 and 4.0 release 2. **Vgetversion** returns a value of `VSET_VERSION` when encountering a vgroup of this vset version. The third version is version 2. This vset version corresponds to all HDF library versions before version 3.2, and **Vgetversion** returns a value of `VSET_OLD_VERSION` when encountering a vgroup of this vset version.

FORTRAN

```
integer function vfgver(vgroup_id)

integer vgroup_id
```

Vinqtagref/vfinqtr

intn Vinqtagref(int32 *vgroup_id*, int32 *tag*, int32 *ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag	IN:	Tag of object to be checked for
ref	IN:	Reference number of object to be checked for

Purpose Tests whether a given HDF object is linked to a given vgroup.

Return value Returns `TRUE` (or `1`) if the tag and reference number are linked to the specified vgroup, and `FALSE` (or `0`) otherwise.

Description The HDF object is specified by the *tag* and *ref* parameters. This routine then checks if the tag/reference number pair can be found in the vgroup specified by *vgroup_id*.

FORTRAN

```
integer function vfinqtr(vgroup_id, tag, ref)

integer vgroup_id, tag, ref
```

Vinquire/vfinq

intn Vinquire(int32 *vgroup_id*, int32 **n_entries*, char **vgroup_name*)

vgroup_id	IN:	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 Retrieves the number of entries in a vgroup and the name of the vgroup. .

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

Description The maximum length of the vgroup's name is defined by `VGNAMELENMAX`.

FORTRAN

```
integer function vfinq(vgroup_id, n_entries, vgroup_name)

integer vgroup_id, n_entries
character* (*) vgroup_name
```

Vinsert/vfinsrt

int32 Vinsert(int32 *vgroup_id*, int32 *v_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
v_id	IN:	Identifier for the item, vdata or vgroup, to insert into a vgroup

Purpose 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` (or `-1`) otherwise.

Description Essentially, **Vinsert** allows vgroups or vdatas to be grouped together. To insert other HDF objects (i.e., 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 vgroup_id, v_id
```

Visvg/vfisvg

intn Visvg(int32 *vgroup_id*, int32 *v_ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
v_ref	IN:	Vgroup or vdata reference number to be queried

Purpose Determines whether the vgroup or vdata specified by *v_ref* is contained within the vgroup identified by *vgroup_id*.

Return value Returns `TRUE` (or `1`) if the reference number specified by the *v_ref* parameter refers to an object stored in the specified vgroup and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vfisvg(vgroup_id, v_ref)

integer vgroup_id, v_ref
```

Visvs/vfisvs

intn Visvs(int32 *vgroup_id*, int32 *vdata_ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vdata_ref	IN:	Vdata reference number to be queried

Purpose Determines whether the vdata specified by *vdata_ref* is contained within another the vgroup identified by *vgroup_id*.

Return value Returns `TRUE` (or `1`) if the vdata reference number specified by the *vdata_ref* parameter is that of an object stored in the specified vgroup and `FAIL` (or `-1`) otherwise.

Description Generally called after **Vgettagrefs** to determine which entries are vdatas.

FORTRAN

```
integer function vfisvs(vgroup_id, vdata_ref)

integer vgroup_id, vdata_ref
```

Vlone/vflone

int32 Vlone(int32 *file_id*, int32 *ref_array*[], int32 *maxsize*)

file_id	IN:	File identifier returned by **Hopen**
ref_array	OUT:	Buffer for the reference numbers of the lone vgroups
maxsize	IN:	Maximum number of vgroups to store in *ref_array*.

Purpose 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` (or `-1`) otherwise.

Description This routine is provided for applications to locate all lone vgroups (i.e., those that are not grouped with other objects) in a HDF file. Another use is to locate all vgroups at the top of the 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 65,000 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 target 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 file_id, ref_array(*), maxsize
```

Vnattrs/vfnatts

intn Vnattrs(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Returns the number of attributes assigned to the vgroup identified by *vgroup_id*.

Return value Returns the total number of attributes assigned to the specified vgroups if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vfnatts(vgroup_id)

integer group_id
```

Vnrefs/vnrefs

int32 Vnrefs(int32 *vgroup_id*, int32 *tag*)

vgroup_id	IN:	Vgroup identifier
tag	IN:	Target tag definition

Purpose Returns the number of tags of the type specified by *tag* in the vgroup referred to by *vgroup_id*.

Return value Returns the total number of tags if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vnrefs(vgroup_id, tag)

integer vgroup_id, tag
```

Vntagrefs/vntrc

int32 Vntagrefs(int32 *vgroup_id*)

vgroup_id	IN:	The vgroup identifier returned by **Vattach**

Purpose Returns a count of the number of tag and reference number pairs stored in the given vgroup.

Return value Returns the number of HDF elements linked to the vgroup and or `FAIL` (or `-1`) otherwise.

Description 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 vntrc(vgroup_id)

integer vgroup_id
```

Vsetattr/vfsnatt/vfscatt

intn Vsetattr(int32 *vgroup_id*, char **attr_name,* int32 *data_type,* int32 *count,* VOIDP *values*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_name	IN:	Name of the attribute
data_type	IN:	Data type of the attribute
count	IN:	Number of values the attribute contains
values	IN:	Buffer containing the attribute values

Purpose Attaches an attribute to a vgroup.

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

Description If the attribute already exists, the new values will replace the current ones, provided the data type and order have not been changed. If either the data type or the order have been changed, **Vsetattr** will exit on an error condition.

FORTRAN

```
integer vfsnatt(vgroup_id, attr_name, data_type, count, values)

integer vgroup_id, data_type, count
<valid numerical type> values
character* (*) attr_name


integer vfscatt(vgroup_id, attr_name, data_type, count, values)

integer vgroup_id, data_type, count
character* (*) attr_name, values
```

Vsetclass/vfscls

int32 Vsetclass(int32 *vgroup_id*, const char **vgroup_class*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_class	IN:	Name of class for vgroup

Purpose Sets the class name for a vgroup.

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

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

```
integer function vfscls(vgroup_id, vgroup_class)

integer vgroup_id
character* (*) vgroup_class
```

Vsetname/vfsnam

int32 Vsetname(int32 *vgroup_id*, const char **vgroup_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_name	IN:	Name to be assigned to the vgroup

Purpose Sets the name for a vgroup.

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

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.

The maximum length of *vgroup_name* is `VGNAMELENMAX`. If a longer name is specified it will be truncated to that length.

FORTRAN

```
integer function vfsnam(vgroup_id, vgroup_name)

integer vgroup_id
character* (*) vgroup_name
```

Vstart/vfstart

intn Vstart(int32 *file_id*)

file_id	IN:	File identifier returned by **Hopen**

Purpose Initializes the internal vdata and vgroup data structures in an HDF file.

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

Description 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

```
integer function vfstart(file_id)

integer file_id
```

VFfieldesize/vffesiz

int32 VFfieldesize(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
field_index	IN:	Number determined by assigning each field in the record a zero-based number

Purpose Retrieves the size, as stored on disk, of the specified field in the specified vdata.

Return value Returns the size of a value for the specified field if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vffesiz(vdata_id, field_index)

integer vdata_id, field_index
```

VFfieldisize/vffisiz

int32 VFfieldisize(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
field_index	IN:	Number determined by assigning each field in the record a zero-based index starting with `0`

Purpose Retrieves the size, as stored in memory, of the specified field in a given vdata.

Return value Returns the size of a value for the specified field if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vffisiz(vdata_id, field_index)

integer vdata_id, field_index
```

VFfieldname/vffname

char *VFfieldname(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
field_index	IN:	Number determined by assigning each field in the record a zero-based index

Purpose Retrieves the fieldname for the specified field in a given vdata.

Return value Returns a pointer to the fieldname if successful and `NULL` otherwise. The Fortran-77 version of this routine, `vffname`, returns `SUCCEED` (or `0`) or `FAIL` (or `-1`).

Description The Fortran-77 version of this routine, `vffname`, returns the field name in the `fname` parameter.

FORTRAN

```
integer function vffname(vdata_id, field_index, fname)

integer vdata_id, field_index
character* (*) fname
```

VFfieldorder/vffordr

int32 VFfieldorder(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
field_index	IN:	Number determined by assigning each field in the record a zero-based index

Purpose Retrieves the order of the specified field in a given vdata.

Return value Returns the order of the field if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vffordr(vdata_id, field_index)

integer vdata_id, field_index
```

VFfieldtype/vfftype

int32 VFfieldtype(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
field_index	IN:	Number determined by assigning each field in the record a zero-based index

Purpose Retrieves the data type for the specified field in a given vdata.

Return value Returns the data type if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vfftype(vdata_id, field_index)

integer vdata_id, field_index
```

VFnfields/vfnflds

int32 VFnfields(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **VSattach**

Purpose Returns the total number of fields in the specified vdata.

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

FORTRAN

```
integer function vfnflds(vdata_id)

integer vdata_id
```

VHmakegroup/vhfmkgp

int32 VHmakegroup(int32 *file_id*, int32 *tag_array*[], int32 *ref_array*[], int32 *n_objects*, char **vgroup_name*, char **vgroup_class*)

file_id	IN:	File identifier returned by **Hopen**
tag_array	IN:	Array of tags to be stored
ref_array	IN:	Array of reference numbers to be stored
n_objects	IN:	Number of data objects to be stored
vgroup_name	IN:	Name of the vgroup to be created
vgroup_class	IN:	Class of the vgroup to be created

Purpose Groups a collection of data objects within a vgroup.

Return value Reference number of newly-created vgroup if successful, `FAIL` (or `-1`) otherwise.

Description This routine creates a new vgroup named *vgroup_name* of class *vgroup_class* in the HDF file specified by *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** initializes the VH interface, it must precede calls to **VHmakegroup**. It is not necessary, however, to call **Vattach** or **Vdetach** in conjunction with **VHmakegroup**.

FORTRAN

```
integer function vhfmkgp(file_id, tag_array, ref_array,
                         n_objects, vgroup_name, vgroup_class)

integer file_id, n_objects
character* (*) vgroup_name, vgroup_class
integer tag_array(*), ref_array(*)
```

VHstoredata/vhfsd/vhfscd

int32 VHstoredata(int32 *file_id*, char **fieldname*, uint8 *buf*[], int32 *n_records*, int32 *data_type*, char **vdata_name*, char **vdata_class*)

file_id	IN:	File identifier returned by **Hopen**
fieldname	IN:	Field name for the new vdata
buf	IN:	Buffer the records to be read
n_records	IN:	Number of records to be stored
data_type	IN:	Data type for the field in each record
vdata_name	IN:	Name of the vdata to be created
vdata_class	IN	Class of the vdata to be created

Purpose Creating vdatas containing records limited to one field with one component per field.

Return value Reference number of newly-created vdata if successful, `FAIL` (or `-1`) otherwise

Description This routine 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 data type of the field is specified by *data_type* and *n_records* of data from buf are stored with a field name specified by *fieldname*.

Because **Vstart** initializes the VH interface, it must precede **VHstoredata**. It is not necessary, however, to call **VSattach** or **VSdetach** in conjunction with **VHstoredata**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**vhfsd**) and the other for buffered character data (**vhfsdc**).

FORTRAN

```
integer function vhfsd(file_id, fieldname, buf, n_records,
                            data_type, vdata_name, vdata_class)

integer file_id, n_records, data_type
character* (*) access, vdata_name, vdata_class
<valid numeric data type> buf(*)

integer function vhfscd(file_id, fieldname, buf, n_records,
                            data_type, vdata_name, vdata_class)

integer file_id, n_records, data_type
character* (*) access, vdata_name, vdata_class
```

```
character* (*) buf
```

VHstoredatam/vhfsdm/vhfscdm

int32 VHstoredatam(int32 *file_id*, char **fieldname*, uint8 *buf*[], int32 *n_records*, int32 *data_type*, char **vdata_name*, char **vdata_class*, int32 *order*)

file_id	IN:	File identifier returned by **Hopen**
fieldname	IN:	Field name for the new vdata
buf	IN:	Buffer the records are to be read from
n_records	IN:	Number of records to be stored
data_type	IN:	Data type of data elements
vdata_name	IN:	Name of the vdata to be created
vdata_class	IN:	Class of the vdata to be created
order	IN:	Number of components per field

Purpose Creates vdatas containing records with one field containing one or more components.

Return value Reference number of the newly-created vdata if successful, `FAIL` (or `-1`) otherwise

Description This routine 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 data type of the vdata and all its components is specified by *data_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** initializes the VH interface, it must precede **VHstoredatam**. It is not necessary, however, to call **VSattach** or **VSdetach** in conjunction with **VHstoredatam**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**vhfsdm**) and the other for buffered character data (**vhfscdm**).

FORTRAN

```
integer function vhfsdm(file_id, fieldname, buf, n_records,
                        data_type, vdata_name, vdata_class
                         order)

integer file_id, n_records, data_type, order
character* (*) access, vdata_name, vdata_class
<valid numeric data type> buf(*)

integer function vhfscdm(file_id, fieldname, buf, n_records,
```

```
                            data_type, vdata_name, vdata_class
                            order)

integer file_id, n_records, data_type, order
character* (*) access, vdata_name, vdata_class
character* (*) buf
```

VQueryref/vqref

int32 VQueryref(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier

Purpose Returns the reference number of the specified vgroup.

Return value Returns the reference number if successful and `0` otherwise.

FORTRAN
```
integer function vqref(vgroup_id)

integer vgroup_id
```

VQuerytag/vqtag

int32 VQuerytag(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier

Purpose Returns the tag of the specified vgroup.

Return value Returns the tag if successful and `0` otherwise.

FORTRAN

```
integer function vqtag(vgroup_id)

integer vgroup_id
```

VSappendable/vsapp

int32 VSappendable(int32 *vdata_id*, int32 *block_size*)

vdata_id	IN:	Vdata identifier
block_size	IN:	Standard block size of appended data

Purpose Specifies that the vdata referred to by *vdata_id* can be appended to.

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

Description NOTE: This routine is obsolete.

FORTRAN

```
integer function vsapp(vdata_id, block_size)

integer vdata_id, block_size
```

VSattach/vsfatch

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:	Access mode

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` (or `-1`) otherwise.

Description **VSattach** returns an identifier to the vdata, through which all further operations on that vdata are carried out. See also **VSdetach**.

Valid values for *access* are: "`r`" for read access and "`w`" for write access.

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 an vdata with a *vdata_ref* of `1` with "`r`" access.

If *access* is "`w`", then *vdata_ref* must be the 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 file_id, vdata_ref
character*1 access
```

VSattrinfo/vsfainf

intn VSattrinfo(int32 *vdata_id*, int32 *field_index,* int32 *attr_index,* char **attr_name,* int32 **data_type,* int32 **count,* int32 **size*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
field_index	IN:	Index of the target field
attr_index	IN:	Index of the target attribute
attr_name	OUT:	Name of the target attribute
data_type	OUT:	Data type of the target attribute
count	OUT:	Attribute count
size	OUT:	Size of the target attribute

Purpose Returns the name, data type, number of values, and the size of the values of the specified attribute of the specified vdata field or vdata.

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

Description The values of the *attr_name*, *data_type*, *count* and *size* parameters can be set to `NULL`, if the information returned by these parameters are not needed.

The *field_index* in **VSattrinfo** is the same as with the *field_index* parameter in **VSsetattr**. It can be set to either a integer field index, or `_HDF_ENTIRE_VDATA` to specify the vdata referred to by *vdata_id*.

FORTRAN

```
integer function vsfainf(vdata_id, field_index, attr_index,
    attr_name, data_type, count, size)

integer vdata_id, field_index, attr_index
character* (*) attr_name
integer data_type, count, size
```

VSdetach/vsfdtch

int32 VSdetach(int32 *vdata_id*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose Detaches from the current vdata thereby terminating further access to that vdata.

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

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

```
integer function vsfdtch(vdata_id)

integer vdata_id
```

VSelts/vsfelts

int32 VSelts(int32 *vdata_id*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose Determines the number of records in the specified vdata.

Return value The number of elements in the vdata if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vsfelts(vdata_id)

integer vdata_id
```

VSfdefine/vsffdef

intn VSfdefine(int32 *vdata_id*, char **fieldname*, int32 *data_type*, int32 *order*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
fieldname	IN:	Name of field to be defined
data_type	IN:	Data type of the field values
order	IN:	Order of the new field

Purpose Defines a new field for in a vdata.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-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 (*data_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, *data_type* may be one of the following:

```
8-bit character DFNT_CHAR8          4
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, as velocity has three components.

Example

The following two calls to **VSfdefine** define two fields in the newly-created vdata. The first field, named VEL, is a three-component float-valued field, while the second field, named PRESSURE, is a single-component float32-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, data_type,
        order)

integer vdata_id, data_type, order
character* (*) fieldname
```

VSfexist/vsfex

intn VSfexist(int32 *vdata_id*, char **fields*)

vdata_id	IN:	Vdata 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` (or `-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 vdata_id
character* (*) fields
```

VSfind/vsffnd

int32 VSfind(int32 *file_id*, char **vdata_name*)

file_id	IN:	File identifier returned by **Hopen**
vdata_name	IN:	Name of the vdata

Purpose Queries the specified HDF file for a vdata with a given name.

Return value Returns the vdata reference number if successful and `0` otherwise.

Description If there is more than one vdata 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 file_id
character* (*) vdata_name
```

VSfindex/vsffidx

intn VSfindex(int32 *vdata_id*, char **fieldname*, int32 **findex*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
fieldname	IN:	Name of the field
findex	OUT:	Returned index of the specified field

Purpose Retrieves the index of the specified field in the specified vdata.

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

Description This routine searches for field names only. It doesn't search the vdata name - use **VSinquire** or **VSgetname** to do this.

FORTRAN

```
integer function vsffidx(vdata_id, fieldname, findex)

integer vdata_id, findex
character* (*) fieldname
```

VSfindattr/vsffdat

intn VSfindattr(int32 *vdata_id*, int32 *field_index,* char **attr_name*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
field_index	IN:	Index of the target field
attr_name	OUT:	Name of the target attribute

Purpose Returns the index of an attribute with the specified name.

Return value Returns the index of the target attribute if successful and `FAIL` (or `-1`) otherwise.

Description The attribute must to attached to the vdata identified by the the specified vdata identifier. If *field_index* is set to `_HDF_ENTIRE_VDATA`, the index of the attribute attached to the vdata referred to by the value of *vdata_id* is returned. When using the Fortran-77 version of this routine, specify a *field_index* value of `-1` to return the vdata attribute indices.

As with **VSsetattr**, if *field_index* is set to a zero-based integer value, the value will be used as the index of the target vdata field, and the index of the attribute with the name specifiedby the *attr_name* parameter will be returned.

FORTRAN

```
integer function vsffdat(vdata_id, field_index, attr_name)

integer vdata_id, field_index
character* (*) attr_name
```

VSfindclass

int32 VSfindclass(int32 *file_id*, const char **vdata_class*)

file_id	IN:	File identifier returned by **Hopen**
vdata_class	IN:	Class of the vdata

Purpose Returns the reference number of the first vdata coresponding to the specified vdata class.

Return value Returns the reference number of the vdata if successful and `0` or an error code otherwise.

VSfnattrs/vsffnas

int32 VSfnattrs (int32 *vdata_id*, int32 *field_attrs*)

vdata_id	IN:	Vdata identifier returned by **Vattach**
field_attrs	OUT:	Index of the target field

Purpose Returns the number of attributes attached to the specified vdata field *or* the number of attributes attached to the vdata identified by *vdata_id*.

Return value Returns the number of attributes assigned to this vdata or its fields when successful, and `FAIL` (or `-1`) otherwise.

Description This routine is different from the **VSnattrs** routine, which returns the number of attributes of the specified vdata *and* the fields contained in it.

If *field_index* is set to `_HDF_ENTIRE_VDATA`, the number of attributes attached to the vdata referred to by the value of *vdata_id* is returned. When using the Fortran-77 version of this routine, specify a *field_index* value of `-1` to return the number of vdata attributes.

As with **VSsetattr**, if *field_index* is set to a zero-based integer value, it will be used as the index of the target vdata field, and the attributes attached to that field will be returned.

FORTRAN

```
integer function vsffnas(vdata_id, field_index)

integer vdata_id, field_index
```

VSfpack/vsfcpak/vsfnpak

intn VSfpack(int32 *vdata_id*, intn *action*, char **fields_in_buf*, VOIDP *buf*, intn *buf_size*, intn *n_records*, char **fields*, VOIDP *bufptrs*[])

vdata_id	IN:	Vata identifier returned from **VSattach**
action	IN:	Action to be performed
fields_in_buf	IN:	Fields in *buf* to write to or read from the vdata
buf	IN/OUT:	Buffer for the vdata values
buf_size	IN:	Buffer size in bytes
n_records	IN:	Number of records to pack or unpack
fields	IN:	Names of the fields to be packed or unpacked. It may be a subset of the *fields_in_buf*. `NULL` stands for all fields in *buf*
bufptrs	IN/OUT:	Array of pointers to the field buffers

Purpose Packs field data into a buffer or unpacks buffered field data into vdata fields.

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

Description The calling program should supply the number of field buffers corresponding to the number of fields to be packed or unpacked and is responsible to allocate sufficient space for each buffer. This should be at least *n_records**(total size of all fields specified in *fields_in_buf*) in length.

Valid values of *action* are: `_HDF_VSPACK` (or `0`) which pack field values into *buf*, or `_HDF_VSUNPACK` (or `1`) which unpacks vdata values into field buffers.

There are two Fortran-77 versions of **VSfpack**: **vsfnpak** and **vsfcpak**. The **vsfnpak** routine packs or unpacks a numeric field and **vsfcpak** a character type field. These routines pack one field at a time.

When **VSfpack** is called to pack field values into *buf*, *fields_in_buf* should contain all fields of the vdata. `NULL` can be used in C programs for *fields_in_buf*, a blank character ' ' in Fortran-77 programs. When **VSfpack** is called to unpack field values, *fields_in_buf* may be a subset of the vdata fields. `NULL` can be passed into **VSfpack** and the space character in **vsfnpak** and **vsfcpak** as the *fields_in_buf* parameter to specify all fields in the vdata.

If the *fields_in_buf* parameter is set to `NULL`, all fields in the vdata will be packed.

FORTRAN

```
integer function vsfnpak(vdata_id, action, fields_in_buf,
                  buf, buf_size, n_records, fields, bufptrs)

integer vdata_id, action, buf, buf_size, n_records, bufptrs
character* (*) fields_in_buf, fields


integer function vsfcpak(vdata_id, action, fields_in_buf,
                  buf, buf_size, n_records, fields, bufptrs)

integer vdata_id, action, buf, buf_size, n_records
character* (*) fields_in_buf, fields, bufptrs
```

VSgetattr/vsfgnat/vsfgcat

intn VSgetattr(int32 *vdata_id*, int32 *field_index,* int32 *attr_index,* VOIDP *values*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
field_index	IN:	Index of the target field
attr_index	IN:	Index of the target attribute
values	OUT:	Buffer containing the attribute values

Purpose Returns the values of the specified attribute of the specified vdata field or vdata.

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

Description The *attr_index* parameter is the ordinal, zero-based (as with *field_index* in **VSsetattr** and this routine) index of the target attribute.

If *field_index* is set to `_HDF_ENTIRE_VDATA`, the value of the attribute attached to the vdata referred to by the value of *vdata_id* is returned. If *field_index* is set to a zero-based integer value, the value will be used as the index of the target vdata field, and the attribute located at the ordinal position specified by *attr_index* will be returned.

FORTRAN

```
integer function vsfgnat(vdata_id, field_index, attr_index,
        values)

integer vdata_id, field_index, attr_index
<valid numeric data type> values(*)


integer function vsfgcat(vdata_id, field_index, attr_index,
        values)

integer vdata_id, n_records, interlace
character* (*) values
```

VSgetclass/vsfgcls

int32 VSgetclass(int32 *vdata_id*, char **vdata_class*)

vdata_id	IN:	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 Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) otherwise.

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

```
integer function vsfgcls(vdata_id, vdata_class)

integer vdata_id
character* (*) vdata_class
```

VSgetfields/vsfgfld

int32 VSgetfields(int32 *vdata_id*, char **fields*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
fields	OUT:	Buffer for the 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` (or `-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 vdata_id
character* (*) fields
```

VSgetid/vsfgid

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` (or `-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 routine is generally used to sequentially search the file for vdatas.

To initiate a search, this routine must be called with *vdata_ref* equal to `-1`. Doing so returns the reference number of the first vdata in the file. Searching past the last vdata in a file will result in an error condition.

FORTRAN

```
integer function vsfgid(file_id, vdata_ref)

integer file_id, vdata_ref
```

VSgetinterlace/vsfgint

int32 VSgetinterlace(int32 *vdata_id*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose Returns the interlace type of the vdata specified by *vdata_id*.

Return value Returns `FULL_INTERLACE` or `NO_INTERLACE` if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vsfgint(vdata_id)

integer vdata_id
```

VSgetname/vsfgnam

int32 VSgetname(int32 *vdata_id*, char **vdata_name*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_name	OUT:	Buffer for the vdata name

Purpose Retrieves the name of the specified vdata.

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

Description **VSgetname** returns in *vdata_name* the name, if any, of the specified vdata.The maximum length of the class name is defined by `VSNAMELENMAX`. If there is no name for the vdata, a null string is placed in the *vdata_name* parameter.

FORTRAN

```
integer function vsfgnam(vdata_id, vdata_name)

integer vdata_id
character* (*) vdata_name
```

VSgetversion/vsgver

int32 VSgetversion(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier

Purpose Retrieves the version number of a vdata.

Return value Returns the version number if successful and either `FAIL` (or `-1`) or an error code otherwise.

FORTRAN

```
integer vsgver(vdata_id)

integer vdata_id
```

VSinquire/vsfinq

intn VSinquire(int32 *vdata_id*, int32 **n_records*, int32 **interlace*, char **fields*, int32 **vdata_size*, char **vdata_name*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
n_records	OUT:	Number of records in the vdata
interlace	OUT:	Interlace type
fields	OUT:	Comma-separated string listing all fields in the vdata (e.g. "PX, PY,PZ")
vdata_size	OUT:	Size, in bytes, of a record in the vdata for the local machine
vdata_name	OUT:	Name, if any, of the vdata

Purpose General vdata inquiry routine.

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

Description **VSinquire** retrieves the information for each of the variables listed above. If any of the parameters are NULL, that data is not retrieved. Refer to the Reference Manual pages on **VSgetfields**, **VSgetinterlace**, **VSsizeof** and **VSgetname**.

Valid values for *interlace* are: FULL_INTERLACE or NO_INTERLACE.

FORTRAN

```
integer function vsfinq(vdata_id, n_records, interlace,
        fields, vdata_size, vdata_name)

integer vdata_id, n_records, interlace, vdata_size
character* (*) fields, vdata_name
```

VSisattr/vsfisat

intn VSisattr(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned from **VSattach**

Purpose Determines whether the specified vdata contains attribute data.

Return value Returns `TRUE` if the vdata is an attribute, and `FALSE` otherwise.

Description As attributes are stored by the HDF library as vdatas, a means of testing whether or not a particular vdata is attribute data is needed, and provided by the **VSisattr** routine.

FORTRAN

```
integer function vsfisat(vdata_id)

integer vdata_id
```

VSlone/vsflone

int32 VSlone(int32 *file_id*, int32 *ref_array*[], int32 *maxsize*)

file_id	IN:	File identifier returned by **Hopen**
ref_array	OUT:	Buffer for the returned list of vdata reference numbers
maxsize	IN:	Maximum number of vdata reference numbers to be stored in the array

Purpose Locates all lone vdatas (i.e. those that are not grouped with other objects) in a HDF file.

Return value Returns the total number of lone vdatas in the file if successful and `FAIL` (or `-1`) otherwise.

Description The integer array *ref_array* must be provided, and the maximum size of the array must be specified in the parameter *maxsize*. At most *maxsize* reference numbers will be returned in *ref_array*.

An array size of 65,000 integers for *ref_array* is generally 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**. Refer to the Reference Manual page on **Vlone**.

Example This 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 file_id, maxsize
integer ref_array(*)
```

VSnattrs/vsfnats

intn VSnattrs(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **Vattach**

Purpose Returns the number of attributes of the specified vdata *and* the vdata fields contained in it.

Return value Returns the number of attributes if successful and `FAIL` (or `-1`) otherwise.

Description This routine is different from the **VSfnattrs** routine, which returns the number of attributes of a field contained in the specified vdata *or* the specified vdata.

FORTRAN

```
integer function vsfnats(vdata_id)

integer vdata_id
```

VSread/vsfrd/vsfrdc

int32 VSread(int32 *vdata_id*, uint8 **databuf*, int32 *n_records*, int32 *interlace*)

vdata_id	IN:	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 mode of the data stored in the buffer

Purpose Retrieves data from the specified vdata.

Return value Returns the total number of records read if successful and `FAIL` (or `-1`) otherwise.

Description The number of records to be read must be specified in *n_records* and the interlace mode of the data in the buffer must be specified in *interlace*. Use `FULL_INTERLACE` (recommended) or `NO_INTERLACE`.

The vdata is first attached with read access, then **VSsetfields** is called to specify which field(s) are to be read from that vdata. Each subsequent call to **VSread** will return successive records from that vdata.

If the call is successful, the data returned in *databuf* is formatted according to the specified interlace mode, 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**. Refer to the Reference Manual pages on **VSseek** and **VSsetfields**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**vsfrd**) and the other for buffered character data (**vsfrdc**).

FORTRAN

```
integer function vsfrd(vdata_id, databuf, n_records, interlace)

integer vdata_id, n_records, interlace
<valid numeric data type> databuf(*)


integer function vsfrdc(vdata_id, databuf, n_records,
                        interlace)

integer vdata_id, n_records, interlace
character*(*) databuf
```

VSseek/vsfseek

int32 VSseek(int32 *vdata_id*, int32 *record*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
record	IN:	Target record number

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` (or `-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 call to **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* value of `0`. Each seek is constrained to a record boundary within the vdata.

FORTRAN

```
integer function vsfseek(vdata_id, record)

integer vdata_id, record
```

VSsetattr/vsfsnat/vsfscat

intn VSsetattr(int32 *vdata_id*, int32 *findex,* char **attr_name,* int32 *data_type,* int32 *count,* VOIDP **values*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
findex	IN:	Number determined by assigning each field in a record a number starting with 0
attr_name	IN:	Name of the attribute
data_type	IN:	Data type of the attribute
count	IN:	Number of attribute values
values	OUT:	Buffer containing the returned values of the attribute

Purpose Sets an attribute of a vdata or the field of a vdata.

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

Description If the field already has an attribute with the same name, the current values will be replaced with the new values if the new data type and order are the same as the current ones. Any changes in the field data type or order will result in an error condition.

An *findex* value of `_HDF_VDATA` (0xffffffff) represents the entire vdata.

FORTRAN

```
integer function vsfsnat(vdata_id, findex, attr_name,
    data_type, count, values)

integer vdata_id, findex, data_type, count, values
character* (*) attr_name


integer function vsfscat(vdata_id, findex, attr_name,
    data_type, count, values)

integer vdata_id, findex, data_type, count
character* (*) attr_name, values
```

VSsetclass/vsfscls

int32 VSsetclass(int32 *vdata_id*, char **vdata_class*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_class	IN:	Name of the vdata class

Purpose Sets the class name of the vdata.

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

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. Refer to the Reference Manual page on **VSgetclass**.

FORTRAN

```
integer function vsfscls(vdata_id, vdata_class)

integer vdata_id
character* (*) vdata_class
```

VSsetexternalfile/vsfsextf

intn VSsetexternalfile(int32 *vdata_id*, char **filename*, int32 *offset*)

vdata_id	IN:	Vdata identifier returned from **VSattach**
filename	IN:	Name of the external file
offset	IN:	Offset, in bytes, of the location in the external file the new data is to be written

Purpose Stores vdata information in an external file.

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

Description Only the data will be stored externally. Attributes and all metadata will remain in the primary HDF file.

IMPORTANT: The calling program must ensure that the external files are relocated along with the primary file.

Read the Reference Manual page on **SDsetexternalfile** for more information on using the external file feature.

FORTRAN

```
integer function vsfsextf(vdata_id, filename, offset)

integer vdata_id, offset
character* (*) filename
```

VSsetfields/vsfsfld

intn VSsetfields(int32 *vdata_id*, char **fields*)

vdata_id	IN:	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` (or `0`) if successful and `FAIL` (or `-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 `MAX_FIELD_SIZE` 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. If an empty vdata is attached for a read, **VSsetfields** will return `FAIL`.

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 vdata_id
character* (*) fields
```

VSsetinterlace/vsfsint

intn VSsetinterlace(int32 *vdata_id*, int32 *interlace*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
interlace	IN:	Interlace mode for storing data

Purpose Sets the interlace mode of the specified vdata.

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

Description **VSsetinterlace** determines the interlace mode for the field 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 mode 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 vdata_id, interlace
```

VSsetname/vsfsnam

int32 VSsetname(int32 *vdata_id*, char **vdata_name*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_name	IN:	Name for the vdata

Purpose Assigns a name to a vdata.

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

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. Refer to the Reference Manual page on **VSgetname**.

FORTRAN

```
integer function vsfsnam(vdata_id, vdata_name)

integer vdata_id
character* (*) vdata_name
```

VSsizeof/vsfsiz

int32 VSsizeof(int32 *vdata_id*, char **fields*)

vdata_id	IN:	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` (or `-1`) otherwise. If one than one field is specified, **VSsizeof** will return the sum of the sizes of all of the fields.

Description Specify a single field or several fields (comma-separated) in the parameter fields. The field or fields 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 or fields 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 vdata_id
character* (*) fields
```

VSwrite/vsfwrt/vsfwrtc

int32 VSwrite(int32 *vdata_id*, unsigned char **databuf*, int32 *n_records*, int32 *interlace*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
databuf	IN:	Buffer of records to be stored in the vdata
n_records	IN:	Number of records to be stored
interlace	IN:	Interlace mode of the buffer in memory

Purpose Writes data to the specified vdata.

Return value Returns the total number of records written if successful and `FAIL` (or `-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 interlace mode of the vdata fields to bewritten to. 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 the *databuf* parameter 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. This can be done either by the calling program or by **VSfpack**. The latter is recommended.

The *interlace* parameter 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. Refer to the Reference Manual page on **VSsetfields**, **VSseek**, **VHstoredata**.

Note that there are two Fortran-77 versions of this routine; one for buffered numeric data (**vsfwrt**) and the other for buffered character data (**vsfwrtc**).

FORTRAN

```
integer function vsfwrt(vdata_id, databuf, n_records,
        interlace)

integer vdata_id, n_records, interlace
<valid numeric data type> databuf(*)


integer function vsfwrtc(vdata_id, databuf, n_records,
        interlace)

integer vdata_id, n_records, interlace
character* (*) databuf
```

VSQuerycount/vsqfnelt

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

FORTRAN

```
integer function vsqfnelt(vdata_id, n_records)

integer vdata_id, n_records
```

VSQueryfields/vsqfflds

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

FORTRAN

```
integer function vsqfflds(vdata_id, fields)

integer vdata_id
character*(*) fields
```

VSQueryinterlace/vsqfintr

intn VSQueryinterlace(int32 *vdata_id*, int32 **interlace*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
interlace	OUT:	Interlace mode

Purpose Returns the type of interlace in the vdata in interlace.

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

Valid values for *interlace* are: `FULL_INTERLACE` and `NO_INTERLACE`.

FORTRAN

```
integer function vsqfintr(vdata_id, interlace)

integer vdata_id, interlace
```

VSQueryname/vsqfname

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 specified vdata.

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

FORTRAN

```
integer function vsqfname(vdata_id, vdata_name)

integer vdata_id
character*(*) vdata_name
```

VSQueryref/vsqref

int32 VSQueryref(int32 *vdata_id*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose Retrieves the reference number of the specified vdata.

Return value Returns the reference number of the vdata if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vsqref(vdata_id)

integer vdata_id
```

VSQuerytag/vsqtag

int32 VSQuerytag(int32 *vdata_id*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**

Purpose Retrieves the tag of the specified vdata.

Return value Returns the tag of the vdata if successful and `FAIL` (or `-1`) otherwise.

FORTRAN

```
integer function vsqtag(vdata_id)

integer vdata_id
```

VSQueryvsize/vsqfsiz

intn VSQueryvsize(int32 *vdata_id*, int32 **vdata_size*)

vdata_id	IN:	Vdata access identifier returned from **VSattach**
vdata_size	OUT:	Native size, in bytes, of a record in the vdata

Purpose Retrieves the size (native to the local machine), in bytes, of a record in the vdata in *vdata_size*.

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

FORTRAN

```
integer function vsqfvsiz(vdata_id, vdata_size)

integer vdata_id, vdata_size
```

Section 3

HDF Definition List

3.1 Definition List Overview

This section of the Reference Manual contains a listing of all definitions used with HDF routines. The definitions are categorized by their name prefix (the portion of the name before the underscore) into tables. The tables themselves are alphebetized by name.

This section is primarily intended to be of use to Fortran programmers whose compilers do not support include files, and need to know the values of the definitions so that they can be explicitly defined in their programs.

TABLE 3A ***_INTERLACE - Interlacing Mode Codes**

Definition Name	Definition Value
FULL_INTERLACE	0
NO_INTERLACE	1

TABLE 3B ***_WILDCARD - Wildcard Code**

Definition Name	Definition Value
DFREF_WILDCARD	0

TABLE 3C **AN_* - Multifile Annotation Codes**

Definition Name	Definition Value
AN_DATA_LABEL	0
AN_DATA_DESC	1
AN_FILE_LABEL	2
AN_FILE_DESC	3

TABLE 3D

COMP_* - Compression Codes

Definition Name	Definition Value
COMP_NONE	0
COMP_RLE	11
COMP_IMCOMP	12
COMP_JPEG	2

TABLE 3E

COMP_CODE_* - Compression Codes

Definition Name	Definition Value
COMP_CODE_NONE	0
COMP_CODE_RLE	1
COMP_CODE_NBIT	2
COMP_CODE_SKPHUFF	3
COMP_CODE_DEFLATE	4
COMP_CODE_INVALID	5

TABLE 3F

DF_* - Maximum Length Codes

Definition Name	Definition Value
DF_MAXFNLEN	256

TABLE 3G

DFACC_* - File Access Codes

Definition Name	Definition Value
DFACC_READ	1
DFACC_WRITE	2
DFACC_CREATE	4
DFACC_ALL	7
DFACC_RDONLY	1
DFACC_RDWR	3
DFACC_CLOBBER	4
DFACC_DEFAULT	0
DFACC_SERIAL	1
DFACC_PARALLEL	9

TABLE 3H

DFE_* - Error Codes

Definition Name	Definition Value
DFE_NOERROR	0
DFE_NONE	0
DFE_FNF	1
DFE_DENIED	2
DFE_ALROPEN	3
DFE_TOOMANY	4
DFE_BADNAME	5
DFE_BADACC	6
DFE_BADOPEN	7
DFE_NOTOPEN	8
DFE_CANTCLOSE	9
DFE_READERROR	10
DFE_WRITEERROR	11
DFE_SEEKERROR	12
DFE_RDONLY	13
DFE_BADSEEK	14
DFE_PUTELEM	15
DFE_GETELEM	16
DFE_CANTLINK	17
DFE_CANTSYNC	18
DFE_BADGROUP	19
DFE_GROUPSETUP	20
DFE_PUTGROUP	21
DFE_GROUPWRITE	22
DFE_DFNULL	23
DFE_ILLTYPE	24
DFE_BADDDLIST	25
DFE_NOTDFFILE	26
DFE_SEEDTWICE	27
DFE_NOSUCHTAG	28
DFE_NOFREEDD	29
DFE_BADTAG	30
DFE_BADREF	31
DFE_NOMATCH	32

DFE_NOTINSET	33
DFE_BADOFFSET	34
DFE_CORRUPT	35
DFE_NOREF	36
DFE_DUPDD	37
DFE_CANTMOD	38
DFE_DIFFFILES	39
DFE_BADAID	40
DFE_OPENAID	41
DFE_CANTFLUSH	42
DFE_CANTUPDATE	43
DFE_CANTHASH	44
DFE_CANTDELDD	45
DFE_CANTDELHASH	46
DFE_CANTACCESS	47
DFE_CANTENDACCESS	48
DFE_TABLEFULL	49
DFE_NOTINTABLE	50
DFE_UNSUPPORTED	51
DFE_NOSPACE	52
DFE_BADCALL	53
DFE_BADPTR	54
DFE_BADLEN	55
DFE_NOTENOUGH	56
DFE_NOVALS	57
DFE_ARGS	58
DFE_INTERNAL	59
DFE_NORESET	60
DFE_GENAPP	61
DFE_UNINIT	62
DFE_CANTINIT	63
DFE_CANTSHUTDOWN	64
DFE_BADDIM	65
DFE_BADFP	66
DFE_BADDATATYPE	67
DFE_BADMCTYPE	68
DFE_BADNUMTYPE	69
DFE_BADORDER	70

DFE_RANGE	71
DFE_BADCONV	72
DFE_BADTYPE	73
DFE_BADSCHEME	74
DFE_BADMODEL	75
DFE_BADCODER	76
DFE_MODEL	77
DFE_CODER	78
DFE_CINIT	79
DFE_CDECODE	80
DFE_CENCODE	81
DFE_CTERM	82
DFE_CSEEK	83
DFE_MINIT	84
DFE_COMPINFO	85
DFE_CANTCOMP	86
DFE_CANTDECOMP	87
DFE_NODIM	88
DFE_BADRIG	89
DFE_RINOTFOUND	90
DFE_BADATTR	91
DFE_BADTABLE	92
DFE_BADSDG	93
DFE_BADNDG	94
DFE_VGSIZE	95
DFE_VTAB	96
DFE_CANTADDELEM	97
DFE_BADVGNAME	98
DFE_BADVGCLASS	99
DFE_BADFIELDS	100
DFE_NOVS	101
DFE_SYMSIZE	102
DFE_BADATTACH	102
DFE_BADVSNAME	103
DFE_BADVSCLASS	104
DFE_VSWRITE	105
DFE_VSREAD	106
DFE_BADVH	107

DFE_VSCANTCREATE	108
DFE_VGCANTCREATE	109
DFE_CANTATTACH	110
DFE_CANTDETACH	111
DFE_BITREAD	112
DFE_BITWRITE	113
DFE_BITSEEK	114
DFE_TBBTINS	115
DFE_BVNEW	116
DFE_BVSET	117
DFE_BVGET	118
DFE_BVFIND	119

TABLE 3I

DFNT_* - Machine Word Representation and Data Type Codes

Definition Name	Definition Value
DFNT_HDF	0
DFNT_NATIVE	4096
DFNT_CUSTOM	8192
DFNT_LITEND	16384
DFNT_NONE	0
DFNT_QUERY	0
DFNT_VERSION	1
DFNT_FLOAT32	5
DFNT_FLOAT	5
DFNT_FLOAT64	6
DFNT_DOUBLE	6
DFNT_FLOAT128	7
DFNT_INT8	20
DFNT_UINT8	21
DFNT_INT16	22
DFNT_UINT16	23
DFNT_INT32	24
DFNT_UINT32	25
DFNT_INT64	26
DFNT_UINT64	27
DFNT_INT128	28

DFNT_UINT128	29
DFNT_UCHAR8	3
DFNT_UCHAR	3
DFNT_CHAR8	4
DFNT_CHAR	4
DFNT_CHAR16	42
DFNT_UCHAR16	43
DFNT_NFLOAT32	4101
DFNT_NFLOAT	4101
DFNT_NFLOAT64	4102
DFNT_NDOUBLE	4102
DFNT_NFLOAT128	4103
DFNT_NINT8	4116
DFNT_NUINT8	4117
DFNT_NINT16	4118
DFNT_NUINT16	4119
DFNT_NINT32	4120
DFNT_NUINT32	4121
DFNT_NINT64	4122
DFNT_NUINT64	4123
DFNT_NINT128	4124
DFNT_NUINT128	4125
DFNT_NUCHAR8	4099
DFNT_NUCHAR	4099
DFNT_NCHAR8	4100
DFNT_NCHAR	4100
DFNT_NCHAR16	4138
DFNT_NUCHAR16	4139
DFNT_LFLOAT32	16389
DFNT_LFLOAT	16389
DFNT_LFLOAT64	16390
DFNT_LDOUBLE	16390
DFNT_LFLOAT128	16391
DFNT_LINT8	16404
DFNT_LUINT8	16405
DFNT_LINT16	16406
DFNT_LUINT16	16407
DFNT_LINT32	16408

DFNT_LUINT32	16409
DFNT_LINT64	16410
DFNT_LUINT64	16411
DFNT_LINT128	16412
DFNT_LUINT128	16413
DFNT_LUCHAR8	16387
DFNT_LUCHAR	16387
DFNT_LCHAR8	16388
DFNT_LCHAR	16388
DFNT_LCHAR16	16426
DFNT_LUCHAR16	16427

TABLE 3J

DFNTF_* - Floating-point Format Codes

Definition Name	Definition Value
DFNTF_NONE	0
DFNTF_HDFDEFAULT	1
DFNTF_IEEE	1
DFNTF_VAX	2
DFNTF_CRAY	3
DFNTF_PC	4
DFNTF_CONVEX	5
DFNTF_VP	6

TABLE 3K

DFO_* - Object (?) Codes

Definition Name	Definition Value
DFO_FORTRAN	1
DFO_C	2

TABLE 3L

DFTAG_* - Object Tag Codes

Definition Name	Definition Value
DFTAG_WILDCARD	0
DFTAG_NULL	1
DFTAG_LINKED	20

DFTAG_VERSION	30
DFTAG_COMPRESSED	40
DFTAG_VLINKED	50
DFTAG_VLINKED_DATA	51
DFTAG_CHUNKED	60
DFTAG_CHUNK	61
DFTAG_FID	100
DFTAG_FD	101
DFTAG_TID	102
DFTAG_TD	103
DFTAG_DIL	104
DFTAG_DIA	105
DFTAG_NT	106
DFTAG_MT	107
DFTAG_ID8	200
DFTAG_IP8	201
DFTAG_RI8	202
DFTAG_CI8	203
DFTAG_II8	204
DFTAG_ID	300
DFTAG_LUT	301
DFTAG_RI	302
DFTAG_CI	303
DFTAG_RIG	306
DFTAG_LD	307
DFTAG_MD	308
DFTAG_MA	309
DFTAG_CCN	310
DFTAG_CFM	311
DFTAG_AR	312
DFTAG_DRAW	400
DFTAG_RUN	401
DFTAG_XYP	500
DFTAG_MTO	501
DFTAG_T14	602
DFTAG_T105	603
DFTAG_SDG	700
DFTAG_SDD	701

DFTAG_SD	702
DFTAG_SDS	703
DFTAG_SDL	704
DFTAG_SDU	705
DFTAG_SDF	706
DFTAG_SDM	707
DFTAG_SDC	708
DFTAG_SDT	709
DFTAG_SDLNK	710
DFTAG_NDG	720
DFTAG_CAL	731
DFTAG_FV	732
DFTAG_BREQ	799
DFTAG_EREQ	780
DFTAG_SDRAG	781
DFTAG_VG	1965
DFTAG_VH	1962
DFTAG_VS	1963
DFTAG_RLE	11
DFTAG_IMC	12
DFTAG_IMCOMP	12
DFTAG_JPEG	13
DFTAG_GREYJPEG	14
DFTAG_JPEG5	15
DFTAG_GREYJPEG5	16

TABLE 3M

HDF_* - Vdata Interface, Linked-block Element and Vset Packing Mode Codes

Definition Name	Definition Value
HDF_VDATA	-1
HDF_VSPACK	0
HDF_VSUNPACK	1
HDF_ENTIRE_VDATA	-1
HDF_APPENDABLE_BLOCK_LEN	4096
HDF_APPENDABLE_BLOCK_NUM	16
HDF_VSPACK	0
HDF_VSUNPACK	1

TABLE 3N

MFGR_* - Interlacing Mode Codes

Definition Name	Definition Value
MFGR_INTERLACE_PIXEL	0
MFGR_INTERLACE_LINE	1
MFGR_INTERLACE_COMPONENT	2

TABLE 3O

SD_* - Scientific Data Set Configuration Codes

Definition Name	Definition Value
SD_UNLIMITED	0
SD_DIMVAL_BW_COMP	1
SD_DIMVAL_BW_INCOMP	0
SD_FILL	0
SD_NOFILL	256
SD_RAGGED	-1

TABLE 3P

SPECIAL_* - Special Element Identifier Codes

Definition Name	Definition Value
SPECIAL_LINKED	1
SPECIAL_EXT	2
SPECIAL_COMP	3
SPECIAL_VLINKED	4
SPECIAL_CHUNKED	5

TABLE 3Q

SUCCEED/FAIL - Routine Return Status Codes

Definition Name	Definition Value
SUCCEED	0
FAIL	-1