HDF4 Reference Manual

HDF4 Release 2.4 • June 2009

The HDF Group

The HDF Group and HDF Information and Contacts

Information regarding The HDF Group and HDF products is available from The HDF Group's website: http://www.hdfgroup.org

HDF Help Desk assistance is available via email: help@hdfgroup.org

Business queries and contacts can be made through the website or by mail:

http://www.hdfgroup.org/about/contact.html

The HDF Group

1901 South First Street, Suite C-2

Champaign, IL 61801

USA

Table of Contents

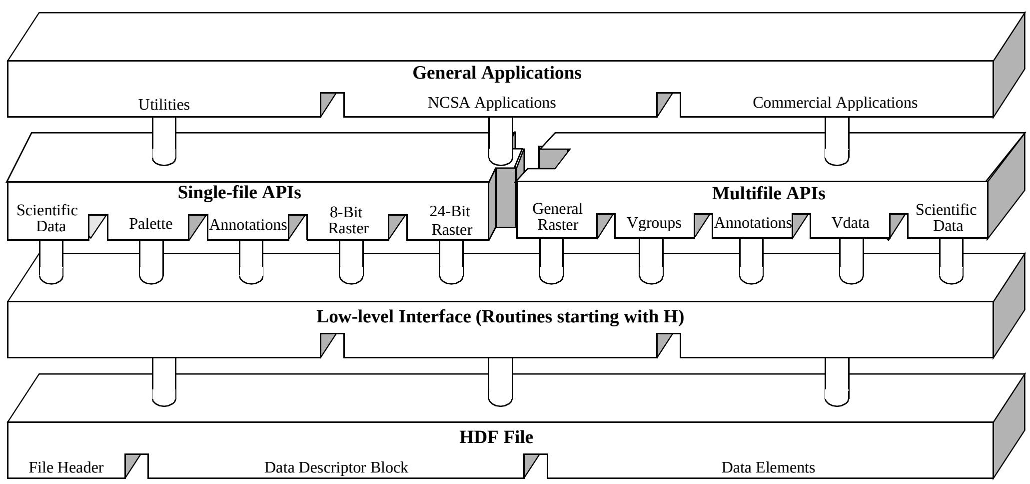
Section 1

Introduction to the HDF APIs

1.1 Overview of the HDF Interfaces

The HDF library structure consists of two interface layers and one application layer 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 operate on one file at a time, whereas the multifile interfaces can operate on several files simultaneously. The highest HDF layer includes various NCSA and commercial applications and a collection of command-line utilities that operate on HDF files or the data objects they contain.

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



1.2 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. These routines are prefaced with 'H'. For a more detailed discussion of the low-level interface, consult the *HDF Specifications and Developer's Guide* from the HDF WWW home page at `http://www.hdfgroup.org`.

The low-level interface provides a collection of routines that are prefaced with either 'H', 'HE', or 'HX'. The H routines are for managing HDF files. The HE routines provide error handlings. The HX routines are for managing HDF external 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.

1.3 Multifile 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 AN, GR, SD, VS, VSQ, VF, V, and VH. 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 raster image interfaces. The SD interface is the multifile version of the scientific data set interface. The VS, VSQ, and VF interfaces support the vdata model. The V and VH interfaces provide support 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 operations on more than one file at a time.

1.3.1 Scientific Data Sets: SD Interface

The scientific data set interface provides a collection of routines for reading and writing arrays of data. 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’.

1.3.2 Annotations: 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. Annotations consist of labels and descriptions.

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’.

1.3.3 General Raster Images: GR Interface

The routines in the GR interface provide multifile operations on general raster 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’.

1.3.4 Scientific Data Sets: netCDF Interface

The SD interface is designed to be as compatible 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 *HDF 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`.

1.3.5 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 records whose values are stored in fixed length fields. In addition to its records, a vdata may contain four kinds of identifying information: a name, class, data type and a number of field names.

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

1.3.6 Vdata Query: 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’. The equivalent FORTRAN-77 routine names are prefaced by ’vsq’.

1.3.7 Vdata Fields: 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.

1.3.8 Vgroups: V Interface

The vgroup interface provides a collection of routines for grouping and manipulating HDF data objects in the file. Each vgroup may contain one or more vdatas, vgroups, or other HDF data objects. In addition to its members, a vgroup may also be given a name and a class.

Every routine name in the vgroup interface are prefaced by ’V’. The equivalent FORTRAN-77 routine names are prefaced by ’vf’.

1.3.9 Vdata/Vgroups: 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’. The equivalent FORTRAN-77 routine names are prefaced by ’vh’.

1.3.10 Vgroup Inquiry: VQ 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 ’VQ’. The equivalent Fortran-77 routine names are prefaced by ’vq’.

1.4 Single-File Application Interfaces

The HDF single-file application interfaces include several independent modules each is 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.

1.4.1 24-bit Raster Image Sets: DF24 Interface

The HDF 24-bit raster interface provides a collection of routines for managing 24-bit raster image sets. A 24-bit raster image set is comprised of a 24-bit raster image array and its accompanied dimension record. Raster image sets may also include a palette.

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’.

1.4.2 8-bit Raster Image Sets: DFR8 Interface

The HDF 8-bit raster interface provides a collection of routines for managing 8-bit raster image sets. An 8-bit raster image set is comprised of an 8-bit raster image array and its accompanied dimension record. 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’.

1.4.3 Palettes: 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’.

1.4.4 Scientific Data Sets: DFSD Interface

There are two HDF interfaces that support multidimensional 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 existence of the single-file scientific data set interface is simply to support backward compatibility for previously created files and applications. It is recommended that the multifile scientific data set interface is to be used where possible.

The single-file scientific data set interface provides a collection of routines for reading and writing arrays of data. A scientific data set is comprised of a scientific data array and its accompanied rank, name and number type. 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’.

1.4.5 Annotations: 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.

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’.

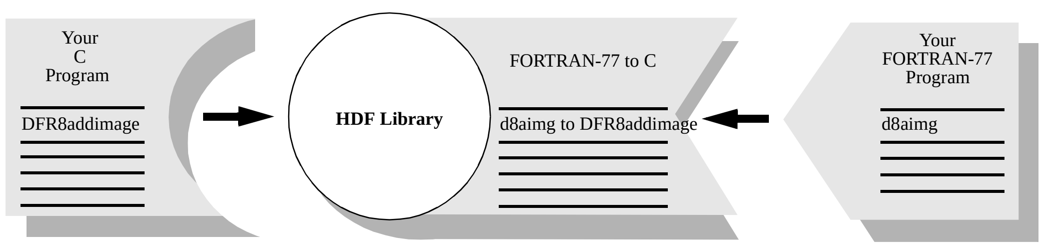
1.5 FORTRAN-77 and C Language Issues

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 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 1b **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 the compiler include and preprocess header files. If this capability is not available, the user may have to copy the declarations, definitions, and values needed from the files `dffunc.inc` and `hdf.inc` into the user application. If the capability is available, the files can be included in the Fortran code. The files reside in the `include/` subdirectory of the directory where the HDF library is installed on the user's system.

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 1A.)

TABLE 1A

Data Type Definitions

Definition Name	Definition Value	Description
DFNT_CHAR8	4	8-bit character type
DFNT_CHAR	4	Same as DFNT_CHAR8
DFNT_UCHAR8	3	8-bit unsigned character type
DFNT_UCHAR	3	Same as DFNT_UCHAR8
DFNT_INT8	20	8-bit integer type
DFNT_UINT8	21	8-bit unsigned integer type
DFNT_INT16	22	16-bit integer type
DFNT_UINT16	23	16-bit unsigned integer type
DFNT_INT32	24	32-bit integer type
DFNT_UINT32	25	32-bit unsigned integer type
DFNT_FLOAT32	5	32-bit floating-point type
DFNT_FLOAT64	6	64-bit floating-point type
DFNT_NINT8	(DFNT_NATIVE | DFNT_INT8)	8-bit native integer type
DFNT_NUINT8	(DFNT_NATIVE | DFNT_UINT8)	8-bit native unsigned integer type
DFNT_NINT16	(DFNT_NATIVE | DFNT_INT16)	16-bit native integer type
DFNT_NUINT16	(DFNT_NATIVE | DFNT_UINT16)	16-bit native unsigned integer type
DFNT_NINT32	(DFNT_NATIVE | DFNT_INT32)	32-bit native integer type

DFNT_NUINT32	(DFNT_NATIVE | DFNT_UINT32)	32-bit native unsigned integer type
DFNT_NFLOAT32	(DFNT_NATIVE | DFNT_FLOAT32)	32-bit native floating-point type
DFNT_NFLOAT64	(DFNT_NATIVE | DFNT_FLOAT64)	64-bit native floating-point type

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

1.5.6 String and 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 a string of an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold the data to be stored in the string.
- *real x(*)* means that *x* refers to an array of reals of indefinite size and of indefinite rank. It is the responsibility of the calling program to allocate an actual array with the correct number of dimensions and dimension sizes.
- *<valid numeric data type > x* means that *x* may have one of the numeric data types listed in the Description column of Table 1A on page 5.
- *<valid data type > x* means that *x* may have any of the data types listed in the Description column of Table 1A on page 5.

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 listed 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.

Error Code	Code Definition
DFE_BADOPEN	Miscellaneous open error.
DFE_NOTOPEN	File can't be closed because it hasn’t been opened.
DFE_CANTCLOSE	**fclose** error
DFE_READERROR	Read error.
DFE_WRITEERROR	Write error.
DFE_SEEKERROR	Seek error.
DFE_RDONLY	File is read only.
DFE_BADSEEK	Attempt to seek past end of element.
DFE_PUTELEM	**Hputelement** error.
DFE_GETELEM	**Hgetelement** error.
DFE_CANTLINK	Cannot initialize link information.
DFE_CANTSYNC	Cannot synchronize memory with file.
DFE_BADGROUP	Error from **DFdiread** in opening a group.
DFE_GROUPSETUP	Error from **DFdisetup** in opening a group.
DFE_PUTGROUP	Error on putting a tag/reference number pair into a group.
DFE_GROUPWRITE	Error when writing group contents.
DFE_DFNULL	Data file reference is a null pointer.
DFE_ILLTYPE	Data file contains an illegal type: internal error.
DFE_BADDDLIST	The DD list is non-existent: internal error.
DFE_NOTDFFILE	The current file is not an HDF file and it is not zero length.
DFE_SEEDTWICE	The DD list already seeded: internal error.
DFE_NOSUCHTAG	No such tag in the file: search failed.
DFE_NOFREEDD	There are no free 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.

Error Code	Code Definition
DFE_DUPDD	The new tag/reference number pair has been allocated.
DFE_CANTMOD	Old element doesn't exist. Cannot modify.
DFE_DIFFFILES	Attempt to merge objects in different files.
DFE_BADAID	An invalid AID was received.
DFE_OPENAID	Active AIDs still exist.
DFE_CANTFLUSH	Cannot flush DD back to file.
DFE_CANTUPDATE	Cannot update the DD block.
DFE_CANTHASH	Cannot add a DD to the hash table.
DFE_CANTDELDD	Cannot delete a DD in the file.
DFE_CANTDELHASH	Cannot delete a DD from the hash table.
DFE_CANTACCESS	Cannot access specified tag/reference number pair.
DFE_CANTENDACCESS	Cannot end access to data element.
DFE_TABLEFULL	Access table is full.
DFE_NOTINTABLE	Cannot find element in table.
DFE_UNSUPPORTED	Feature not currently supported.
DFE_NOSPACE	malloc failed.
DFE_BADCALL	Routine calls were in the wrong order.
DFE_BADPTR	NULL pointer argument was specified.
DFE_BADLEN	Invalid length was specified.
DFE_NOTENOUGH	Not enough space for the data.
DFE_NOVALS	Values were not available.
DFE_ARGS	Invalid arguments passed to the routine.
DFE_INTERNAL	Serious internal error.
DFE_NORESET	Too late to modify this value.
DFE_GENAPP	Generic application level error.
DFE_UNINIT	Interface was not initialized correctly.
DFE_CANTINIT	Cannot initialize the interface the operation requires.
DFE_CANTSHUTDOWN	Cannot shut down the interface the operation requires.
DFE_BADDIM	Negative number of dimensions, or zero dimensions, was specified.
DFE_BADFP	File contained an illegal floating point number.

Error Code	Code Definition
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_NOENCODER	Encoder not available.
DFE_NOSZLIB	SZIP library not available.
DFE_COMPVERSION	Version error from zlib Note: when Z_VERSION_ERROR (-6) returned from zlib.
DFE_READCOMP	Error in reading compressed data. Note: when one of the following error codes returned from zlib: Z_ERRNO (-1) Z_STREAM_ERROR (-2) Z_DATA_ERROR (-3) Z_MEM_ERROR (-4) Z_BUF_ERROR (-5)
DFE_NODIM	A dimension record was not associated with the image.
DFE_BADRIG	Error processing a RIG.

Error Code	Code Definition
DFE_RINOTFOUND	Cannot find raster image.
DFE_BADATTR	Invalid attribute.
DFE_BADTABLE	The nsdg table has incorrect information.
DFE_BADSDG	Error in processing an SDG.
DFE_BADNDG	Error in processing an NDG.
DFE_VGSIZE	Too many elements in the vgroup.
DFE_VTAB	Element not in `vtab[]`.
DFE_CANTADDELEM	Cannot add the tag/reference number pair to the vgroup.
DFE_BADVGNAME	Cannot set the vgroup name.
DFE_BADVGCLASS	Cannot set the vgroup class.
DFE_BADFIELDS	Invalid fields string passed to vset routine.
DFE_NOVS	Cannot find the vset in the file.
DFE_SYMSIZE	Too many symbols in the users table.
DFE_BADATTACH	Cannot write to a previously attached vdata.
DFE_BADVSNAME	Cannot set the vdata name.
DFE_BADVSCLASS	Cannot set the vdata class.
DFE_VSWRITE	Error writing to the vdata.
DFE_VSREAD	Error reading from the vdata.
DFE_BADVH	Error in the vdata header.
DFE_VSCANTCREATE	Cannot create the vdata.
DFE_VGCANTCREATE	Cannot create the vgroup.
DFE_CANTATTACH	Cannot attach to a vdata or vset.
DFE_CANTDETACH	Cannot detach a vdata or vset with write access.
DFE_BITREAD	A bit read error occurred.
DFE_BITWRITE	A bit write error occurred.
DFE_BITSEEK	A bit seek error occurred.
DFE_TBBTINS	Failed to insert the element into tree.
DFE_BVNEW	Failed to create a bit vector.
DFE_BVSET	Failed when setting a bit in a bit vector.

Error Code	Code Definition
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 4.1r4 library. For each interface, 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(type1 *parameter1*, type2 *parameter2*, ... , typeN *parameterN*)

parameter1	IN/ OUT:	Definition of the first parameter
parameter2	IN/ OUT:	Definition of the second parameter
. . .	. . .	
parameterN	IN/ OUT	Definition of the Nth parameter

Purpose Section containing the functionality of the routine.

Return value Section describing the return value, if any.

Description This optional section describes the proper use of the routine, the specification of the parameters, and any special circumstances surrounding the use of the routine. This section also identifies any prerequisite routines and provides appropriate references.

FORTRAN `This section provides a synopsis of the equivalent FORTRAN-77 routine or routines.`

ANnannlen/afannlen

int32 ANnannlen(int32 *ann_id*)

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

Purpose Returns the length of an annotation.

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

Description **ANnannlen** returns the number of characters contained in the annotation specified by the parameter *ann_id*. This function is commonly used to determine the size of a buffer to store the annotation upon reading.

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:	AN interface identifier returned by **ANstart**
annot_type	IN:	Type of the annotation
obj_tag	IN:	Tag of the object
obj_ref	IN:	Reference number of the object
ann_list	OUT:	Buffer for the annotation identifiers

Purpose Retrieves the annotation identifiers of an object.

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

Description **ANannlist** obtains a list of identifiers of the annotations that are of the type specified by the parameter *annot_type* and are attached to the object identified by its tag, *obj_tag*, and its reference number, *obj_ref*.

Since this routine is implemented only to obtain the identifiers of data annotations and not file annotations, the valid values of *annot_type* are `AN_DATA_LABEL` (or `0`) and `AN_DATA_DESC` (or `1`). To obtain file annotation identifiers, use **ANfileinfo** to determine the number of file labels and descriptions, and then use **ANselect** to obtain each file annotation identifier.

Sufficient space must be allocated for *ann_list* to hold the list of annotation identifiers. This can be done by using **ANnumann** to obtain the number of annotation identifiers to be retrieved, and then allocating memory for *ann_list* using this number.

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(ann_type **annot_type*)

annot_type	IN:	Type of the annotation

Purpose Returns the annotation tag corresponding to an annotation type.

Return value Returns the annotation tag (*ann_tag*) if successful, and `DFTAG_NULL` (or `0`) otherwise.

Description **ANatype2tag** returns the tag that corresponds to the annotation type specified by the parameter *annot_type*.

The following table lists the valid values of *annot_type* in the left column and the corresponding values for the returned annotation tag on the right.

Annotation Type	Annotation Tag
`AN_DATA_LABEL (or 0)`	`DFTAG_DIL (or 104)`
`AN_DATA_DESC (or 1)`	`DFTAG_DIA (or 105)`
`AN_FILE_LABEL (or 2)`	`DFTAG_FID (or 100)`
`AN_FILE_DESC (or 3)`	`DFTAG_FD (or 101)`

FORTRAN

```
integer function afatypetag(annot_type)

integer annot_type
```

ANcreate/afcreate

int32 ANcreate(int32 *an_id*, uint16 *obj_tag*, uint16 *obj_ref*, ann_type *annot_type*)

an_id	IN:	AN interface identifier returned by **ANstart**
obj_tag	IN:	Tag of the object to be annotated
obj_ref	IN:	Reference number of the object to be annotated
annot_type	IN:	Type of the data annotation

Purpose Creates a data annotation for an object.

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

Description **ANcreate** creates a data annotation of type *annot_type* for the object specified by its tag, *obj_tag*, and its reference number, *obj_ref*. The returned data annotation identifier can represent either a data label or a data description.

Valid values for *annot_type* are `AN_DATA_LABEL` (or `0`) or `AN_DATA_DESC` (or `1`).

Use **ANcreatef** to create a file annotation.

Currently, the user must write to a newly-created annotation before creating another annotation of the same type. Creating two consecutive annotations of the same type causes the second call to **ANcreate** to return `FAIL` (or `-1`).

FORTRAN

```
integer function afcreate(an_id, obj_tag, obj_ref, annot_type)

integer an_id, obj_tag, obj_ref, annot_type
```

ANcreatef/affcreate

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

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

Purpose Creates a file annotation.

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

Description **ANcreatef** creates a file annotation of the type specified by the parameter *annot_type*. The file annotation identifier returned can either represent a file label or a file description.

Valid values for *annot_type* are `AN_FILE_LABEL` (or `2`) and `AN_FILE_DESC` (or `3`).

Use **ANcreate** to create a data annotation.

Currently, the user must write to a newly-created annotation before creating another annotation of the same type. Creating two consecutive annotations of the same type causes the second call to **ANcreate** to return `FAIL` (or `-1`).

FORTRAN

```
integer function affcreate(an_id, annot_type)

integer an_id, annot_type
```

ANend/afend

int32 ANend(int32 *an_id*)

an_id	IN:	AN interface identifier returned by ANstart

Purpose Terminates access to an AN interface.

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

Description **ANend** terminates access to the AN interface identified by *an_id*, which is previously initialized by a call to **ANstart**. Note that there must be one call to **ANend** for each call to **ANstart**.

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 **ANendaccess** terminates access to the annotation identified by the parameter *ann_id*. Note that there must be one call to **ANendaccess** for every call to **ANselect**, **ANcreate** or **ANcreatef**.

FORTRAN

```
integer function afendaccess(ann_id)

integer ann_id
```

ANfileinfo/affileinfo

intn ANfileinfo(int32 *an_id*, int32 *n_*file_labels*, int32 **n_file_descs*, int32 **n_data_labels*, int32 **n_data_descs*)

an_id	IN:	AN interface identifier returned by **ANstart**
n_file_labels	OUT:	Number of file labels
n_file_descs	OUT:	Number of file descriptions
n_data_labels	OUT:	Number of data labels
n_data_descs	OUT:	Number of data descriptions

Purpose Retrieves the number of annotations of each type in a file.

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

Description **ANfileinfo** retrieves the total number of the four kinds of annotations and stores them in the appropriate parameters. The total number of data labels of all data objects in the file is stored in *n_data_labels*. The total number of data descriptions of all data objects in the file is stored in *n_data_descs*. The total number of file labels is stored in *n_file_labels* and the total number of file descriptions in *n_file_descs*.

Note that the numbers of data labels and descriptions refer to the total number of data labels and data descriptions in the file, not for a specific object. Use **ANnumann** to determine these numbers for a specific object.

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

FORTRAN

```
integer function affileinfo(an_id, n_file_labels, n_file_descs,
                    n_data_labels, n_data_descs)

integer an_id, n_file_labels, n_file_descs

integer n_data_labels, n_data_descs
```

ANget_tagref/afgettagref

int32 ANget_tagref(int32 *an_id*, int32 *index*, ann_type *annot_type*, uint16 **ann_tag*, uint16 **ann_ref*)

an_id	IN:	AN interface identifier returned by **ANstart**
index	IN:	Index of the annotation
annot_type	IN:	Type of the annotation
ann_tag	OUT:	Tag of the annotation
ann_ref	OUT:	Reference number of the annotation

Purpose Retrieves the tag/reference number pair of an annotation given its index and type.

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

Description **ANget_tagref** retrieves the tag and reference number of the annotation identified by its index, the parameter *index*, and by its annotation type, the parameter *annot_type*. The tag is stored in the parameter *ann_tag* and the reference number is stored in the parameter *ann_ref*.

The parameter *index* is a nonnegative integer and is less than the total number of annotations of type *annot_type* in the file. Use **ANfileinfo** to obtain the total number of annotations of each type in the file.

The following table lists the valid values of the parameter *annot_type* in the left column, and the corresponding values of the parameter *ann_tag* in the right column.

Annotation Type	Annotation Tag
AN_DATA_LABEL (or 0)	DFTAG_DIL (or 104)
AN_DATA_DESC (or 1)	DFTAG_DIA (or 105)
AN_FILE_LABEL (or 2)	DFTAG_FID (or 100)
AN_FILE_DESC (or 3)	DFTAG_FD (or 101)

FORTRAN

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

integer an_id, index, annot_type

integer ann_tag, ann_ref
```

ANid2tagref/afidtagref

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

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

Purpose Retrieves the tag/reference number pair of an annotation given its identifier.

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

Description **ANid2tagref** retrieves the tag/reference number pair of the annotation identified by the parameter *ann_id*. The tag is stored in the parameter *ann_tag* and the reference number is stored in the parameter *ann_ref*.

Possible values returned in *ann_tag* are `DFTAG_DIL` (or `104`) for a data label, `DFTAG_DIA` (or `105`) for a data description, `DFTAG_FID` (or `100`) for a file label and `DFTAG_FD` (or `101`) for a file description.

FORTRAN

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

integer ann_id, ann_tag, ann_ref
```

ANnumann/afnumann

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

an_id	IN:	AN interface identifier returned by **ANstart**
annot_type	IN:	Type of the annotation
obj_tag	IN:	Tag of the object
obj_ref	IN:	Reference number of the object

Purpose Returns the number of annotations of a given type attached to an object.

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

Description **ANnumann** returns the total number of annotations that are of type *annot_type* and that are attached to the object identified by its tag, *obj_tag*, and its reference number, *obj_ref*.

Since this routine is implemented only to obtain the total number of data annotations and not file annotations, the valid values of *annot_type* are `AN_DATA_LABEL` (or `0`) and `AN_DATA_DESC` (or `1`). To obtain the total number of file annotations or all data annotations, use **ANfileinfo**.

FORTRAN

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

integer an_id, obj_tag, obj_ref, annot_type
```

ANreadann/afreadann

int32 ANreadann(int32 *ann_id*, char* *ann_buf*, int32 *ann_length*)

ann_id	IN:	Annotation identifier returned by **ANcreate**, **ANcreatef** or **ANselect**
ann_buf	OUT:	Buffer for the annotation
ann_length	IN:	Length of the buffer *ann_buf*

Purpose Reads an annotation.

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

Description **ANreadann** reads the annotation identified by the parameter *ann_id* and stores the annotation in the parameter *ann_buf*.

The parameter *ann_length* specifies the size of the buffer *ann_buf*. If the length of the file or data label to be read is greater than or equal to *ann_length*, the label will be truncated to *ann_length* - 1 characters. If the length of the file or data description is greater than *ann_length*, the description will be truncated to *ann_length* characters. The HDF library adds a `NULL` character to the retrieved label but not to the retrieved description. The user must add a `NULL` character to the retrieved description if the C library string functions are to operate on this description.

FORTRAN

```
integer function afreadann(ann_id, ann_buf, ann_length)

integer ann_id, ann_length

character*(*) ann_buf
```

ANselect/afselect

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

an_id	IN:	AN interface identifier returned by **ANstart**
index	IN:	Location of the annotation in the file
annot_type	IN:	Type of the annotation

Purpose Obtains an existing annotation.

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

Description **ANselect** obtains the identifier of the annotation specified by its index, *index*, and by its annotation type, *annot_type*.

The parameter *index* is a nonnegative integer and is less than the total number of annotations of type *annot_type* in the file. Use **ANfileinfo** to obtain the total number of annotations of each type in the file.

Valid values of *annot_type* are `AN_DATA_LABEL` (or 0), `AN_DATA_DESC` (or 1), `AN_FILE_LABEL` (or 2), and `AN_FILE_DESC` (or 3).

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 AN interface.

Return value Returns the AN interface identifier (*an_id*) if successful and `FAIL` (or `-1`) otherwise.

Description **ANstart** initializes the AN interface for the file identified by the parameter *file_id*. A call to **ANstart** is required before any AN functions can be invoked. **ANstart** is used with the **ANend** function to define the extent of AN interface session. A call to **ANend** is required for each call to **ANstart**.

FORTRAN

```
integer function afstart(file_id)

integer file_id
```

ANtag2atype/aftagatype

ann_type ANtag2atype(uint16 *ann_tag*)

ann_tag IN: Tag of the annotation

Purpose Returns the annotation type corresponding to an annotation tag.

Return value Returns the annotation type if successful or `AN_UNDEF` (or `-1`) otherwise.

Description **ANtag2atype** returns the annotation type that corresponds to the annotation tag specified by the parameter *ann_tag*.

The following table lists the valid values of *ann_tag* in the left column and the corresponding values of the returned annotation type in the right column.

Annotation Tag	Annotation Type
`DFTAG_DIL (or 104)`	`AN_DATA_LABEL (or 0)`
`DFTAG_DIA (or 105)`	`AN_DATA_DESC (or 1)`
`DFTAG_FID (or 100)`	`AN_FILE_LABEL (or 2)`
`DFTAG_FD (or 101)`	`AN_FILE_DESC (or 3)`

FORTRAN

```
integer function aftagatype(ann_tag)

integer ann_tag
```

ANtagref2id/aftagrefid

int32 ANtagref2id(int32 *an_id*, uint16 *ann_tag*, uint16 *ann_ref*)

an_id	IN:	AN interface identifier returned by **ANstart**
ann_tag	IN:	Tag of the annotation
ann_ref	IN:	Reference number of the annotation

Purpose Returns the identifier of an annotation given its tag/reference number pair.

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

Description **ANtagref2id** returns the identifier of the annotation specified by its tag, *ann_tag*, and its reference number, *ann_ref*.

Valid values of *ann_tag* are `DFTAG_DIL` (or `104`) for a data label, `DFTAG_DIA` (or `105`) for a data description, `DFTAG_FID` (or `100`) for a file label, and `DFTAG_FD` (or `101`) for a file description.

FORTRAN

```
integer function aftagrefid(an_id, ann_tag, ann_ref)

integer an_id, ann_tag, ann_ref
```

ANwriteann/afwriteann

int32 ANwriteann(int32 *ann_id*, char* *ann*, int32 *ann_length*)

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

Purpose Writes an annotation.

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

Description **ANwriteann** writes the annotation text provided in the parameter *ann* to the annotation specified by the parameter *ann_id*. The parameter *ann_length* specifies the number of characters in the annotation text.

If the annotation has already been written with text, **ANwriteann** will overwrite the current text.

FORTRAN

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

integer ann_id, ann_length

character*(*) ann
```

GRattrinfo/mgatinf

intn GRattrinfo(int32 *[obj]_id*, int32 *attr_index*, char **name*, int32 **data_type*, int32 **count*)

[obj]_id	IN:	Raster image identifier (*ri_id*), returned by **GRcreate** or **GRselect**, or GR interface identifier (*gr_id*), returned by **GRstart**
attr_index	IN:	Index of the attribute
name	OUT:	Buffer for the name of the attribute
data_type	OUT:	Data type of the attribute
count	OUT:	Number of attribute values

Purpose Retrieves information about an attribute.

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

Description **GRattrinfo** retrieves the name, data type, and number of values of the attribute, specified by its index, *attr_index*, for the data object identified by the parameter *obj_id*. The name is stored in the parameter *name*, the data type is stored in the parameter *data_type*, and the number of values is stored in the parameter *count*. If the value of any of the output parameters is `NULL`, the corresponding information will not be retrieved.

The value of the parameter *attr_index* can be obtained using **GRfindattr**, **GRnametoindex** or **GRreftoindex**, depending on available information. Valid values of *attr_index* range from 0 to the total number of attributes attached to the object - 1. The total number of attributes attached to the file can be obtained using the routine **GRfileinfo**. The total number of attributes attached to an image can be obtained using the routine **GRgetiminfo** .

FORTRAN

```
integer function mgatinf([obj]_id, attr_index, name, data_type,
                         count)

integer [obj]_id, data_type, attr_index, count

character*(*) name
```

GRcreate/mgcreat

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

gr_id	IN:	GR interface identifier returned by **GRstart**
name	IN:	Name of the raster image
ncomp	IN:	Number of pixel components in 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 new raster image.

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

Description **GRcreate** creates a raster image with the values provided in the parameters *name*, *ncomp*, *data_type*, *interlace_mode* and *dim_sizes*.

The parameter *name* specifies the name of the image and must not be `NULL`. The length of the name should not be longer than `MAX_GR_NAME` (or `256`).

The parameter *ncomp* specifies the number of pixel components in the raster image and must have a value of at least 1.

The parameter *data_type* specifies the type of the raster image data and can be any of the data types supported by the HDF library. The data types supported by HDF are listed in Table 1A in Section I of this manual.

The parameter *interlace_mode* specifies the interlacing in which the raster image is to be written. The valid values of *interlace_mode* are: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`).

The array *dimsizes* specifies the size of the two dimensions of the image. The dimensions must be specified and their values must be greater than `0`.

Once a raster image has been created, it is not possible to change its name, data type, dimension sizes or number of pixel components. However, it is possible to create a raster image and close the file before writing any data values to it. Later, the values can be added to or modified in the raster image, which then can be obtained using **GRselect**.

Images created with the GR interface are actually written to disk in pixel interlace mode; any user-specified interlace mode is stored in the file with the image and the image is automatically converted to that mode when it is read with a GR interface function.

Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- **o** For **SDreaddata**, those parameters are expressed in (*y*,*x*) or [*row*,*column*] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- **o** For **GRreadimage**, those parameters are expressed in (*x*,*y*) or [*column*,*row*] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

FORTRAN

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

integer gr_id, data_type, interlace_mode, ncomp, dim_sizes(2)

character*(*) name
```

GRend/mgend

intn GRend(int32 *gr_id*)

gr_id	IN:	GR interface identifier returned by **GRstart**

Purpose Terminates the GR interface session.

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

Description **GRend** terminates the GR interface session identified by the parameter *gr_id*.

GRend, together with **GRstart**, defines the extent of a GR interface session. **GRend** disposes of the internal structures initialized by the corresponding call to **GRstart**. There must be a call to **GRend** for each call to **GRstart**; failing to provide one may cause loss of data.

GRstart and **GRend** do not manage file access; use **Hopen** and **Hclose** to open and close HDF files. **Hopen** must be called before **GRstart** and **Hclose** must be called after **GRend**.

FORTRAN

```
integer function mgend(gr_id)

integer gr_id
```

GRendaccess/mgendac

intn GRendaccess(int32 *ri_id*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**

Purpose Terminates access to a raster image.

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

Description **GRendaccess** terminates access to the raster image identified by the parameter *ri_id* and disposes of the raster image identifier. This access is initiated by either **GRselect** or **GRcreate**. There must be a call to **GRendaccess** for each call to **GRselect** or **GRcreate**; failing to provide this will result in loss of data. Attempts to access a raster image identifier disposed of by **GRendaccess** will result in an error condition.

FORTRAN

```
integer function mgendac(ri_id)

integer ri_id
```

GRfileinfo/mgfinfo

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

gr_id	IN:	GR interface identifier returned by **GRstart**
n_images	OUT:	Number of raster images in the file
n_file_attrs	OUT:	Number of global attributes in the file

Purpose Retrieves the number of raster images and the number of global attributes in the file.

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

Description **GRfileinfo** retrieves the number of raster images and the number of global attributes for the GR interface identified by the parameter *gr_id*, and stores them into the parameters *n_images* and *n_file_attrs*, respectively.

The term "global attributes" refers to attributes that are assigned to the file instead of individual raster images. These attributes are created by **GRsetattr** with the object identifier parameter set to a GR interface identifier (*gr_id*) rather than a raster image identifier (*ri_id*).

GRfileinfo is useful in finding the range of acceptable indices for **GRselect** calls.

FORTRAN

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

integer gr_id, n_images, n_file_attrs
```

GRfindattr/mgfndat

int32 GRfindattr(int32 *[obj]_id*, char **attr_name*)

[obj]_id	IN:	Raster image identifier (*ri_id*), returned by **GRcreate** or **GRselect**, or GR interface identifier (*gr_id*), returned by **GRstart**
attr_name	IN:	Name of the attribute

Purpose Finds the index of a data object's attribute given an attribute name.

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

Description **GRfindattr** returns the index of the attribute whose name is specified by the parameter *attr_name* for the object identified by the parameter *obj_id.*

FORTRAN

```
integer function mgfndat([obj]_id, attr_name)

integer [obj]_id

character*(*) attr_name
```

GRgetattr/mggnatt/mggcatt

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

[obj]_id	IN:	Raster image identifier (*ri_id*), returned by **GRcreate** or **GRselect**, or GR interface identifier (*gr_id*), returned by **GRstart**
attr_index	IN:	Index of the attribute
values	OUT:	Buffer for the attribute values

Purpose Reads the values of an attribute for a data object.

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

Description **GRgetattr** obtains all values of the attribute that is specified by its index, *attr_index*, and is attached to the object identified by the parameter *obj_id*. The values are stored in the buffer *values*.

The value of the parameter *attr_index* can be obtained by using **GRfindattr**, **GRnametoindex**, or **GRreftoindex**, depending on available information. Valid values of *attr_index* range from 0 to the total number of attributes of the object - 1. The total number of attributes attached to the file can be obtained using the routine **GRfileinfo**. The total number of attributes attached to the image can be obtained using the routine **GRgetiminfo**.

GRgetattr only reads all values assigned to the attribute and not a subset.

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

FORTRAN

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

integer [obj]_id, attr_index

<valid numeric data type> values(*)


integer function mggcatt([obj]_id, attr_index, values)

integer [obj]_id, attr_index

character*(*) values
```

GRgetchunkinfo/mggichnk

intn GRgetchunkinfo(int32 *ri_id*, HDF_CHUNK_DEF **cdef*, int32 **flag*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**

C only:

cdef	OUT:	Pointer to the chunk definition
flag	OUT:	Pointer to the compression flag

Fortran only:

dim_length	OUT:	Array of chunk dimensions
flag	OUT:	Compression flag

Purpose Retrieves chunking information for a raster image.

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

Description **GRgetchunkinfo** retrieves chunking information about the raster image identified by the parameter *ri_id* into the parameters *cdef* and *flags* in C, and into the parameters *dim_length* and *flag* in Fortran. Note that only chunk dimensions are retrieved, compression information is not available.

The value returned in the parameter *flag* indicates if the raster image is not chunked, chunked, or chunked and compressed. The following table shows the possible values of the parameter *flag* and the corresponding characteristics of the raster image.

Values of *flag* in C	Values of *flag* in Fortran	Raster Image Characteristics
HDF_NONE	-1	Not chunked
HDF_CHUNK	0	Chunked and not compressed
HDF_CHUNK | HDF_COMP	1	Chunked and compressed with either the run-length encoding (RLE), Skipping Huffman or GZIP compression algorithms

In C, if the raster image is chunked and not compressed, **GRgetchunkinfo** fills the array `chunk_lengths` in the union `cdef` with the values of the corresponding chunk dimensions. If the raster image is chunked and compressed, **GRgetchunkinfo** fills the array `chunk_lengths` in the structure `comp` of the union `cdef` with the values of the corresponding chunk dimensions. Refer to the page on **GRsetchunk** in this manual for specific information on the union `HDF_CHUNK_DEF`. In Fortran, chunk dimensions are retrieved into the array `dim_length`. If the chunk length for each dimension is not needed, `NULL` can be passed in as the value of the parameter `cdef` in C.

FORTRAN

```
integer function mggichnk(ri_id, dim_length, flag)

integer ri_id, dim_length, flag
```

GRgetcompinfo/mggcompress

intn GRgetcompinfo(int32 *ri_id*, comp_coder_t **comp_type*, comp_info **c_info*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
comp_type	OUT:	Type of compression
C only: *c_info*	OUT:	Pointer to compression information structure
Fortran only: *comp_prm*	OUT:	Compression parameters array

Purpose Retrieves raster image data compression type and compression information.

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

Description **GRgetcompinfo** retrieves the compression type and compression information for the specified raster image. **GRgetcompinfo** replaces **GRgetcompress** because this function has flaws, causing failure for some chunked and chunked/compressed data.

The compression method is returned in the parameter *comp_type*. Valid values of *comp_type* are as follows:

`COMP_CODE_NONE` (or `0`) for no compression
`COMP_CODE_RLE` (or `1`) for RLE run-length encoding
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for SZIP compression
`COMP_CODE_JPEG` (or `7`) for JPEG compression

When a compression method requires additional parameters, those values are returned in the *c_info* struct in C and the array parameter *comp_prm* in Fortran.

The *c_info* struct is of type `comp_info`, contains algorithm-specific information for the library compression routines, and is described in the `hcomp.h` header file and in the **GRsetcompress** entry in this reference manual..

The *comp_prm* parameter is an array of one element:

- With Skipping Huffman compression, *comp_prm(1)* contains the skip value, `skphuff_skp_size`.
- In the case of GZIP compression, *comp_prm(1)* contains the deflation value, `deflate_value`.
- *comp_prm* is ignored with other compression methods. (There are no relevant RLE parameters and the `quality` and `force_baseline` data are not available for JPEG images. If **GRgetcompinfo** is called for either an RLE or a JPEG image, the function will return only the compression type; *c_info* will contain only zeros.)
- Currently, Fortran GR interface doesn’t support Szip compression.

FORTRAN

```
integer function mggcompress(ri_id, comp_type, comp_prm)

integer ri_id, comp_type, comp_prm(1)
```

GRgetiminfo/mggiinf

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

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
gr_name	OUT:	Buffer for the name of the raster image
ncomp	OUT:	Number of components in the raster image
data_type	OUT:	Data type of the raster image data
interlace_mode	OUT:	Interlace mode of the stored raster image data
dim_sizes	OUT:	Sizes of raster image dimension
num_attrs	OUT:	Number of attributes attached to the raster image

Purpose Retrieves general information about a raster image.

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

Description **GRgetiminfo** retrieves the name, number of components, data type, interlace mode, dimension sizes, and number of attributes of the raster image identified by the parameter *ri_id*.

GRgetiminfo stores the name, number of components, data type, interlace mode and dimension sizes of the image in the parameters *gr_name, ncomp*, *data_type*, *interlace_mode*, and *dim_sizes*, respectively. It also retrieves the number of attributes attached to the image into the parameter *num_attrs*. If the value of any of the output parameters are set to `NULL` in C, the corresponding information will not be retrieved.

The buffer *gr_name* is assumed to have sufficient space allocated to store the entire name of the raster image.

The valid values of the parameter *data_type* are listed in Table 1A in Section I of this manual.

FORTRAN

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

integer ri_id, ncomp, data_type, interlace_mode, num_attrs

integer dim_sizes[2]

character*(*) gr_name
```

GRgetlutid/mggltid

int32 GRgetlutid(int32 *ri_id*, int32 *pal_index*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
pal_index	IN:	Index of the palette

Purpose Gets the identifier of a palette given its index.

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

Description **GRgetlutid** gets the identifier of the palette attached to the raster image identified by the parameter *ri_id.* The palette is identified by its index, *pal_index.*

Currently, only one palette can be assigned to a raster image, which means that *pal_index* should always be set to `0`.

FORTRAN

```
integer function mggltid(ri_id, pal_index)

integer ri_id, pal_index
```

GRgetlutinfo/mgglinf

intn GRgetlutinfo(int32 *pal_id*, int32 **ncomp*, int32 **data_type*, int32 **interlace_mode*, 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
interlace_mode	OUT:	Interlace mode of the stored palette data
num_entries	OUT:	Number of color lookup table entries in the palette

Purpose Retrieves information about a palette.

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

Description **GRgetlutinfo** retrieves the number of pixel components, data type, interlace mode, and number of color lookup table entries of the palette identified by the parameter *pal_id*. These values are stored in the parameters *ncomp*, *data_type*, *interlace_mode*, and *num_entries*, respectively. In C if the value of any of the output parameters are set to `NULL`, the corresponding information will not be retrieved.

FORTRAN

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

integer pal_id, ncomp, data_type, interlace_mode, num_entries
```

GRgetnluts/mggnluts

intn GRgetnluts(int32 *ri_id*)

ri_id	IN:	Data set identifier returned by **GRcreate** or **GRselect**

Purpose Retrieves the number of palettes for an image.

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

Description **GRgetnluts** retrieves the number of palettes (or color look-up tables, commonly abbreviated as LUTs) available for the specified raster image.

There can currently be either 0 or 1 palettes assigned to an image. If multiple palettes are supported in a future release, this function may return values greater than 1.

FORTRAN

```
integer function mggnluts(ri_id)

integer ri_id
```

GRidtoref/mgid2rf

uint16 GRidtoref(int32 *ri_id*)

ri_id	IN:	Raster image identifier returned by **GRselect** or **GRcreate**

Purpose Maps a raster image identifier to a reference number.

Return value Returns the reference number of the raster image if successful and `0` otherwise.

Description **GRidtoref** returns the reference number of the raster image identified by the parameter *ri_id*.

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

FORTRAN

```
integer function mgid2rf(ri_id)

integer ri_id
```

GRluttoref/mglt2rf

uint16 GRluttoref(int32 *pal_id*)

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

Purpose Maps a palette identifier to a reference number.

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

Description **GRluttoref** returns the reference number of the palette identified by the parameter *pal_id*.

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

FORTRAN

```
integer function mglt2rf(pal_id)

integer pal_id
```

GRnametoindex/mgn2ndx

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

gr_id	IN:	GR interface identifier returned by **GRstart**
ri_name	IN:	Name of the raster image

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

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

Description **GRnametoindex** returns, for the GR interface identified by the parameter *gr_id*, the index (*index*) of the raster image named *gr_name*.

The value of *index* can be passed into **GRselect** to obtain the raster image identifier (*ri_id*).

FORTRAN

```
integer function mgn2ndx(gr_id, gr_name)

integer gr_id

character*(*) gr_name
```

GRreadchunk/mgrchnk/mgrcchnk

intn GRreadchunk(int32 *ri_id*, int32 **origin*, VOIDP *datap*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
origin	IN:	Origin of the chunk to be read
datap	IN:	Buffer for the chunk to be read

Purpose Reads a data chunk from a chunked raster image (pixel-interlace only)

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

Description **GRreadchunk** reads the entire chunk of data from the chunked raster image identified by the parameter *ri_id* and stores it in the buffer datap. Chunk to be read is specified by the parameter origin. This function has less overhead than **GRreadimage** and should be used whenever an entire chunk of data is to be read.

GRreadchunk will return `FAIL` (or `-1`) when an attempt is made to use it to read from a non-chunked raster image.

The parameter *origin* is a two-dimensional array which specifies the coordinates of the chunk according to the chunk position in the overall chunk array. Refer to Chapter 8, "General Raster Images (GR API)," in the *HDF User's Guide*.

The buffer *datap* contains the chunk data organized in pixel interlace mode.

FORTRAN

```
integer mgrchnk(ri_id, origin, datap)

integer ri_id, origin(2)

<valid_numeric_datatype> datap(*)


integer mgrcchnk(ri_id, origin, char_datap)

integer ri_id, origin(2)

character*(*) char_datap
```

GRreadimage/mgrdimg/mgrcimg

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

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
start	IN:	Array specifying the starting location from where raster image data is read
stride	IN:	Array specifying the interval between the values that will be read along each dimension
edge	IN:	Array specifying the number of values to be read along each dimension
data	OUT:	Buffer for the image data

Purpose Reads a raster image.

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

Description **GRreadimage** reads the subsample of the raster image specified by the parameter *ri_id* into the buffer *data*. The subsample is defined by the values of the parameters *start*, *stride* and *edge*.

The array *start* specifies the starting location of the subsample to be read. Valid values of each element in the array *start* are `0` to (the size of the corresponding raster image dimension `- 1`). The first element of the array *start* specifies an offset from the beginning of the array *data* along the fastest-changing dimension, which is the second dimension in C and the first dimension in Fortran. The second element of the array *start* specifies an offset from the beginning of the array *data* along the second fastest-changing dimension, which is the first dimension in C and the second dimension in Fortran. For example, if the first value of the array *start* is `2` and the second value is `3`, the starting location of the subsample to be read is at the fourth row and third column in C, and at the third row and fourth column in Fortran.

The array *stride* specifies the reading pattern along each dimension. For example, if one of the elements of the array *stride* is `1`, then every element along the corresponding dimension of the array *data* will be read. If one of the elements of the array *stride* is `2`, then every other element along the corresponding dimension of the array *data* will be read, and so on. The correspondence between elements of the array *stride* and the dimensions of the array *data* is the same as described above for the array *start*.

Each element of the array *edges* specifies the number of data elements to be read along the corresponding dimension. The correspondence between the elements of the array *edges* and the dimensions of the array *data* is the same as described above for the array *start*.

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

Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- For **SDreaddata**, those parameters are expressed in (`y,x`) or [`row,column`] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For **GRreadimage**, those parameters are expressed in (`x,y`) or [`column,row`] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

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 returned by **GRgetlutid**
pal_data	OUT:	Buffer for the palette data

Purpose Reads a palette.

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

Description **GRreadlut** reads the palette specified by the parameter *pal_id* into the buffer *pal_data*.

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

FORTRAN

```
integer function mgrdlut(pal_id, pal_data)

integer pal_id

<valid numeric data type> pal_data(*)


integer function mgrclut(pal_id, pal_data)

integer pal_id

character*(*) pal_data
```

GRreftoindex/mgr2idx

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

gr_id	IN:	GR interface identifier returned by **GRstart**
gr_ref	IN:	Reference number of the raster image

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

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

Description **GRreftoindex** returns the index of the raster image specified by the parameter *gr_ref*.

FORTRAN

```
integer function mgr2idx(gr_id, gr_ref)

integer gr_id, gr_ref
```

GRreqimageil/mgrimil

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

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
interlace_mode	IN:	Interlace mode

Purpose Specifies the interlace mode to be used in the subsequent raster image read operation(s).

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

Description **GRreqimageil** requests that the subsequent read operations on the image identified by the parameter *ri_id* use the interlace mode specified by the parameter *interlace_mode.*

The parameter *interlace_mode* specifies the interlace mode in which the data will be stored in memory when being read. Valid values of the parameter *interlace_mode* are `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`.)

In the file, the image is always stored in pixel interlace mode, i.e. `MFGR_INTERLACE_PIXEL`. The interlace mode of the raster image specified at creation time is stored in the file along with the raster image. If **GRreqimageil** is not called prior to the call to **GRreadimage**, the raster image will be read and stored in memory according to the interlace mode specified at creation. If **GRreqimageil** is called before **GRreadimage**, **GRreadimage** will read the raster image and store it according to the interlace mode specified in the call to **GRreqimageil**.

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:	Raster image identifier returned by **GRcreate** or **GRselect**
interlace_mode	IN:	Interlace mode

Purpose Specifies the interlace mode to be used in the next palette read operation(s).

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

Description **GRreqlutil** requests that the subsequent read operations on the palette attached to the image identified by the parameter *ri_id*, use the interlace mode *interlace_mode*.

The parameter *interlace_mode* specifies the interlace mode in which the data will be stored in memory when being read. Valid values of the parameter *interlace_mode* are `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`).

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

Purpose Selects the existing raster image.

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

Description **GRselect** obtains the identifier of the raster image specified by the its index, *index*.

Valid values of the parameter *index* range from `0` to (the total number of raster images in the file `- 1`). The total number of the raster images in the file can be obtained by using **GRfileinfo**.

FORTRAN

```
integer function mgselct(gr_id, index)

integer gr_id, index
```

GRsetattr/mgsnatt/mgscatt

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

[obj]_id	IN:	Raster image identifier (*ri_id*), returned by **GRcreate** or **GRselect** or GR interface identifier (*gr_id*), returned by **GRstart**
attr_name	IN:	Name of the attribute
data_type	IN:	Data type of the attribute
count	IN:	Number of values in the attribute
values	IN:	Buffer for the attribute values

Purpose Assigns an attribute to a raster image or a file.

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

Description **GRsetattr** attaches the attribute to the object specified by the parameter *obj_id*. The attribute is defined by its name, *attr_name*, data type, *data_type*, number of attribute values, *count*, and the attribute values, *values*. **GRsetattr** provides a generic way for users to define metadata. It implements the `label = value` data abstraction.

If an GR interface identifier (*gr_id)* is specified as the parameter *obj_id*, a global attribute is created which applies to all objects in the file. If a raster image identifier (*ri_id*) is specified as the parameter *obj_id*, an attribute is attached to the specified raster image.

The parameter *attr_name* can be any ASCII string.

The parameter *data_type* can contain any data type supported by the HDF library. These data types are listed in Table 1A in Section I of this manual.

Attribute values are passed in the parameter *values*. The number of attribute values is defined by the parameter *count* . If more than one value is stored, all values must have the same data type. If an attribute with the given name, data type and number of values exists, it will be overwritten. Currently, the only predefined attribute is the fill value, identified by the `FILL_ATTR` definition.

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

FORTRAN

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

integer ri_id, comp_type, comp_prm(*)

integer [obj]_id, data_type, count

character*(*) attr_name

<valid numeric data type> values(*)
```

```
integer function mgscatt([obj]_id, attr_name, data_type, count,
                  values)

integer [obj]_id, data_type

integer count

character*(*) values, attr_name
```

GRsetcompress/mgscompress

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

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
comp_type	IN:	Compression method for the image data
C only: *c_info*	IN:	Pointer to the `comp_info` union
Fortran only: *comp_prm*	IN:	Compression parameters array

Purpose Specifies if the raster image will be stored in a file as a compressed raster image.

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

Description **GRsetcompress** specifies if the raster image specified by the parameter *ri_id* will be stored in the file in compressed format.

The compression method is specified by the parameter *comp_type*. Valid values of the parameter *comp_type* are:

`COMP_CODE_NONE` (or `0`) for no compression
`COMP_CODE_RLE` (or `1`) for RLE run-length encoding
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for SZIP compression
`COMP_CODE_JPEG` (or `7`) for JPEG compression

The compression method parameters are specified by the parameter *c_info* in C and the parameter *comp_prm* in Fortran. The parameter *c_info* has type `comp_info`, which is described in the hcomp.h header file. It contains algorithm-specific information for the library compression routines.

The skipping size for the Skipping Huffman algorithm is specified in the field `c_info.skphuff.skp_size` in C and in the parameter *comp_prm*(1) in Fortran.

The deflate level for the GZIP algorithm is specified in the field `c_info.deflate.level` in C and in the parameter *comp_prm*(1) in Fortran.

The parameter *c_info* is a pointer to a union structure of type `comp_info`. This union structure is defined as follows:

```
typedef union tag_comp_info
{
    struct
    {
    /* Not used by GRsetcompress */
    } jpeg;

    struct
    {
    /* Not used by GRsetcompress */
    } nbit;

    struct
    {  /* struct to contain info about how to compress size of the
        elements when skipping */
        intn skp_size;
    } skphuff;

    struct
    {  /* struct to contain info about how to compress or decom-
      press
        gzip encoded dataset how hard to work when compressing
        data*/
        intn level;
    } deflate;

      struct
        {
            int32 options_mask;   /* IN */
            int32 pixels_per_block;   /* IN */
            int32 pixels_per_scanline; /* OUT: computed */
            int32 bits_per_pixel; /* OUT: size of NT */
            int32 pixels; /* OUT: size of dataset or chunk */
         }
       szip;  /* for szip encoding */
} comp_info;
```

FORTRAN

```
integer mgscompress(ri_id, comp_type, comp_prm)

integer ri_id, comp_type, comp_prm(*)
```

GRsetchunk/mgschnk

intn GRsetchunk(int32 *ri_id*, HDF_CHUNK_DEF *cdef*, int32 *flags*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
C only:		
cdef	IN:	Chunk definition
flags	IN:	Compression flags
Fortran only:		
dim_length	IN:	Chunk dimensions array
comp_type	IN:	Type of compression
comp_prm	IN:	Compression parameters array

Purpose Makes a raster image a chunked raster image.

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

Description **GRsetchunk** makes the raster image specified by the parameter *ri_id* a chunked raster image according to the chunking and compression information provided in the parameters *cdef* and *flags* in C, or in the parameters *comp_type* and *comp_prm* in Fortran.

C only:

The parameter *cdef* is a union of type HDF_CHUNK_DEF, which is defined as follows:

```
typedef union hdf_chunk_def_u
    {
     int32 chunk_lengths[2];  /* chunk lengths along each dim */

     struct
          {
           int32 chunk_lengths[2];
           int32 comp_type;              /* compression type */
           struct comp_info cinfo;
          } comp;

     struct
          {
          /* is not used in GR interface */
          } nbit;
     } HDF_CHUNK_DEF
```

Valid values of the parameter *flags* are `HDF_CHUNK` for chunked and uncompressed data and (`HDF_CHUNK | HDF_COMP`) for chunked and compressed data. Data can be compressed using run-length encoding (RLE), Skipping Huffman, GZIP, or Szip compression algorithms.

If the parameter *flags* has a value of `HDF_CHUNK`, the chunk dimensions must be specified in the field `cdef.chunk_lengths[]`. If the parameter *flags* has a value of (`HDF_CHUNK | HDF_COMP`), the following must be specified:

1) The chunk dimensions in the field `cdef.comp.chunk_lengths[]`.
2) The compression type in the field `cdef.comp.comp_type`. Valid values of compression type values are listed below.

`COMP_CODE_NONE` (or `0`) for uncompressed data
`COMP_CODE_RLE` (or `1`) for RLE compression
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression
`COMP_CODE_SZIP` (or `5`) for Szip compression

For Skipping Huffman and GZIP compression, parameters are passed in corresponding fields of the structure `cinfo`.

- o Specify skipping size for Skipping Huffman compression in the field `cdef.comp.cinfo.skphuff.skp_size`, which must be an integer of value 1 or greater.
- o Specify the deflate level for GZIP compression in the field `cdef.comp.cinfo.deflate_level`. Valid deflate level values are integers between 0 and 9 inclusive.
- o Specify the options mask and the number of pixels per block for Szip compression in the fields `c_info.szip.options_mask` and `c_info.szip.pixels_per_block`, respectively.

Refer to the **SDsetcompress** entry in this reference manual for details on these parameters.

Fortran only:

The *dim_length* array specifies the chunk dimensions.

The parameter *comp_type* specifies the compression type. Valid compression types and their values used are defined in the hdf.inc file, and are listed below.

`COMP_CODE_NONE` (or `0`) for uncompressed data
`COMP_CODE_RLE` (or `1`) for RLE compression
`COMP_CODE_SKPHUFF` (or `3`) for Skipping Huffman compression
`COMP_CODE_DEFLATE` (or `4`) for GZIP compression

The parameter *comp_prm* specifies the compression parameters for the Skipping Huffman and GZIP compression methods. It contains only one element which is set to the skipping size for Skipping Huffman compression or the deflate level for GZIP compression. Currently, Fortran GR interface does not support Szip compression.

FORTRAN

```
integer function mgschnk(ri_id, dim_length, comp_type, comp_prm)

integer ri_id, dim_length, comp_type, comp_prm
```

GRsetchunkcache/mgscchnk

intn GRsetchunkcache(int32 *ri_id*, int32 *maxcache*, int32 *flags*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
maxcache	IN:	Maximum number of chunks to cache
flags	IN:	Flags determining the behavior of the routine

Purpose Specifies the maximum number of chunks to cache.

Return value Returns the value of the parameter *maxcache* if successful and `FAIL` (or `-1`) otherwise.

Description **GRsetchunkcache** sets the maximum number of chunks to be cached for the chunked raster image specified by the parameter *ri_id*. The maximum number of the chunks is specified by the parameter *maxcache*.

Currently, the only valid value of the parameter *flags* is `0`.

If **GRsetchunkcache** is not called, the maximum number of chunks in the cache is set to the number of chunks along the fastest-changing dimension. Refer to the discussion of the **GRsetchunkcache** routine in the *HDF User's Guide* for more specific information on the routine's behavior.

FORTRAN

```
integer function mgscchnk(ri_id, maxcache, flags)

integer ri_id, maxcache, flags
```

GRsetexternalfile/mgsxfil

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

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
filename	IN:	Name of the external file
offset	IN:	Offset in bytes from the beginning of the external file to where the data will be written

Purpose Specifies that the raster image will be written to an external file.

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

Description **GRsetexternalfile** specifies that the raster image identified by the parameter *ri_id* will be written to the external file specified by the parameter *filename* at the offset specified by the parameter *offset.*

Data can only be moved once for any given raster image, and it is the user's responsibility to make sure the external data file is kept with the “original” file.

If the raster image already exists, its data will be moved to the external file . Space occupied by the data in the primary file will not be released. To release the space in the primary file use the `hdfpack` command-line utility. If the raster image does not exist, its data will be written to the external file during the subsequent calls to **GRwritedata**.

See the reference manual entries for **HXsetcreatedir** and **HXsetdir** for more information on the options available for accessing external files.

FORTRAN

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

integer ri_id, offset

character*(*) filename
```

GRstart/mgstart

int32 GRstart(int32 *file_id*)

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

Purpose Initializes the GR interface.

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

Description **GRstart** initializes the GR interface for the file specified by the parameter *file_id*.

This routine is used with the **GRend** routine to define the extent of the GR 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.

FORTRAN

```
integer function mgstart(file_id)

integer file_id
```

GRwritechunk/mgwchnk/mgwcchnk

intn GRwritechunk(int32 *ri_id*, int32 *origin*, const VOIDP *datap*)

ri_id	IN:	Raster image identifier returned by **GRcreate** or **GRselect**
origin	IN:	Origin of the chunk to be written
datap	IN:	Buffer for the chunk to be written

Purpose Writes a data chunk to a chunked raster image (pixel-interlace only)

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

GRwritechunk returns `FAIL` (or `-1`) when an attempt is made to use it to write to a non-chunked raster image.

Description **GRwritechunk** writes the entire chunk of data stored in the buffer *datap* to the chunked raster image identified by the parameter *ri_id*. Writing starts at the location specified by the parameter *origin*. This function has less overhead than **GRwriteimage** and should be used whenever an entire chunk of data is to be written.

The parameter *origin* is a two-dimensional array which specifies the coordinates of the chunk according to the chunk position in the overall chunk array. Refer to Chapter 8, ”General Raster Images (GR API),” in the *HDF User's Guide*.

The *datap* buffer contains the chunk's data organized in a pixel interlace mode.

FORTRAN

```
integer mgwchnk(ri_id, origin, datap)

integer ri_id, origin(2)

<valid_numeric_datatype> datap(*)


integer mgwcchnk(ri_id, origin, char_datap)

integer ri_id, origin(2)

character*(*) char_datap
```

GRwriteimage/mgwrimg/mgwcimg

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

ri_id	IN:	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

Purpose Writes a raster image.

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

Description **GRwriteimage** writes the subsample of the raster image data stored in the buffer *data* to the raster image specified by the parameter *ri_id*. The subsample is defined by the values of the parameters *start*, *stride* and *edge*.

The array *start* specifies the starting location of the subsample to be written. Valid values of each element in the array *start* are `0` to (the size of the corresponding raster image dimension `- 1`). The first element of the array *start* specifies an offset from the beginning of the array *data* along the fastest-changing dimension, which is the second dimension in C and the first dimension in Fortran. The second element of the array *start* specifies an offset from the beginning of the array *data* along the second fastest-changing dimension, which is the first dimension in C and the second dimension in Fortran. For example, if the first value of the array *start* is 2 and the second value is 3, the starting location of the subsample to be written is at the fourth row and third column in C, and at the third row and fourth column in Fortran.

The array *stride* specifies the writing pattern along each dimension. For example, if one of the elements of the array *stride* is `1`, then every element along the corresponding dimension of the array *data* will be written. If one of the elements of the *stride* array is `2`, then every other element along the corresponding dimension of the array *data* will be written, and so on. The correspondence between elements of the array *stride* and the dimensions of the array *data* is the same as described above for the array *start*.

Each element of the array *edges* specifies the number of data elements to be written along the corresponding dimension. The correspondence between the elements of the array *edges* and the dimensions of the array *data* is the same as described above for the array *start*.

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

Note

Regarding an important difference between the SD and GR interfaces: The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- o For **SDreaddata**, those parameters are expressed in (`y,x`) or [`row,column`] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- o For **GRreadimage**, those parameters are expressed in (`x,y`) or [`column,row`] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

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_mode*, int32 *num_entries*, VOIDP *pal_data*)

pal_id	IN:	Palette identifier returned by **GRgetlutid**
ncomp	IN:	Number of components in the palette
data_type	IN:	Data type of the palette data
interlace_mode	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 a palette.

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

Description **GRwritelut** writes a palette with the number of pixel components specified by the parameter *ncomp*, the data type of the palette data specified by the parameter *data_type*, the interlace mode specified by the parameter *interlace_mode*, and the number of entries in the palette specified by the parameter *num_entries*. The palette data itself is stored in the *pal_data* buffer. **Currently only "old-style" palettes are supported**, i.e *ncomp* = `3`, *num_entries* = `256`, *data_type* = `uint8`.

The parameter *ncomp* specifies the number of pixel components in the palette and must have a value of at least 1.

The parameter *data_type* specifies the type of the palette data and can be any of the data types supported by the HDF library. The data types supported by HDF are listed in Table 1A in Section I of this manual.

The parameter *interlace_mode* specifies the interlacing in which the palette is to be written. The valid values of *interlace_mode* are: `MFGR_INTERLACE_PIXEL` (or `0`), `MFGR_INTERLACE_LINE` (or `1`) and `MFGR_INTERLACE_COMPONENT` (or `2`.)

The buffer *pal_data* is assumed to have sufficient space allocated to store all of the palette data.

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

FORTRAN

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

integer pal_id, ncomp, data_type, interlace_mode, num_entries

<valid numeric data type> pal_data(*)
```

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

integer pal_id, ncomp, data_type, interlace_mode, num_entries

character*(*) pal_data
```

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
```

Hgetfileversion/hgfilver

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` (or `80`) characters as defined in `hfile.h`.

FORTRAN

```
integer function hgfilver(file_id, major_v, minor_v, release,
                    string)

integer file_id, major_v, minor_v, release

character*(*) string
```

Hgetlibversion/hglibver

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` (or `80`) characters as defined in `hfile.h`.

FORTRAN

```
integer function hglibver(major_v, minor_v, release, string)

integer major_v, minor_v, release

character*(*) string
```

Hishdf/hishdff

intn Hishdf(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 corrupted.

FORTRAN

```
integer function hishdff(filename)

character*(*) filename
```

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.

The number of data descriptors in a block, *n_dds*, is a non-negative integer with a default value of DEF_NDDS (or 16) and a minimum value of MIN_NDDS (or 4). If the specified value of *n_dds* is less than MIN_NDDS, then it will be set to MIN_NDDS.

HDF provides several access code definitions:

DFACC_CREATEIf file exists, delete it, then open a new file for read/write.
DFACC_READOpen for read only. If file does not exist, return an error.
DFACC_WRITEOpen 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_WRITE, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the specified 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.

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

FORTRAN

```
integer function hopen(filename, access, n_dds)

character*(*) filename

integer access, n_dds
```

HCget_config_info

intn HCget_config_info(comp_coder_t *coder_type*, uint32 **compression_config_info*)

coder_type	IN:	Type of compression
compression_config_info	OUT:	Flags indicating status of compression method

Purpose Retrieves information about the configuration of a compression method.

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

Description **HCget_config_info** retrieves the configuration status of the compression type specified by *coder_type*, returning that status information as flags in *compression_config_info*.

Valid values of *coder_type* are as follows:

`COMP_CODE_NONE`	no compression
`COMP_CODE_RLE`	RLE run-length encoding
`COMP_CODE_NBIT`	NBIT compression
`COMP_CODE_SKPHUFF`	Skipping Huffman compression
`COMP_CODE_DEFLATE`	GZIP compression
`COMP_CODE_SZIP`	Szip compression
`COMP_CODE_JPEG`	JPEG compression

The compression method, *coder_type*, used for a data set can be obtained as the returned value of the *comp_type* parameter in an **SDgetcompinfo** call.

The configuration flags returned in *compression_config_info* include the following:

`0`	Compression method is not enabled.
`COMP_DECODER_ENABLED`	Decoding is enabled.
`COMP_ENCODER_ENABLED`	Encoding is enabled.

If the returned value is `COMP_DECODER_ENABLED|COMP_ENCODER_ENABLED`, the compression method is enabled for both encoding and decoding.

In the general case, any available compression type can be configured in any mode:

```
COMP_DECODER_ENABLED
COMP_ENCODER_ENABLED
COMP_DECODER_ENABLED |COMP_ENCODER_ENABLED
```

As of this writing (HDF4 Release 2.1, February 2005), only the Szip compression library is actually used with the HDF libraries in more than one configuration (see immediately below.) As a third-party product, it is distributed in both decode-only and encode/decode configurations. All other compression methods are currently distributed or used in an encode/decode configuration if they are available at all, and **HCget_config_info** returns either `0` or `COMP_DECODER_ENABLED|COMP_ENCODER_ENABLED` when they are used.

Due to licensing requirements, the Szip library is available in both decode-only and encode/decode configurations. Therefore, the full range of values can be returned for Szip compression.

- **o** If the Szip version available on a system is decode-only, **HCget_config_info** will return `COMP_DECODER_ENABLED` in *compression_config_info*.
- **o** If the available Szip library is configured as encode/decode, *compression_config_info* will contain the value `COMP_DECODER_ENABLED|COMP_ENCODER_ENABLED` upon return.

Note

Regarding Szip compression in HDF4:
Szip compression is available only through the SD interface and is documented in the **SDsetcompress** and **SDgetcompinfo** reference manual entries. Aside from the configuration discovery capability documented in **HCget_config_info**, Szip compression is not accessible through the HC interface.

See also

Regarding Szip usage and licensing:
See `http://hdf.ncsa.uiuc.edu/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

Regarding compression in HDF4:
See the **SDsetcompress** and **SDgetcompinfo** entries in this reference manual for a more general description of dataset compression information.

FORTRAN

`currently unavailable`

HDdont_atexit/hddontatexit

intn HDdont_atexit(void)

Purpose Indicates to the library that an **atexit()** routine is **_not_** to be installed.

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

Description This routine indicates to the library that an **atexit()** cleanup routine should not be installed. The purpose for this is in situations where the library is dynamically linked into an application and is unlinked from the application before **exit()** gets called. In those situations, a routine installed with **atexit()** would jump to a routine which was no longer in memory, causing errors.

In order to be effective, this routine *must* be called before any other HDF function calls, and *must* be called each time the library is loaded/linked into the application (the first time and after it has been unloaded).

If this routine is used, certain memory buffers will not be deallocated, although in theory a user could call **HPend** on their own.

FORTRAN `integer hddontatexit( )`

HEprint/heprntf/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.

Fortran function returns `0` (zero) on success or `-1` on failure.

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 **heprnt** 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`.

The FORTRAN-77 routine **heprntf** is available on all platforms; **heprnt** is not supported on Microsoft Windows platforms.

The **heprntf** parameter *filename* is the name of the file to which error output is to be written. If the value of *filename* is an empty string (`''`), error output will be written to standard output, `stdout`.

FORTRAN

```
integer function heprntf(filename, level)

character*(*) filename

integer level


integer function heprnt(level)

integer level
```

HEstring/hestringf

const char *HEstring(hdf_err_code_t *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.

FORTRAN

```
integer function hestringf(error_code, error_message)

integer error_code

character*(*) error_message
```

HXsetcreatedir/hxiscdir

intn HXsetcreatedir(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 routines **HXcreate** or **SDsetexternal**), the HDF library accesses the external file just like the **open** call by default. Refer to the Reference Manual 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(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 *obj_id*, int32 *attr_index*, char **attr_name*, int32 **data_type*, int32 **count*)

obj_id	IN:	Identifier of the object to which the attribute is attached to
attr_index	IN:	Index of the attribute
attr_name	OUT:	Name of the attribute
data_type	OUT:	Data type of the attribute values
count	OUT:	Total number of values in the attribute

Purpose Retrieves information about an attribute.

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

Description **SDattrinfo** retrieves the name, data type, and number of values of the attribute specified by its index, *attr_index*, and stores them in the parameters *attr_name*, *data_type*, and *count*, respectively. This routine should be used before reading the values of an attribute with **SDreadattr**.

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

Valid values of the parameter *attr_index* range from `0` to the number of attributes attached to the object - 1.

Valid values of the parameter *data_type* can be found in Table 1A of Section I of this manual.

FORTRAN

```
integer function sfgainfo(obj_id, attr_index, attr_name,
                    data_type, count)

character*(*) attr_name

integer obj_id, attr_index, data_type, count
```

SDcheckempty/sfchempty

int32 SDcheckempty(int32 *sds_id*, intn **emptySDS*)

sds_id	IN:	SDS identifier
emptySDS	OUT:	Boolean value indicating whether the SDS is empty

Purpose Determines whether a scientific dataset (an SDS) is empty.

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

Description **SDcheckempty** sets the parameter *emptySDS* to `TRUE` if the dataset identified by *sds_id* has not been written with data, and to `FALSE`, otherwise.

The Fortran routine, **sfchempty**, returns `1` in *emptySDS* if the dataset is empty and `0` otherwise.

FORTRAN

```
integer function sfchempty(sds_id, emptySDS)

integer sds_id, emptySDS
```

SDcreate/sfcreate

int32 SDcreate(int32 *sd_id*, char **name*, int32 *data_type*, int32 *rank*, int32 *dimsizes*[])

sd_id	IN:	SD interface identifier returned by **SDstart**
name	IN:	Name of the data set
data_type	IN:	Data type for the values in the data set
rank	IN:	Number of the data set dimensions
dimsizes	IN:	Array containing the size of each dimension

Purpose Creates a new data set.

Return value Returns the data set identifier (*sds_id)* if successful and `FAIL` (or `-1`) otherwise.

Description **SDcreate** creates a data set with the name specified by the parameter *name*, the values of the data type specified by parameter *data_type*, the number of dimensions specified by the parameter *rank*, and the dimension sizes specified by the array *dimsizes*.

Once a data set has been created, it is not possible to change its name, data type, or rank. However, it is possible to create a data set and close the file before writing any data values to it. The values can be added or modified at a future time. To add data or modify an existing data set, use **SDselect** to get the data set identifier instead of **SDcreate**.

If the parameter *name* is `NULL` in C or an empty string in Fortran, the default name “Data Set” will be generated. The length of the name specified by the *name* parameter is no longer limited to 64 characters starting in HDF 4.2r2. Note that when an older version of the library reads a data set, which was created by a library of version 4.2r2 or later and has the name that is longer than 64 characters, the retrieved name will contain some garbage after 64 characters.

The calling program must ensure that the length of the *dimsizes* array is the value of the *rank* parameter, which is between `0` and `MAX_VAR_DIMS` (or `32`). Note that, in order for HDF4 and NetCDF models to work together, HDF allows SDS to have rank 0. However, there is no intention for data to be written to this type of SDS, but only to store attribute as part of the data description. Consequently, setting compression and setting chunk are disallowed.

To create a data set with an unlimited dimension, assign the value of `SD_UNLIMITED` (or `0`) to *dimsizes*[0] in C and to *dimsizes*(*rank*) in Fortran.

The *data_type* parameter can contain any data type supported by the HDF library. These data types are listed in Table 1A in Section I of this manual.

See the notes regarding the potential performance impact of unlimited dimension data sets in Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance" the *HDF User’s Guide*.

Note

Regarding an important difference between the SD and GR interfaces: The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- For **SDreaddata**, those parameters are expressed in (*y*,*x*) or [*row*,*column*] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For **GRreadimage**, those parameters are expressed in (*x*,*y*) or [*column*,*row*] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

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 **size*, int32 **data_type*, int32 **num_attrs*)

dim_id	IN:	Dimension identifier returned by **SDgetdimid**
name	OUT:	Name of the dimension
size	OUT:	Size of the dimension
data_type	OUT:	Data type of the dimension scale
num_attrs	OUT:	Number of attributes assigned to the dimension

Purpose Retrieves information about a dimension.

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

Description **SDdiminfo** retrieves the name, size, data type, and number of values of the dimension specified by the parameter *dim_id*, and stores them in the parameters *name*, *size*, *data_type*, and *num_attrs*, respectively.

If the output value of the parameter *size* is set to `0`, then the dimension specified by the *dim_id* parameter is unlimited. To get the number of records of an unlimited dimension, use **SDgetinfo**.

If scale information has been stored for this dimension via **SDsetdimscale**, the *data_type* parameter will contain the data type. Valid data types can be found in Table 1A of Section I of this manual. If no scale information has been stored for this dimension, the value returned in the *data_type* parameter will be `0`.

If the user has not named the dimension via **SDsetdimname**, a default dimension name of “`fakeDim[x]`” will be generated by the library, where `[x]` denotes the dimension index. If the name is not desired, the parameter *name* can be set to `NULL` in C and an empty string in Fortran.

FORTRAN

```
integer function sfgdinfo(dim_id, name, size, data_type,
                  num_attrs)

character*(*) name

integer dim_id, size, data_type, num_attrs
```

SDend/sfend

intn SDend(int32 *sd_id*)

sd_id	IN:	SD interface identifier returned by **SDstart**

Purpose Terminates access to an SD interface.

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

Description **SDend** closes the file and frees memory allocated by the library when SD interface activities are completed. If the calling program exits without invoking this routine, recent changes made to the in-core file data are likely not to be flushed to the file. Note that each **SDstart** must have a matching **SDend**.

FORTRAN

```
integer function sfend(sd_id)

integer sd_id
```

SDendaccess/sfendacc

intn SDendaccess(int32 *sds_id*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**

Purpose Terminates access to a data set.

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

Description **SDendaccess** frees the memory taken up by the HDF library's data structures devoted to the data set identified by the parameter *sds_id*.

Failing to call this routine after all operations on the specified data set are complete may result in loss of data. This routine must be called once for each call to **SDcreate** or **SDselect**.

FORTRAN

```
integer function sfendacc(sds_id)

integer sds_id
```

SDfileinfo/sffinfo

intn SDfileinfo(int32 *sd_id*, int32 **num_datasets*, int32 **num_global_attrs*)

sd_id	IN:	SD interface identifier returned by **SDstart**
num_datasets	OUT:	Number of data sets in the file
num_global_attrs	OUT:	Number of global attributes in the file

Purpose Retrieves the number of data sets and the number of global attributes in a file.

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

Description **SDfileinfo** returns the number of data sets in the parameter *num_datasets* and the number of global attributes in the parameter *num_global_attrs*. The term “global attributes” refers to attributes that are assigned to the file. The global attributes are created by **SDsetattr** using an SD interface identifier (*sd_id*) rather than a data set identifier (*sds_id*).

The value returned by the parameter *num_datasets* includes the number of coordinate variable data sets. To determine if the data set is a coordinate variable, use **SDiscoordvar**.

FORTRAN

```
integer function sffinfo(sd_id, num_datasets, num_global_attrs)

integer sd_id, num_datasets, num_global_attrs
```

SDfindattr/sffattr

int32 SDfindattr(int32 *obj_id*, char **attr_name*)

obj_id	IN:	Identifier of the object to which the attribute is attached
attr_name	IN:	Name of the attribute

Purpose Finds the index of an attribute given its name.

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

Description **SDfindattr** retrieves the index of the object's attribute with the name specified by the parameter *attr_name*.

The attribute is attached to the object specified by the parameter *obj_id*. The parameter *obj_id* can be either an SD interface identifier (*sd_id*), returned by **SDstart**, a data set identifier (*sds_id*), returned by **SDselect**, or a dimension identifier (*dim_id*), returned by **SDgetdimid**.

Wildcard characters are not allowed in the parameter *attr_name*. **SDfindattr** searches for the name specified in the parameter *attr_name* in a case-sensitive manner.

FORTRAN

```
integer function sffattr(obj_id, attr_name)

integer obj_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:	Data set identifier returned by **SDcreate** or **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 a data set.

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

Description **SDgetcal** reads the calibration record attached to the data set identified by the parameter *sds_id*. A calibration record is comprised of four 64-bit floating point values followed by a 32-bit integer. The information is listed in the following table:

cal	calibration factor
cal_err	calibration error
offset	uncalibrated offset
offset_err	uncalibrated offset error
data_type	data type of the uncalibrated data

The relationship between a calibrated value `cal_value` and the original value *orig_value* is defined as *orig_value* = *cal* * (*cal_value* - *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/sfgichnk

intn SDgetchunkinfo(int32 *sds_id*, HDF_CHUNK_DEF **cdef*, int32 **flag*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
C only:		
cdef	OUT:	Pointer to the chunk definition
flag	OUT:	Compression flag
Fortran only:		
dim_length	OUT:	Array of chunk dimensions
flag	OUT:	Compression flag

Purpose Retrieves chunking information for a data set.

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

Description **SDgetchunkinfo** retrieves chunking information about the data set identified by the parameter *sds_id* into the parameters *cdef* and *flag* in C, and to the parameters *dim_length* and *flag* in Fortran.

Currently, only information about chunk dimensions is retrieved into the corresponding *cdef* structure element for each type of compression in C, and in the *dim_length* array in Fortran. No information on compression parameters is available in the `comp` structure of the `HDF_CHUNK_DEF` union. Refer to the page on **SDsetchunk** in this manual for specific information on the `HDF_CHUNK_DEF` union.

If the data set is empty, **SDgetchunkinfo** will fail. Thus, application must first verify that the data set has been written with data, before calling **SDgetchunkinfo**. **SDcheckempty** determines whether the data set is empty.

The value returned in the *flag* parameter indicates the data set type (i.e., if the data set is not chunked, chunked, and chunked and compressed).

If the chunk length for each dimension is not needed, `NULL` can be passed in as the value of the *cdef* parameter in C.

The following table shows the type of the data set, possible values of the *flag* parameter, and the corresponding *cdef* structure element filled with the chunk's dimensions.

Type of Data Set	Values of *flag* in C	Values of *flag* in Fortran	*cdef* Structure Element Filled with the Chunk's Dimensions
Not chunked	`HDF_NONE`	`-1`	None
Chunked	`HDF_CHUNK`	`0`	cdef.chunk_lengths[]

Type of Data Set	**Values of *flag* in C**	**Values of *flag* in Fortran**	***cdef* Structure Element Filled with the Chunk's Dimensions**
Chunked and compressed with either the run-length encoding (RLE), Skipping Huffman, GZIP, or Szip compression algorithms	HDF_CHUNK | HDF_COMP	1	cdef.comp.chunk_lengths[ ]
Chunked and compressed with NBIT compression	HDF_CHUNK | HDF_NBIT	2	cdef.nbit.chunk_lengths[]

FORTRAN

```
integer function sfgichnk(sds_id, dim_length, flag)

integer sds_id, dim_length(*), flag
```

SDgetcompinfo/sfgcompress

intn SDgetcompinfo(int32 *sds_id*, comp_coder_t **comp_type*, comp_info **c_info*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
comp_type	OUT:	Type of compression
c_info	OUT:	Pointer to compression information structure

Purpose Retrieves data set compression type and compression information.

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

Description **SDgetcompinfo** retrieves the compression type and compression information for a data set, when the data is either compressed, chunked or chunked and compressed. **SDgetcompinfo** replaces **SDgetcompress** because this function has flaws, causing failure for some chunked and chunked/compressed data.

The compression method is returned in the parameter *comp_type*. Valid values of *comp_type* are as follows:

`COMP_CODE_NONE` for no compression
`COMP_CODE_RLE` for RLE run-length encoding
`COMP_CODE_NBIT` for NBIT compression
`COMP_CODE_SKPHUFF` for Skipping Huffman compression
`COMP_CODE_DEFLATE` for GZIP compression
`COMP_CODE_SZIP` for Szip compression

Additional compression method parameters are returned in the *c_info* struct in C and the array parameter *comp_prm* in Fortran. Note that *c_info* and *comp_prm* come into place only with compression modes that require additional parameters (i.e., other than *comp_type*); they are ignored in other cases.

The *c_info* struct is of type `comp_info`, contains algorithm-specific information for the library compression routines, and is described in the **SDsetcompress** entry in this reference manual and in the `hcomp.h` header file.

The *comp_prm* parameter is an array returning one or more parameters, as required by the compression method in use. Each compression parameter is returned as an element of the array, as follows:

- With Skipping Huffman compression, *comp_prm* is a 1-element array and *comp_prm(1)* contains the skip value, `skphuff_skp_size`.
- In the case of GZIP compression, *comp_prm* is also a 1-element array and *comp_prm(1)* contains the deflation value, `deflate_value`.
- In the case of NBIT compression, *comp_prm* is a 4-element array with `sign_ext` in *comp_prm(1)*, `fill_one` in *comp_prm(2)*, `start_bit` in *comp_prm(3)*, and `bit_len` in *comp_prm(4)*. The fields `sign_ext`, `fill_one`, `start_bit`, and `bit_len` are discussed in the **SDsetnbitdataset/sfsnbit** entry of this reference manual.
- In the case of Szip compression, *comp_prm* is a 5-element array with `option_mask` in *comp_prm(1)*, `pixels_per_block` in *comp_prm(2)*, `pixels_per_scanline` in *comp_prm(3)*, `bits_per_pixel` in *comp_prm(4)*, and `pixels` in *comp_prm (5)*.

In the general case, any available compression type can be configured in any mode:

`COMP_DECODER_ENABLED` Decode data only
`COMP_ENCODER_ENABLED` Encode data only
`COMP_DECODER_ENABLED |COMP_ENCODER_ENABLED`
Decode and encode data

As of this writing (HDF4 Release 2.1, February 2005), only the Szip compression library is actually used with the HDF libraries in more than one configuration (see immediately below). As a third-party product, it is distributed in both decode-only and encode/decode configurations. All other compression methods are currently distributed or used in an encode/decode configuration if they are available at all. See also **HCget_config_info**.

SDgetcompinfo will succeed for an Szip-compressed dataset whether the available Szip library is configured either for encoding/decoding or for decoding-only.

If the Szip configuration is decode-only, i.e., an **HCget_config_info** call on the dataset will return only `COMP_DECODER_ENABLED` in *compression_config_info*. Note that in such a case the file must be opened in read-only mode, i.e. with **SDstart**(*filename*, `DFACC_RDONLY`).

If the Szip configuration is encode/decode, i.e., an **HCget_config_info** call on the dataset will return `COMP_ENCODER_ENABLED|COMP_DECODER_ENABLED` in *compression_config_info*. In this case, the file and dataset can be opened in read/write mode.

Note **Regarding uncompressed data or an empty data set:**
SDgetcompinfo will succeed and the parameter *comp_type* will have the value `COMP_CODE_NONE` if either of the following conditions exists:

- The data set is not compressed.
- No data has been written to the SDS.

Note **Regarding Szip usage and licensing:**
See `http://www.hdfgroup.org/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

FORTRAN

```
integer function sfgcompress(sds_id, comp_type, comp_prm)

integer sds_id, comp_type, comp_prm(*)
```

SDgetdatasize/

intn SDgetdatasize(int32 *sds_id*, int32 **comp_size*, int32 **orig_size*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
comp_size	OUT:	Size of compressed data
orig_size	OUT:	Size of original data

Purpose Retrieves the sizes of original and compressed data.

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

Description **SDgetdatastrs** retrieves the original and compressed data sizes, in bytes, of the data set specified by the parameter *sds_id*. `NULL` can be passed in for undesired parameter.

The values of comp_size and orig_size will be returned as described below:

Type of data	*comp_size*	*orig_size*
no data	0	0
compressed	size of compressed data	size of original data
chunked/compressed	size of compressed data in all chunks	size of original data in all chunks
uncompressed	size of original data	size of original data
unlimited dimension	size of original data	size of original data

FORTRAN `Currently unavailable`

SDgetdatastrs/sfgdtstr

intn SDgetdatastrs(int32 *sds_id*, char **label*, char **unit*, char **format*, char **coordsys*, intn *length*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
label	OUT:	Label (predefined attribute)
unit	OUT:	Unit (predefined attribute)
format	OUT:	Format (predefined attribute)
coordsys	OUT:	Coordinate system (predefined attribute)
length	IN:	Maximum length of the above predefined attributes

Purpose Retrieves the predefined attributes of a data set.

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

Description **SDgetdatastrs** retrieves the predefined attributes for the data set specified by the parameter *sds_id*. The predefined attributes are label, unit, format, and coordinate system. They are then stored in the parameters *label*, *unit*, *format*, and *coordsys*, respectively. Refer to Section 3.10 of the *HDF User's Guide* for more information on predefined attributes.

If a particular data string is not stored, the first character of the corresponding **SDgetdatastrs** parameter is '`\0`' in C. In FORTRAN, the parameter contains an empty string. Each string buffer must include the space to hold the null termination character. In C, if a user does not want a string back, `NULL` can be passed in for that string. Data strings are set by the **SDsetdatastrs** routine.

FORTRAN

```
integer function sfgdtstr(sds_id, label, unit, format, coordsys,
                    length)

integer sds_id, length

character*(*) label, unit, format, coordsys
```

SDgetdimid/sfdimid

int32 SDgetdimid(int32 *sds_id*, intn *dim_index*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
dim_index	IN:	Index of the dimension

Purpose Returns the identifier of a dimension given its index.

Return value Returns the dimension identifier (*dim_id*) if successful and `FAIL` (or `-1`) otherwise.

Description **SDgetdimid** returns the identifier of the dimension specified by its index, the parameter *dim_index*.

The dimension index is a nonnegative integer and is less than the total number of data set dimensions returned by **SDgetinfo**.

FORTRAN

```
integer function sfdimid(sds_id, dim_index)

integer sds_id, dim_index
```

SDgetdimscale/sfgdscale

intn SDgetdimscale(int32 *dim_id*, VOIDP *scale_buf*)

dim_id	IN:	Dimension identifier returned by **SDgetdimid**
scale_buf	OUT:	Buffer for the scale values

Purpose Retrieves the scale values for a dimension.

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

Description **SDgetdimscale** retrieves the scale values of the dimension identified by the parameter *dim_id* and stores the values in the buffer *scale_buf*.

SDdiminfo should be used to determine whether a scale has been set for the dimension, i.e., that the dimension scale data type is a valid HDF data type (not `0`). Also use **SDdiminfo** to obtain the number of scale values for space allocation before calling **SDgetdimscale**.

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

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

FORTRAN

```
integer function sfgdscale(dim_id, scale_buf)

integer dim_id

<valid numeric data type> scale_buf(*)
```

SDgetdimstrs/sfgdmstr

intn SDgetdimstrs(int32 *dim_id*, char **label*, char **unit*, char **format*, intn *length*)

dim_id	IN:	Dimension identifier returned by **SDgetdimid**
label	OUT:	Label (predefined attribute)
unit	OUT:	Unit (predefined attribute)
format	OUT:	Format (predefined attribute)
length	IN:	Maximum length of the above predefined attributes

Purpose Retrieves the predefined attributes of a dimension.

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

Description **SDgetdimstrs** retrieves the predefined attributes associated with the dimension identified by the parameter *dim_id*. The predefined attributes are label, unit, and format. These predefined attributes are stored in the parameters *label*, *unit*, and *format*, respectively. Refer to Section 3.10 of the *HDF User's Guide* for more information on predefined attributes.

If a particular data string was not stored, the first character of the corresponding **SDgetdimstrs** parameter is '`\0`'. Each string buffer must include space for the null termination character. If a user does not want a string returned, the corresponding parameter can be set to `NULL` in C and an empty string in Fortran. The predefined attributes are set by **SDsetdimstrs**.

FORTRAN

```
integer function sfgdmstr(dim_id, label, unit, format, length)

integer dim_id, length

character*(*) label, unit, format
```

SDgetfilename/sfgetfname

intn SDgetfilename(int32 *file_id*, char **filename*)

file_id	IN:	A file identifier
filename	OUT:	Name of the file

Purpose Given a file identifier, retrieves the name of the file.

Return value Returns the length of the file name, without '`\0`', on success, and `FAIL`, otherwise.

FORTRAN

```
integer function sfgetfname(file_id, filename)

character*(*) filename

integer file_id
```

SDgetfillvalue/sfgfill/sfgcfill

intn SDgetfillvalue(int32 *sds_id*, VOIDP *fill_value*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
fill_value	OUT:	Buffer for the returned fill value

Purpose Reads the fill value of a data set, if the value has been set.

Return value Returns `SUCCEED` (or `0`) if a fill value is retrieved and `FAIL` (or `-1`) otherwise, including when the fill value is not set.

Description **SDgetfillvalue** reads the fill value which has been set for the data set specified by the parameter *sds_id*. It is assumed that the data type of the fill value is the same as that of the data set.

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_value)

integer sds_id

<valid numeric data type> fill_value


integer function sfgcfill(sds_id, fill_value)

integer sds_id

character*(*) fill_value
```

SDgetinfo/sfginfo

intn SDgetinfo(int32 *sds_id*, char **sds_name*, int32 **rank*, int32 *dimsizes*[], int32 **data_type*, int32 **num_attrs*)

sds_id	IN:	Data set identifier returned by **SDcreate** and **SDselect**
sds_name	OUT:	Name of the data set
rank	OUT:	Number of dimensions in the data set
dimsizes	OUT:	Array containing the size of each dimension in the data set
data_type	OUT:	Data type for the data stored in the data set
num_attrs	OUT:	Number of attributes for the data set

Purpose Retrieves the name, rank, dimension sizes, data type and number of attributes for a data set.

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

Description **SDgetinfo** retrieves the name, number of dimensions, sizes of dimensions, data type, and number of attributes of the data set identified by *sds_id*, and stores them in the parameters *sds_name*, *rank*, *dimsizes*, *data_type*, and *num_attrs*, respectively.

The buffer *sds_name* can have at most `64` characters. If the name of the data set is not desired, then the parameter *sds_name* can be set to `NULL` in C and an empty string in Fortran.

The maximum value of the *rank* parameter is `MAX_VAR_DIMS` (or `32`).

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

FORTRAN

```
integer function sfginfo(sds_id, sds_name, rank, dimsizes,
                    data_type, num_attrs)

character*(*) sds_name

integer sds_id, rank, dimsizes(*)

integer data_type, num_attrs
```

SDgetnamelen/sfgetnamelen

intn SDgetnamelen(int32 *obj_id*, uint16 *name_len*)

obj_id	IN:	Identifier of the object
name_len	OUT:	Length of the object's name

Purpose Retrieves the length of the name of a file, a dataset, or a dimension.

Return value Returns the length of the object's name on success, and `FAIL` (`-1`) otherwise.

Description Given an identifier of a file, a dataset, or a dimension, **SDgetnamelen** retrieves the length of its name into *name_len*. The length does not include the character '`\0`'.

FORTRAN

```
integer function sfgetnamelen(obj_id, length)

integer obj_id, length
```

SDgetnumvars_byname/sfgnvars_byname

intn SDgetnumvars_byname(int32 *sd_id*, char **sds_name, int32 *n_vars*)

sd_id	IN:	SD interface identifier returned by **SDstart**
sds_name	IN:	Name of the data set
n_vars	OUT:	Number of variables named *sds_name*

Purpose Get the number of data sets having the same name.

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

Description **SDgetnumvars_byname** retrieves the number of variables with the name specified by the parameter *sds_name*. The variables may include both data sets or coordinate variables. The routine does not accept wildcards in the specified data set name. It also searches on that name in a case-sensitive manner.

FORTRAN

```
integer function sfgnvars_byname(sd_id, sds_name, n_vars)

integer sd_id, n_vars

character*(*) sds_name
```

SDgetrange/sfgrange

intn SDgetrange(int32 *sds_id*, VOIDP *max*, VOIDP *min*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
max	OUT:	Maximum value of the range
min	OUT:	Minimum value of the range

Purpose Retrieves the maximum and minimum values of the range.

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

Description **SDgetrange** retrieves the maximum value of the range into the parameter *max* and the minimum value into the parameter *min*. The maximum and minimum values must be previously set via a call to **SDsetrange**.

It is assumed that the data type for the maximum and minimum range values are the same as that of the data.

FORTRAN

```
integer function sfgrange(sds_id, max, min)

integer sds_id

<valid numeric data type> max, min
```

SDget_maxopenfiles/sfgmaxopenf

intn SDget_maxopenfiles(intn **curr_max*, intn **sys_limit*)

cu	IN:	Data set identifier returned by **SDcreate** or **SDselect**
curr_max	OUT:	Current number of open files
sys_limit	OUT:	Maximum number of open files

Purpose Retrieves current and maximum number of open files.

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

Description **SDget_maxopenfiles** retrieves the current number of open files allowed in HDF, *curr_max*, and the maxinum number of open files allowed on the system, *sys_limit*. If either of the values is not desired, then `NULL` can be passed in.

FORTRAN

```
integer function sfgmaxopenf(cur_max, sys_limit)

integer cur_max, sys_limit
```

SDget_numopenfiles/sfgnumopenf

intn SDget_numopenfiles()

Purpose Returns the number of files currently being opened.

Return value Returns the number of files currently being opened.

FORTRAN

```
integer function sfgnumopenf(cur_num)

integer cur_num
```

SDidtoref/sfid2ref

int32 SDidtoref(int32 *sds_id*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**

Purpose Returns the reference number assigned to a data set.

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

Description **SDidtoref** returns the reference number of the data set specified by the parameter *sds_id*. The reference number is assigned by the HDF library when the data set is created. The specified reference number can be used to add the data set to a vgroup as well as a means of using the HDF annotations interface to annotate the data set.

FORTRAN

```
integer function sfid2ref(sds_id)

integer sds_id
```

SDidtype/sfidtype

hdf_idtype_t SDidtype(int32 *obj_id*)

obj_id	IN:	Identifier of the object

Purpose Given an id, return the type of object the id represents.

Return value Returns a value of type `hdf_idtype_t`.

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

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

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

FORTRAN

```
integer function sfidtype(obj_id, obj_type)

integer obj_id, obj_type
```

SDiscoordvar/sfiscvar

intn SDiscoordvar(int32 *sds_id*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**

Purpose Determines if a data set is a coordinate variable.

Return value Returns `TRUE` (or `1`) if the data set is a coordinate variable, and `FALSE` (or `0`) otherwise.

Description **SDiscoordvar** determines if the data set specified by the parameter *sds_id* is a coordinate variable.

Coordinate variables are created to store metadata associated with dimensions. To ensure compatibility with netCDF, coordinate variables are implemented as data sets.

Note that if there are more than one variable (either multiple data sets or data sets and dimensions) with the same name in the file, **SDiscoordvar** might return the incorrect item. This was a bug in the HDF4 library prior to 4.2r2. For details, please refer to the note "Regarding naming a dimension the same as an SDS' name" in the **SDsetdimname** page.

When the uniqueness of the name is unknown, **SDgetnumvars_byname** can be used to get the number of data sets (or variables, which includes both data sets and coordinate variables) with the same name. **SDnametoindices** can then be used to get a list of structures containing the indices and the types of all the variables of that same name. However, note that the type of the variable will be `UNKNOWN` when the data was created by libraries of pre-4.2r2.

FORTRAN

```
integer function sfiscvar(sds_id)

integer sds_id
```

SDisdimval_bwcomp/sfisdmvc

intn SDisdimval_bwcomp(int32 *dim_id*)

dim_id	IN:	Dimension identifier returned by **SDgetdimid**

Purpose Determines whether a dimension has the old and new representations or the new representation only.

Refer to Chapter 3, "Scientific Data Sets (SD API)" of the *HDF User's Guide*, for information on old and new dimension representations.

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 **SDisdimval_bwcomp** will flag the dimension specified by the parameter *dim_id* as backward-compatible if a vdata with a class name of "DimVal0.0" does not exist in the vgroup for that dimension. If the vdata does exist, the specified dimension will be identified by **SDisdimval_bcomp** 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/sfisrcrd

int32 SDisrecord(int32 *sds_id*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**

Purpose Determines whether a data set is appendable.

Return value Returns `TRUE` (or `1`) if the data set is appendable, and `FALSE` (or `0`) otherwise.

Description **SDisrecord** will determine if the data set specified by the parameter *sds_id* is appendable, which means that the slowest-changing dimension was declared unlimited when the data set was created.

FORTRAN

```
integer sfisrcrd(sd_id)

integer sd_id
```

SDnametoindex/sfn2index

int32 SDnametoindex(int32 *sd_id*, char **sds_name*)

sd_id	IN:	SD interface identifier returned by **SDstart**
sds_name	IN:	Name of the data set

Purpose Determines the index of a data set given its name.

Return value Returns the index of the data set (*sds_index*) if the data set is found and `FAIL` (or `-1`) otherwise.

Description **SDnametoindex** returns the index of the data set with the name specified by the parameter *sds_name*. The routine does not accept wildcards in the specified data set name. It also searches on that name in a case-sensitive manner. If there are more than one data set with the same name, the routine will return the index of the first one.

Note that if there are more than one data set with the same name in the file, writing to a data set returned by this function without verifying that it is the desired data set could cause data corruption. For details, please refer to the note "Regarding naming a dimension the same as an SDS' name" in the **SDsetdimname** page.

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

FORTRAN

```
integer function sfn2index(sd_id, sds_name)

integer sd_id

character*(*) sds_name
```

SDnametoindices/sfn2indices

intn SDnametoindices(int32 *sd_id*, char **sds_name*, hdf_varlist_t* *var_list*)

sd_id	IN:	SD interface identifier returned by **SDstart**
sds_name	IN:	Name of the data set
var_list	OUT:	List of all variables of same name

Purpose Retrieves indices of all variables with the same name.

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

Description **SDnametoindices** retrieves a list of structures `hdf_varlist_t`, containing the indices and the types of all variables of the same name *sds_name*.

The structure `hdf_varlist_t` is defined as:

```
typedef struct varlist
{
    int32 var_index;  /* index of a variable */
    vartype_t var_type; /* type of a variable */
} hdf_varlist_t;
```

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

```
  IS_SDSVAR=0 : variable is an actual SDS
  IS_CRDVAR=1 : variable is a coordinate variable
 UNKNOWN=2 : variable is created before HDF4.2r2, unknown type
```

The routine does not accept wildcards in the specified data set name. It also searches on that name in a case-sensitive manner.

Note that if there are more than one data set with the same name in the file, writing to a data set returned by this function without verifying that it is the desired data set could cause data corruption. For details, please refer to the note "Regarding naming a dimension the same as an SDS' name" in the **SDsetdimname** page.

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

FORTRAN

```
integer function sfn2indices(sd_id, sds_name, var_list, type_list,
                  n_vars)

integer sd_id

character*(*) sds_name

integer var_list(*), type_list(*)

integer n_vars
```

SDreadattr/sfrnatt/sfrcatt

intn SDreadattr(int32 *obj_id*, int32 *attr_index*, VOIDP *attr_buf*)

obj_id	IN:	Identifier of the object the attribute is attached to
attr_index	IN:	Index of the attribute to be read
attr_buf	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 **SDreadattr** reads the values of the attribute specified by the parameter *attr_index* and stores the values in the buffer *attr_buf*. It is assumed that the user has called **SDattrinfo** to retrieve the number of attribute values and allocate sufficient space for the buffer. Note that the routine does not read a subset of attribute values.

The value of *obj_id* can be either an SD interface identifier (*sd_id*), returned by **SDstart**, a data set identifier (*sds_id*), returned by **SDselect**, or a dimension identifier (*dim_id*), returned by **SDgetdimid**.

The value of *attr_index* is a positive integer and is less than the total number of attributes. The index value can be obtained using the routines **SDnametoindex** and **SDreftoindex**. The total number of attributes for the object can be obtained using the routines **SDgetinfo**, **SDattrinfo**, **SDdiminfo** and **SDfileinfo**.

Note that this routine returns an array of characters, not a standard null-terminated string. If an application is running in an environment where a null-terminated string is expected, the application must add the null character before saving the string or using it further.

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.

FORTRAN

```
integer function sfrnatt(obj_id, attr_index, attr_buffer)

integer obj_id, attr_index

<valid numeric data> attr_buffer(*)


integer function sfrcatt(obj_id, attr_index, attr_buffer)

integer obj_id, attr_index

character*(*) attr_buffer
```

SDreadchunk/sfrchnk/sfrcchnk

intn SDreadchunk(int32 *sds_id*, int32 **origin*, VOIDP *datap*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
origin	IN:	Origin of the chunk to be read
datap	OUT:	Buffer for the chunk to be read

Purpose Reads a data chunk from a chunked data set.

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

Description **SDreadchunk** reads the entire chunk of data from the chunked data set identified by the parameter *sds_id*, and stores the data in the buffer *datap*. Reading starts at the location specified by the parameter *origin*. **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 data set.

The parameter *origin* specifies the coordinates of the chunk according to the chunk position in the chunked array. Refer to the Chapter 3, "Scientific Data Sets (SD API)" of the *HDF User's Guide*, for a description of the organization of chunks in a data set.

SDreadchunk will return `FAIL` (or `-1`) when an attempt is made to read from a non-chunked data set.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (**sfrchnk**) and one for character data (**sfrcchnk**).

FORTRAN

```
integer sfrchnk(sds_id, origin, datap)

integer sds_id, origin(*)

<valid numeric data type> datap(*)


integer sfrcchnk(sds_id, origin, datap)

integer sds_id, origin(*)

character*(*) datap(*)
```

SDreaddata/sfrdata/sfrcdata

intn SDreaddata(int32 *sds_id*, int32 *start*[], int32 *stride*[], int32 *edge*[], VOIDP *buffer*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
start	IN:	Array specifying the starting location from where data is read
stride	IN:	Array specifying the interval between the values that will be read along each dimension
edge	IN:	Array specifying the number of values to be read along each dimension
buffer	OUT:	Buffer to store the data read

Purpose Reads a subsample of data from a data set or coordinate variable.

Return value Returns `SUCCEED` (or `0`) if successful or if the data set or coordinate variable contains no data and `FAIL` (or `-1`) otherwise.

Description **SDreaddata** reads the specified subsample of data from the data set or coordinate variable identified by the parameter *sds_id*. The read data is stored in the buffer *buffer.* The subsample is defined by the parameters *start*, *stride* and *edge*.

The array *start* specifies the starting position from where the subsample will be read. Valid values of each element in the array *start* are from 0 to the size of the corresponding dimension of the data set - 1. The dimension sizes are returned by **SDgetinfo**.

The array *edge* specifies the number of values to read along each data set dimension.

The array *stride* specifies the reading pattern along each dimension. For example, if one of the elements of the array *stride* is 1, then every element along the corresponding dimension of the data set will be read. If one of the elements of the array *stride* is 2, then every other element along the corresponding dimension of the data set will be read, and so on. Specifying *stride* value of `NULL` in the C interface or setting all values of the array *stride* to 1 in either interface specifies the contiguous reading of data. If all values in the array *stride* are set to `0` or any value causes striding beyond the end of the associate dimension, **SDreaddata** returns `FAIL` (or `-1`). No matter what stride value is provided, data is always placed contiguously in the buffer.

When reading data from a “chunked” data set using **SDreaddata**, consideration should be given to the issues presented in the section on chunking in Chapter 3, "Scientific Data Sets (SD API)" and Chapter 14, "HDF Performance Issues" in the *HDF User’s Guide*.

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.

Note **Regarding an important difference between the SD and GR interfaces:**
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- For **SDreaddata**, those parameters are expressed in (*y*,*x*) or [*row*,*column*] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For **GRreadimage**, those parameters are expressed in (*x*,*y*) or [*column*,*row*] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

It is sometimes necessary to determine whether and how a dataset is compressed and whether the software necessary to read that data is available. The compression method used on the dataset can be determined with **SDgetcompinfo** and the availability and configuration of the compression software with **HCget_config_info**. Further information is available in the respective entries in this reference manual.

Note **Regarding Szip-compressed data:**
SDreaddata can succeed for an Szip-compressed dataset whether the available Szip library is configured either for encoding/decoding or for decoding-only.

If the available Szip configuration is decode-only, **HCget_config_info** will return only `COMP_DECODER_ENABLED` in *compression_config_info*; the returned flags will not include `COMP_ENCODER_ENABLED`. In such a case, the file must have been opened in read-only mode, i.e. with **SDstart**(*filename*, `DFACC_RDONLY`).

If the Szip avaialble configuration is encode/decode, **HCget_config_info** will return `COMP_ENCODER_ENABLED|COMP_DECODER_ENABLED`. In such a case, the file and dataset can be opened in read/write mode.

See the **HCget_config_info** and **SDgetcompinfo** entries in this reference manual for further information.

Note **Regarding Szip usage and licensing:**
See `http://www.hdfgroup.org/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

FORTRAN

```
integer function sfrdata(sds_id, start, stride, edge, buffer)

integer sds_id, start(*), stride(*), edge(*)

<valid numeric data type> 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 by **SDstart**
sds_ref	IN:	Reference number of the data set

Purpose Returns the index of a data set given the reference number.

Return value Returns the index of the data set (*sds_index*) if the data set is found and `FAIL` (or `-1`) otherwise.

Description **SDreftoindex** returns the index of a data set identified by its reference number, *sds_ref*.

The value of *sds_index* returned by **SDreftoindex** can be passed to **SDselect** to obtain a data set identifier (*sds_id*).

FORTRAN

```
integer function sfref2index(sd_id, sds_ref)

integer sd_id, sds_ref
```

SDreset_maxopenfiles/sfrmaxopenf

intn SDreset_maxopenfiles(intn req_max)

req_max	IN:	Requested maximum number of opened files allowed

Purpose Resets the maximum number of files can be opened at the same time.

Return value Returns the current maximum number of opened files allowed if successful and `FAIL` (or -1) otherwise.

Description Prior to release 4.2r2, the maximum number of files that can be opened at the same time was limited to 32. In HDF 4.2r2 and later versions, if this limit is reached, the library will increase it to the system limit minus 3 to account for stdin, stdout, and stderr.

This function can be called anytime to change the maximum number of open files allowed in HDF to *req_max*. If *req_max* is `0`, **SDreset_maxopenfiles** will simply return the current maximum number of open files allowed. If *req_max* exceeds system limit, **SDreset_maxopenfiles** will reset the maximum number of open files to the system limit, and return that value.

Furthermore, if the system maximum limit is reached, the library will push the error code `DFE_TOOMANY` onto the error stack. User applications can detect this after an **SDstart** fails.

FORTRAN

```
integer function sfrmaxopenf(req_max)

integer req_max
```

SDselect/sfselect

int32 SDselect(int32 *sd_id*, int32 *sds_index*)

sd_id	IN:	SD interface identifier returned by **SDstart**
sds_index	IN:	Index of the data set

Purpose Obtains the data set identifier (*sds_id*) of a data set.

Return value Returns the data set identifier (*sds_id*) if successful and `FAIL` (or `-1`) otherwise.

Description **SDselect** obtains the data set identifier (*sds_id*) of the data set specified by its index, *sds_index*.

The integration with netCDF has required that a dimension (or coordinate variable) is stored as a data set in the file. Therefore, the value of *sds_index* may correspond to the coordinate variable instead of the actual data set. Users should use the routine **SDiscoordvar** to determine whether the given data set is a coordinate variable.

The value of *sds_index* is greater than or equal to 0 and less than the number of data sets in the file. The total number of data sets in a file may be obtained from a call to **SDfileinfo**. The **SDnametoindex** routine can be used to find the index of a data set if its name is known.

FORTRAN

```
integer function sfselect(sd_id, sds_index)

integer sd_id, sds_index
```

SDsetattr/sfsnatt/sfscatt

intn SDsetattr(int32 *obj_id*, char **attr_name*, int32 *data_type*, int32 *count*, VOIDP *values*)

obj_id	IN:	Identifier of the object the attribute is to be attached to
attr_name	IN:	Name of 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 stored in the attribute

Purpose Attaches an attribute to an object.

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

Description **SDsetattr** attaches the attribute to the object specified by the *obj_id* parameter. The attribute is defined by its name, *attr_name*, data type, *data_type*, number of attribute values, *count*, and the attribute values, *values*. **SDsetattr** provides a generic way for users to define metadata. It implements the label = value data abstraction.

The value of *obj_id* can be an SD interface identifier (*sd_id*), returned by **SDstart**, a data set identifier (*sds_id*), returned by **SDcreate** or **SDselect**, or a dimension identifier (*dim_id*), returned by **SDgetdimid**.

If an SD interface identifier (*sd_id*) is specified as the *obj_id* parameter, a global attribute is created which applies to all objects in the file. If a data set identifier (*sds_id*) is specified as the *obj_id* parameter, an attribute is attached to the specified data set. If a dimension identifier (*dim_id*) is specified as the *obj_id* parameter, an attribute is attached to the specified dimension.

The *attr_name* argument can be any ASCII string.

The *data_type* parameter can contain any data type supported by the HDF library. These data types are listed in Table 1A in Section I of this manual.

Attribute values are passed in the parameter *values*. The number of attribute values is defined by the *count* parameter. If more than one value is stored, all values must have the same data type. If an attribute with the given name, data type and number of values exists, it will be overwritten.

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.

FORTRAN

```
integer function sfsnatt(obj_id, attr_name, data_type, count,
                    values)

integer obj_id, data_type, count

character*(*) attr_name

<valid numeric data type> values(*)
```

```
integer function sfscatt(obj_id, attr_name, data_type, count,
                  values)

integer obj_id, data_type, count

character*(*) attr_name, values
```

SDsetblocksize/sfsblsz

intn SDsetblocksize(int32 *sd_id*, int32 *block_size*)

sd_id	IN:	SD interface identifier returned by **SDstart**
block_size	IN:	Size of the block in bytes

Purpose Sets the block size used for storing data sets with unlimited dimensions.

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

Description **SDsetblocksize** sets the block size defined in the parameter *block_size* for all data sets in the file. **SDsetblocksize** is used when creating new data sets only; it has no effect on pre-existing data sets.

SDsetblocksize must be used after calls to **SDcreate** or **SDselect** and before the call to **SDwritedata**.

The *block_size* parameter should be set to a multiple of the desired buffer size.

FORTRAN

```
integer sfsblsz(sd_id, block_size)

integer sd_id, block_size
```

SDsetcal/sfscal

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

sds_id	IN:	Data set identifier returned by **SDcreate** or **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 data set. 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 *cal_value* stored in a data set and the original value is defined as: *orig_value* = *cal* * (*cal_value* - *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.

The calibration information is automatically cleared after a call to **SDreaddata** or **SDwritedata**. Therefore, **SDsetcal** must be called once for each data set that is to be read or 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/sfschnk

intn SDsetchunk(int32 *sds_id*, HDF_CHUNK_DEF *cdef*, int32 *flag*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
C only:		
cdef	IN:	Pointer to the chunk definition
flag	IN:	Compression flag
Fortran only:		
dim_length	IN:	Chunk dimensions array
comp_type	IN:	Type of compression
comp_prm	IN:	Compression parameters array

Purpose Sets the chunk size and the compression method, if any, of a data set.

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

Description **SDsetchunk** makes the data set specified by the parameter *sds_id* a chunked data set according to the chunking and compression information provided in the parameters *cdef* and *flag* in C, and in the parameters *comp_type* and *comp_prm* in Fortran. Note that chunking and unlimited dimension are not supported together; **SDsetchunk** will fail if the dataset has an unlimited dimension.

C only:

The parameter *flag* specifies the type of the data set, i.e., if the data set is chunked or chunked and compressed with either RLE, Skipping Huffman, GZIP, Szip, or NBIT compression methods. Valid values of *flag* are as follows:

`HDF_CHUNK` for a chunked data set with no compression
`HDF_CHUNK` | `HDF_COMP` for a chunked data set compressed with RLE, Skipping Huffman, GZIP, or Szip compression methods
`HDF_CHUNK` | `HDF_NBIT` for a chunked and NBIT-compressed data set

Chunking and compression information are passed in the parameter *cdef*. The parameter *cdef* has a type of `HDF_CHUNK_DEF`, defined in the HDF library as follows:

```
typedef union hdf_chunk_def_u
{
    int32 chunk_lengths[2];    /* chunk lengths along each dim */
    struct
    {
        int32 chunk_lengths[2];
        int32 comp_type;        /* compression type */
        struct comp_info cinfo;
    } comp;
    struct
    {
        int32 chunk_lengths[2];
        intn start_bit;
        intn bit_len;
        intn sign_ext;
        intn fill_one;
    } nbit;
} HDF_CHUNK_DEF
```

There are three pieces of chunking and compression information which should be specified: chunking dimensions, compression type, and, if needed, compression parameters.

If the data set is chunked, i.e., *flag* value is `HDF_CHUNK`, then `chunk_lengths[]` elements of *cdef* union (`cdef.chunk_lengths[]`) have to be initialized to the chunk dimensions.

If the data set is chunked and compressed using RLE, Skipping Huffman, Szip, or GZIP methods (i.e., *flag* value is set up to `HDF_CHUNK | HDF_COMP`), then the elements `chunk_lengths[]` of the structure `comp` in the union *cdef* (`cdef.comp.chunk_lengths[]`) have to be initialized to the chunk dimensions.

If the data set is chunked and NBIT compression is applied (i.e., *flag* values is set up to `HDF_CHUNK | HDF_NBIT`), then the elements `chunk_lengths[]` of the structure `nbit` in the union *cdef* (`cdef.nbit.chunk_lengths[]`) have to be initialized to the chunk dimensions.

Compression types are passed in the field `comp_type` of the structure `cinfo` , which is an element of the structure `comp` in the union *cdef* (`cdef.comp.cinfo.comp_type`). Refer to the **SDsetcompress** page in this manual for the definition of structure `comp_info`. Valid compression methods are:

`COMP_CODE_NONE` for no compression
`COMP_CODE_RLE` for RLE run-length encoding
`COMP_CODE_SKPHUFF` for Skipping Huffman compression
`COMP_CODE_DEFLATE` for GZIP compression
`COMP_CODE_SZIP` for Szip compression

For Skipping Huffman and GZIP compression, parameters are passed in corresponding fields of the structure `cinfo`.

- Specify skipping size for Skipping Huffman compression in the field `cdef.comp.cinfo.skphuff.skp_size`, which must be an integer of value 1 or greater.
- Specify the deflate level for GZIP compression in the field `cdef.comp.cinfo.deflate_level`. Valid deflate level values are integers between 0 and 9 inclusive.
- Specify the options mask and the number of pixels per block for Szip compression in the fields `c_info.szip.options_mask` and `c_info.szip.pixels_per_block`, respectively.

Refer to the **SDsetcompress** entry in this reference manual for details on these parameters.

NBIT compression parameters are specified in the fields `start_bit`, `bit_len`, `sign_ext`, and `fill_one` in the structure `nbit` of the union *cdef*.

Fortran only:

The *dim_length* array specifies the chunk dimensions.

The *comp_type* parameter specifies the compression type. Valid compression types and their values are defined in the `hdf.inc` file, and are listed below:

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

The *comp_prm*(1) parameter specifies the skipping size for the Skipping Huffman compression method and the deflate level for the GZIP compression method. The skipping size value must be 1 or greater; the deflate level must be an integer value between 0 and 9 inclusive.

For NBIT compression, the four elements of the array *comp_prm* correspond to the four NBIT compression parameters listed in the structure `nbit`. The value of *comp_prm*(1) should be set to the value of `start_bit`, the value of *comp_prm*(2) should be set to the value of `bit_len`, the value of *comp_prm*(3) should be set to the value of `sign_ext`, and the value of *comp_prm*(4) should be set to the value of `fill_one`. See the `HDF_CHUNK_DEF` union description and the description of **SDsetnbitdataset** function for NBIT compression parameters definitions.

For Szip compression, the first two elements of the array *comp_prm* correspond to the first two Szip compression parameters listed in the structure `szip`. The value of *comp_prm*(1) should be set to the value of `option_mask` and the value of *comp_prm*(2) should be set to the value of `pixels_per_block`.

FORTRAN

```
integer sfschnk(sds_id, dim_length, comp_type, comp_prm)

integer sds_id, dim_length, comp_type, comp_prm(*)
```

SDsetchunkcache/sfscchnk

intn SDsetchunkcache(int32 *sds_id*, int32 *maxcache*, int32 *flag*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
maxcache	IN:	Maximum number of chunks in the cache
flag	IN:	Flag determining the behavior of the routine

Purpose Sets the size of the chunk cache.

Return value Returns the maximum number of chunks that can be cached (the value of the parameter *maxcache*) if successful and `FAIL` (or `-1`) otherwise.

Description **SDsetchunkcache** sets the size of the chunk cache to the value of the parameter *maxcache*.

Currently the only allowed value of the parameter *flag* is `0`, which designates default operation.

By default, when a generic data set is promoted to be a chunked data set, the parameter *maxcache* is set to the number of chunks along the fastest changing dimension and a cache for the chunks is created.

If the chunk cache is full and the value of the parameter *maxcache* is greater then the current *maxcache* value, then the chunk cache is reset to the new value of *maxcache*. Otherwise the chunk cache remains at the current value of *maxcache*. If the chunk cache is not full, then the chunk cache is set to the new value of *maxcache* only if the new *maxcache* value is greater than the current number of chunks in the cache.

Do not set the value of *maxcache* to be less than the number of chunks along the fastest-changing 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 14, "HDF Performance Issues" in the *HDF User's Guide*, for more information on this.

FORTRAN

```
integer sfscchnk(sds_id, maxcache, flag)

integer sds_id, maxcache, flag
```

SDsetcompress/sfscompress

intn SDsetcompress(int32 *sds_id*, int32 *comp_type*, comp_info **c_info*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
comp_type	IN:	Compression method
C only:		
c_info	IN:	Pointer to the `comp_info` union
Fortran only:		
comp_prm	IN:	Compression parameters array

Purpose Compresses the data set with the specified compression method.

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

Description **SDsetcompress** compresses the data set identified by the parameter *sds_id* according to the compression method specified by the parameter *comp_type* and the compression information specified by the parameter *c_info* in C and *comp_prm* in Fortran. **SDsetcompress** prepares compression information to be used uring the next call to **SDwritedata**.

SDsetcompress 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 parameter *comp_type* is the compression type definition and is set to one of the following:

`COMP_CODE_NONE` for no compression
`COMP_CODE_RLE` for run-length encoding (RLE)
`COMP_CODE_SKPHUFF` for Skipping Huffman
`COMP_CODE_DEFLATE` for GZIP compression
`COMP_CODE_SZIP` for Szip compression

The parameter *c_info* is a pointer to a union structure of type `comp_info`. This union structure is defined as follows:

```
typedef union tag_comp_info
{
    struct
    {
        /* Not used by SDsetcompress */
    } jpeg;
    struct
    {
        /* Not used by SDsetcompress */
    } nbit;
    struct
    {  /* struct to contain info about how to compress size of the
        elements when skipping */
        intn skp_size;
    } skphuff;
    struct
    {  /* struct to contain info about how to compress or decompress
        gzip encoded dataset how hard to work when compressing
        data*/
        intn level;
    } deflate;
    struct
    {  /* struct to contain info about how to compress or decompress
        szip encoded dataset*/
        int32 options_mask;           /* IN */
        int32 pixels_per_block;       /* IN */
        int32 pixels_per_scanline;    /* OUT */
        int32 bits_per_pixel;         /* OUT */
        int32 pixels;                 /* OUT */
    } szip;
} comp_info;
```

The skipping size for the Skipping Huffman algorithm must be 1 or greater and is specified in the field `c_info.skphuff.skp_size` in C and in the parameter *comp_prm*(1) in Fortran.

The deflate level for the GZIP algorithm is specified in the `c_info.deflate.level` field in C and in the parameter *comp_prm*(1) in Fortran. Valid values are integers between 0 and 9 inclusive.

The Szip options mask and the number of pixels per block in a chunked and Szip-compressed dataset are specified in `c_info.szip.options_mask` and `c_info.szip.pixels_per_block`, respectively.

The options mask can contain either of the following values:
- `SZ_EC_OPTION_MASK` - Specifies entropy coding method
- `SZ_NN_OPTION_MASK` - Specifies nearest neighbor coding method

The following guidelines may be helpful in selecting the encoding method:
- The entropy coding method, the EC option specified by `SZ_EC_OPTION_MASK`, is best suited for data that has been processed. The EC method works best for small numbers.
- The nearest neighbor coding method, the NN option specified by `SZ_NN_OPTION_MASK`, preprocesses the data then applies the EC method as above.

Other factors may affect results, but the above criteria provide a good starting point for optimizing data compression.

The Szip values of the number of pixels per scanline, the number of bits in a pixel, and the number of pixels in an image, are computed by the HDF4 library and provided to the user in `c_info.szip.pixels_per_scanline`, `c_info.szip.bits_per_pixel`, and `c_info.szip.pixels`, respectively.

SDsetcompress will succeed in setting Szip compression for a dataset only if the Szip library is available and configured for encoding, i.e., **HCget_config_info** must return the flag `COMP_DECODER_ENABLED|COMP_ENCODER_ENABLED` in *compression_config_info*.

Note **Regarding Szip usage and licensing:**
See `http://www.hdfgroup.org/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

FORTRAN

```
integer sfscompress(sds_id, comp_type, comp_prm)

integer sds_id, comp_type, comp_prm(*)
```

SDsetdatastrs/sfsdtstr

intn SDsetdatastrs(int32 *sds_id*, char **label*, char **unit*, char **format*, char **coordsys*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
label	IN:	Label (predefined attribute)
unit	IN:	Unit (predefined attribute)
format	IN:	Format (predefined attribute)
coordsys	IN:	Coordinate system (predefined attribute)

Purpose Sets the predefined attributes for a data set.

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

Description **SDsetdatastrs** sets the predefined attributes of the data set, identified by *sds_id*, to the values specified in the parameters *label*, *unit*, *format* and *coordsys*. The predefined attributes are label, unit, format, and coordinate system. If the user does not want a string returned, the corresponding parameter can be set to `NULL` in C and an empty string in Fortran.

For more information about predefined attributes, refer to Section 3.10, "Predefined Attributes" of the *HDF User's Guide*.

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 by **SDgetdimid**
dim_name	IN:	Name of the dimension

Purpose Assigns a name to a dimension.

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

Description **SDsetdimname** sets the name of the dimension identified by the parameter *dim_id* to the value specified in the parameter *dim_name*. Dimensions that are not explicitly named by the user will have the default name of "`fakeDim[x]`" specified by the HDF library, where [`x`] denotes the dimension index.

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. If the dimension with the same name has a different size, an error condition will result.

The length of the parameter *dim_name* can be at most `H4_MAX_NC_NAME` (`256`) characters.

Naming dimensions is optional but encouraged.

Note **Regarding naming a dimension the same as an SDS' name**
HDF4 allows a dimension and a one-dimensional SDS to be given the same name. Prior to HDF4.2r2, however, the library did not always adequately track object types; when a dimension and a one-dimensional SDS shared a name, writing to the SDS or the dimension could cause data corruption to the other. The corrupted data was unrecoverable.

This problem was fixed in Release 4.2r2 and such data corruption will not occur in files created with a 4.2r2 or later library. Note, however, that the fix is effective only in new files; a dimension and a one-dimensional SDS of the same name that were created with a pre-4.2r2 HDF4 Library remain vulnerable to data corruption if an application is unaware of the potential conflict. To safely handle pre-4.2r2 files, the library now provides two functions, **SDgetnumvars_byname** and **SDnametoindices**. **SDgetnumvars_byname** can be used to determine whether a name is unique. If the function reports one ('1') variable by that name, the name is unique and no further precaution needs to be taken. If the name is not unique, i.e., the number of variables by that name is greater than one, **SDnametoindices** must then be used to retrieve the index and the type of each variable with that name. The desired variable can then can be safely selected via its index. These functions are described in detail in this manual and the HDF User's Guide.

A similar problem is possible when a multi-dimensional SDS and a dimension are created with the same name by a pre-4.2r2 library. The HDF Group has not seen such a failure, however, and it is thought to be very unlikely. Note that the fix introduced in Release 4.2r2 also prevents data corruption from happening for this situation even though the data was created with libraries prior to 4.2r2, assuming no corruption had yet occurred.

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 by **SDgetdimid**
count	IN:	Total number of values along the dimension
data_type	IN:	Data type of the values along the dimension
data	IN:	Value of each increment along the dimension

Purpose Stores the values of a dimension.

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

Description **SDsetdimscale** stores scale information for the dimension identified by the parameter *dim_id*. Note that it is possible to store dimension scale values without naming the dimension.

For fixed-size arrays, the value of *count* must be equal to the the dimension size or the routine will fail.

Note that, due to the existence of the parameter *data_type*, the dimension scales need not have the same data type as the data set.

Note that if **SDsetdimscale** is called and **SDsetdimname** is subsequently called for the same dimension, **SDsetdimscale** must be called again to reassociate the scale with the new name.

Note that if this dimension has the same name as another data set, **SDsetdimscale** might overwrite the data set. This was a bug in the HDF4 library prior to 4.2r2. For details, please refer to the note "Regarding naming a dimension the same as an SDS' name" in the **SDsetdimname** page.

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

FORTRAN

```
integer function sfsdscale(dim_id, count, data_type, data)

integer dim_id, count, data_type

<valid data type> data(*)
```

SDsetdimstrs/sfsdmstr

intn SDsetdimstrs(int32 *dim_id*, char **label*, char **unit*, char **format*)

dim_id	IN:	Dimension identifier returned by **SDgetdimid**
label	IN:	Label (predefined attribute)
unit	IN:	Unit (predefined attribute)
format	IN:	Format (predefined attribute)

Purpose Sets the predefined attribute of a dimension.

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

Description **SDsetdimstrs** sets the predefined attribute (label, unit, and format) for a dimension and its scale to the values specified in the parameters *label*, *unit* and *format*. If a parameter is set to `NULL` in C and an empty string in Fortran, then the attribute corresponding to that parameter will not be written. For more information about predefined attributes, refer to Section 3.10, "Predefined Attributes" of the *HDF User's Guide*.

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 by **SDgetdimid**
comp_mode	IN:	Compatibility mode to be set

Purpose Determines whether a dimension *will have* the old and new representations or the new representation only.

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

Description **SDsetdimval_comp** sets the compatibility mode specified by the *comp_mode* parameter for the dimension identified by the *dim_id* parameter. The two possible compatibility modes are: "backward-compatible" mode, which implies that the old and new dimension representations are written to the file, and "backward-incompatible" mode, which implies that only the new dimension representation is written to the file.

Unlimited dimensions are always backward-compatible, therefore **SDsetdimval_comp** takes no action on unlimited dimensions.

As of HDF version 4.1r1, the default mode is backward-incompatible. Subsequent calls to **SDsetdimval_comp** will override the settings established in previous calls to the routine.

The *comp_mode* parameter can be set to `SD_DIMVAL_BW_COMP` (or `1`), which specifies backward-compatible mode, or `SD_DIMVAL_BW_INCOMP` (or `0`), which specifies backward-incompatible mode.

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:	Data set identifier returned by **SDcreate** or **SDselect**
filename	IN:	Name of the external file
offset	IN:	Number of bytes from the beginning of the external file to where the data will be written

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 actual data values (i.e., not metadata) of a data set, *sds_id*, into the external data file named by the parameter *filename*, and started at the offset specified by the parameter *offset*. The metadata remains in the original file. Note that this routine works only with HDF post-version 3.2 files.

Data can only be moved once for any given data set, and it is the user's responsibility to make sure the external data file is kept with the “original” file.

If the data set already exists, its data will be moved to the external file. Space occupied by the data in the primary file will not be released. To release the space in the primary file use the `hdfpack` command-line utility. If the data set does not exist, its data will be written to the external file during the consequent calls to **SDwritedata**.

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 *sd_id*, intn *fill_mode*)

sd_id	IN:	SD interface identifier returned by **SDstart**
fill_mode	IN:	Fill mode

Purpose Sets the current fill mode of a file.

Return value Returns the fill mode value before it was reset if successful and `FAIL` (or `-1`) otherwise.

Description **SDsetfillmode** applies the fill mode specified by the parameter *fill_mode* to all data sets contained in the file identified by the parameter *sd_id*.

Possible values of *fill_mode* are `SD_FILL` (or `0`) and `SD_NOFILL` (or `256`). `SD_FILL` is the default mode, and indicates that fill values will be written when the data set is created. `SD_NOFILL` indicates that fill values will not be written.

When a data set without unlimited dimensions is created, by default the first **SDwritedata** call will fill the entire data set with the default or user-defined fill value (set by **SDsetfillvalue**). In data sets with an unlimited 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 has not been specified.

If it is certain that all data set values will be written before any read operation takes place, there is no need to write the fill values. Simply call **SDsetfillmode** with *fill_mode* value set to `SD_NOFILL`, which will eliminate all fill value write operations to the data set. For large data sets, this can improve the speed by almost 50%.

FORTRAN

```
integer function sfsflmd(sd_id, fill_mode)

integer sd_id, fill_mode
```

SDsetfillvalue/sfsfill/sfscfill

intn SDsetfillvalue(int32 *sds_id*, VOIDP *fill_value*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
fill_value	IN:	Fill value

Purpose Sets the fill value for a data set.

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

Description **SDsetfillvalue** sets the fill value specified by the *fill_value* parameter for the data set identified by the *sds_id* parameter.

The fill value is assumed to have the same data type as the data set.

It is recommended to call **SDsetfillvalue** before writing data.

FORTRAN

```
integer function sfsfill(sds_id, fill_value)

integer sds_id

<valid numeric data type> fill_value


integer function sfscfill(sds_id, fill_value)

integer sds_id

character*(*) fill_value
```

SDsetnbitdataset/sfsnbit

intn SDsetnbitdataset(int32 *sds_id*, intn *start_bit*, intn *bit_len*, intn *sign_ext*, intn *fill_one*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
start_bit	IN:	Leftmost bit of the field to be written
bit_len	IN:	Length of the bit field to be written
sign_ext	IN:	Sign extend specifier
fill_one	IN:	Background bit specifier

Purpose Specifies a non-standard bit length for the data set values.

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

Description **SDsetnbitdataset** allows the HDF user to specify that the data set identified by the parameter *sds_id* contains data of a non-standard length defined by the parameters *start_bit* and *bit_len*. Additional information about the non-standard bit length decoding are specified in the parameters *sign_ext* and *fill_one*.

Any length between 1 and 32 bits can be specified. After **SDsetnbitdataset** has been called for the data set array, any read or write operations will involve a conversion between the new data length of the data set 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 preceding paragraph a *start_bit* parameter set to `4` would correspond to 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 preceding paragraph would result in the bit field `1110` being written to the data set. This would correspond to a *start_bit* value of `5` and a *bit_len* value of `4`.

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

The *fill_one* specifies whether to fill the “background” bits with the value `1` or `0`. This parameter can also be set to `TRUE` or `FALSE`.

The "background" bits of a variable-length data set 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 data set 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:	Data set identifier returned by **SDcreate** or **SDselect**
max	IN:	Maximum value of the range
min	IN:	Minimum value of the range

Purpose Sets the maximum and minimum range values for a data set.

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

Description **SDsetrange** sets the maximum and minimum range values of the data set identified by the parameter *sds_id* with the values of the parameters *max* and *min*. The term "range" is used here to describe the range of numeric values stored in a data set.

It is assumed that the data type for the maximum and minimum range values are the same as the data type of the data.

This routine does not compute the maximum and minimum range values, it only stores the values as given. As a result, the maximum and minimum range values may not always reflect the actual maximum and minimum range values in the data set data.

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 file access mode in effect during the current session

Purpose Opens an HDF file and initializes an SD interface.

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

Description **SDstart** opens the file with the name specified by the parameter *filename*, with the access mode specified by the parameter *access_mode*, and returns an SD interface identifier (*sd_id*). This routine must be called for each file before any other SD calls can be made on that file.

The type of identifier returned by **SDstart** is currently not the same as the identifier returned by **Hopen**. As a result, the SD interface identifiers (*sd_id*) returned by this routine are not understood by other HDF interfaces.

To mix SD API calls and other HDF API calls, use **SDstart** and **Hopen** on the same file. **SDstart** must precede all SD calls, and **Hopen** must precede all other HDF function calls. To terminate access to the file, use **SDend** to dispose of the SD interface identifier, *sd_id*, and **Hclose** to dispose of the file identifier, *file_id*.

The file identified by the parameter *filename* can be any one of the following: an XDR-based netCDF file, “old-style” DFSD file or a “new-style” SD file.

The value of the parameter *access_mode* can be one of the following:

`DFACC_READ` - Open existing file for read-only access. If the file does not exist, specifying this mode will cause **SDstart** to return `FAIL` (or `-1`).
`DFACC_WRITE` - Open existing file for read and write access. If the file does not exist, specifying this mode will cause **SDstart** to return `FAIL` (or `-1`).
`DFACC_CREATE` - Create a new file with read and write access. If the file has already existed, its contents will be replaced.

Note Starting from HDF 4.2r2, the maximum number of open files is no longer limited to 32. It can be up to what the system allowed.

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

FORTRAN

```
integer function sfstart(filename, access_mode)

character*(*) filename

integer access_mode
```

SDwritechunk/sfwchnk/sfwcchnk

intn SDwritechunk(int32 *sds_id*, int32 **origin*, VOIDP *datap*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
origin	IN:	Origin of the chunk to be written
datap	IN:	Buffer for the chunk data to be written

Purpose Writes a data chunk to a chunked data set.

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

Description **SDwritechunk** writes the entire chunk of data stored in the buffer *datap* to the chunked data set identified by the parameter *sds_id*. Writing starts at the location specified by the parameter *origin*. **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 data set.

SDwritechunk will return `FAIL` (or `-1`) when an attempt is made to use it to write to a non-chunked data set.

The parameter *origin* specifies the coordinates of the chunk according to the chunk position in the overall chunk array. Refer to Chapter 3, "Scientific Data Sets (SD API)" in the *HDF User's Guide*, for a description of the organization of chunks in a data set.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (**sfwchnk**) and one for character data (**sfwcchnk**).

Note **Regarding Szip-compressed data:**
SDwritechunk can succeed only when the available Szip library is configured for encoding/decoding, i.e., when **HCget_config_info** returns `COMP_ENCODER_ENABLED|COMP_DECODER_ENABLED` in *compression_config_info*.

See the **SDgetcompinfo** and **HCget_config_info** entries in this reference manual for further discussion of compression methods and configuration.

Note **Regarding Szip usage and licensing:**
See `http://www.hdfgroup.org/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

FORTRAN

```
integer sfwchnk(sds_id, origin, datap)

integer sds_id, origin

<valid numeric data type> datap(*)


integer sfwcchnk(sds_id, origin, datap)

integer sds_id, origin

character*(*) datap(*)
```

SDwritedata/sfwdata/sfwcdata

intn SDwritedata(int32 *sds_id*, int32 *start*[], int32 *stride*[], int32 *edge*[], VOIDP *buffer*)

sds_id	IN:	Data set identifier returned by **SDcreate** or **SDselect**
start	IN:	Array specifying the starting location of the data to be written
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
buffer	IN:	Buffer for the values to be written

Purpose Writes a subsample of data to a data set or to a coordinate variable.

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

Description **SDwritedata** writes the specified subsample of data to the data set or coordinate variable identified by the parameter *sds_id*. The data is written from the buffer *buffer.* The subsample is defined by the parameters *start*, *stride* and *edge.*

The array *start* specifies the starting position from where the subsample will be written. Valid values of each element in the array *start* are from `0` to the size of the corresponding dimension of the data set - 1. The dimension sizes are returned by **SDgetinfo**.

The array *edge* specifies the number of values to write along each data set dimension.

The array *stride* specifies the writing pattern along each dimension. For example, if one of the elements of the array *stride* is 1, then every element along the corresponding dimension of the data set will be written. If one of the elements of the array *stride* is 2, then every other element along the corresponding dimension of the data set will be written, and so on. Specifying *stride* value of `NULL` in the C interface or setting all values of the array *stride* to 1 in either interface specifies the contiguous writing of data. If all values in the array *stride* are set to 0, **SDwritedata** returns `FAIL` (or `-1`).

When writing data to a chunked data set using **SDwritedata**, consideration should be given to be issues presented in the section on chunking in Chapter 3, "Scientific Data Sets (SD API)" and Chapter 14, "HDF Performance Issues" in the *HDF User's Guide.*

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.

Note **Regarding an important difference between the SD and GR interfaces:**
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as *start*, *stride*, *edge*, and *dimsizes* and the dimension order in the *data* array. See the **SDreaddata** and **GRreadimage** reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the **SDreaddata** and **GRreadimage** functions, respectively. Both functions take the parameters *start*, *stride*, and *edge*.

- For **SDreaddata**, those parameters are expressed in (*y*,*x*) or [*row*,*column*] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For **GRreadimage**, those parameters are expressed in (*x*,*y*) or [*column*,*row*] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

Note **Regarding compressed data sets:**
If a data set is compressed, it may be necessary to determine whether the compression method is available on the current system and configured so that data can be encoded before being written. The compression method can be determined through the use of **SDgetcompinfo** and the configuration of that method on the current system through **HCget_config_info**.

Note **Regarding Szip-compressed data:**
SDwritedata can succeed only when the available Szip library is configured for encoding/decoding, i.e., when **HCget_config_info** returns `COMP_ENCODER_ENABLED|COMP_DECODER_ENABLED` in *compression_config_info*.

Note **Regarding Szip usage and licensing:**
See `http://www.hdfgroup.org/doc_resource/SZIP/` for information regarding the use of Szip in HDF products and Szip licensing.

FORTRAN

```
integer function sfwdata(sds_id, start, stride, edge, buffer)

integer sds_id

integer start(*), stride(*), edge(*)

<valid numeric data type> buffer(*)


integer function sfwcdata(sds_id, start, stride, edge, buffer)

integer sds_id

integer start(*), stride(*), edge(*)

character*(*) buffer(*)
```

Vaddtagref/vfadtr

int32 Vaddtagref(int32 *vgroup_id*, int32 *tag*, int32 *ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag	IN:	Tag of the object
ref	IN:	Reference number of the object

Purpose Inserts an object into a vgroup.

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

Description **Vaddtagref** inserts the object identified by the parameters *tag* and *ref* into the vgroup identified by the parameter *vgroup_id*.

If an object to be inserted is a data set, duplication of the tag/reference number pair will be allowed. Otherwise, the tag/reference number pair must be unique among the elements within the vgroup or the routine will return `FAIL` (or `-1`).

Note that **Vaddtagref** does not verify that the tag and reference number exist.

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
access	IN:	Type of access

Purpose Initiates access to a new or existing vgroup.

Return value Returns the vgroup identifier (*vgroup_id*) if successful and `FAIL` (or `-1`) otherwise.

Description **Vattach** opens a vgroup with access type specified by the parameter *access* in the file identified by the parameter *file_id*. The vgroup is identified by the reference number, *vgroup_ref*.

Vattach returns the vgroup identifier, *vgroup_id*, for the accessed vgroup. The *vgroup_id* is used for all subsequent operations on this vgroup. Once operations are complete, the vgroup identifier must be disposed of via a call to **Vdetach**. Multiple attaches may be made to the same vgroup simultaneously, and several vgroup identifiers can be created for the same vgroup. Each vgroup identifier must be disposed of independently.

The parameter *file_id* is the file identifier of an opened file. The parameter *vgroup_ref* specifies which vgroup in the file to attach to. If *vgroup_ref* is set to `-1`, a new vgroup will be created. If *vgroup_ref* is set to a positive number, the vgroup with that as a reference number is attached.

Possible values for the parameter *access* are “r” for read access and “w” for write access.

FORTRAN

```
integer function vfatch(file_id, vgroup_ref, access)

integer file_id, vgroup_ref

character*1 access
```

Vattrinfo/vfainfo

intn Vattrinfo(int32 *vgroup_id*, intn *attr_index*, char **attr_name*, int32 **data_type*, int32 **count*, int32 **size*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_index	IN:	Index of the attribute
attr_name	OUT:	Name of the attribute
data_type	OUT:	Data type of the attribute
count	OUT:	Number of values in the attribute
size	OUT:	Size, in bytes, of the attribute values.

Purpose Retrieves the name, data type, number of values, and value size of an attribute for a vgroup.

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

Description **Vattrinfo** retrieves the name, datatype, number of values, and value size of an attribute identified by its index, *attr_index*, in the vgroup, *vgroup_id*. Name, data type, number of values and size are retrieved into the parameters *attr_name*, *data_type*, *count*, and *size*, respectively.

If the attribute's name, data type, number of values, or value size are not needed, the corresponding output parameters can be set to `NULL`.

The valid value *attr_index* range from 0 to the total number of attributes attached to a vgroup - 1. The number of vgroup attributes can be obtained using **Vnattrs**.

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
```

Vdelete/vdelete

int32 Vdelete(int32 *file_id*, int32 *vgroup_ref*)

file_id	IN:	File identifier returned by **Hopen**
vgroup_ref	IN:	Vgroup reference number returned by **Vattach**

Purpose Remove a vgroup from a file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) if not successful.

Description **Vdelete** removes the vgroup identified by the parameter *vgroup_ref* from the file identified by the parameter *file_id*.

This routine will remove the vgroup from the internal data structures and from the file.

FORTRAN

```
integer function vdelete(file_id, vgroup_ref)

integer file_id, vgroup_ref
```

Vdeletetagref/vfdtr

int32 Vdeletetagref(int32 *vgroup_id*, int32 *tag*, int32 *ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag	IN:	Tag of the object
ref	IN:	Reference number of the object

Purpose Deletes an object from a vgroup.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) if not successful or the given tag/reference number pair is not found in the vgroup.

Description **Vdeletetagref** deletes the object specified by the parameters *tag* and *ref* from the vgroup identified by the parameter *vgroup_id*. **Vinqtagref** should be used to check if the tag/reference number pair exists before calling this routine.

If duplicate tag/reference number pairs are found in the vgroup, **Vdeletetagref** deletes the first occurrence. **Vinqtagref** should be used to determine if duplicate tag/reference number pairs exist in the vgroup.

FORTRAN

```
integer function vfdtr(vgroup_id, tag, ref)

integer vgroup_id, tag, ref
```

Vdetach/vfdtch

int32 Vdetach(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Terminates access to a vgroup.

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

Description **Vdetach** detaches the currently-attached vgroup identified by *vgroup_id* and terminates access to that vgroup.

All space associated with the vgroup, *vgroup_id*, will be freed. Each attached vgroup must 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 the 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 Terminates access to a vgroup and/or vdata interface.

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

Description **Vend** terminates access to the vgroup and/or vdata interfaces initiated by **Vstart** and all internal data structures allocated by **Vstart**.

Vend must be called after all vdata and vgroup operations on the file *file_id* are completed. Further attempts to use vdata or vgroup routines after calling **Vend** will result in a `FAIL` (or `-1`) being returned.

FORTRAN

```
integer function vfend(file_id)

integer file_id
```

Vfind/vfind

int32 Vfind(int32 *file_id*, char **vgroup_name*)

file_id	IN:	File identifier returned by **Hopen**
vgroup_name	IN:	Name of the vgroup

Purpose Returns the reference number of a vgroup given its name.

Return value Returns the reference number of the vgroup if successful and `0` otherwise.

Description **Vfind** searches the file identified by the parameter *file_id* for a vgroup with the name specified by the parameter *vgroup_name*, and returns the corresponding reference number.

If more than one vgroup has the same name, **Vfind** will return the reference number of the first one.

FORTRAN

```
integer function vfind(file_id, vgroup_name)

integer file_id

character*(*) vgroup_name
```

Vfindattr/vffdatt

intn Vfindattr(int32 *vgroup_id*, char **attr_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_name	IN:	Name of the attribute

Purpose Returns the index of a vgroup attribute given its name.

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

Description **Vfindattr** searches the vgroup identified by the parameter *vgroup_id* for the attribute with the name specified by the parameter *attr_name*, and returns the index of that attribute.

If more than one attribute has the same name, **Vfindattr** will return the index of the first one.

FORTRAN

```
integer function vffdatt(vgroup_id, attr_name)

integer vgroup_id

character*(*) attr_name
```

Vfindclass/vfndcls

int32 Vfindclass(int32 *file_id*, char **vgroup_class*)

file_id	IN:	File identifier returned by **Hopen**
vgroup_class	IN:	Class name of the vgroup

Purpose Returns the reference number of a vgroup specified by its class name.

Return value Returns the reference number of the vgroup if successful and `0` otherwise.

Description **Vfindclass** searches the file identified by the parameter *file_id* for the vgroup with the class name specified by the parameter *vgroup_class*, and returns the reference number of that vgroup.

If more than one vgroup has the same class name, **Vfindclass** will return the reference number of the first one.

FORTRAN

```
integer function vfndcls(file_id, vgroup_class)

integer file_id

character*(*) vgroup_class
```

Vflocate/vffloc

int32 Vflocate(int32 *vgroup_id*, char **field_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
field_name_list	IN:	List of field names

Purpose Locates a vdata in a vgroup given a list of field names.

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

Description **Vflocate** searches the vgroup identified by the parameter *vgroup_id* for a vdata that contains all of the fields listed in the parameter *field_name_list*. If that vdata is found, **Vflocate** will return its reference number.

FORTRAN

```
integer function vffloc(vgroup_id, field_name)

integer vgroup_id

character*(*) field_name
```

Vgetattr/vfgnatt/vfgcatt

intn Vgetattr(int32 *vgroup_id*, intn *attr_index*, VOIDP *attr_values*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
attr_index	IN:	Index of the attribute
attr_values	OUT:	Buffer for the attribute values

Purpose Retrieves the values of a vgroup attribute.

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

Description **Vgetattr** retrieves the values of the attribute identified by its index, *attr_index*, into the buffer *attr_values* for the vgroup identified by the parameter *vgroup_id*.

The valid values of the parameter *attr_index* range from 0 to the total number of vgroup attributes - 1. The total number of attributes can be obtained using **Vnattrs**. To determine the amount of memory sufficient to hold the attribute values, the user can obtain the number of attribute values and the attribute value size using **Vattrinfo**.

FORTRAN

```
integer function vfgnatt(vgroup_id, attr_index, attr_values)

integer vgroup_id, attr_index

<valid numeric data type> attr_values


integer function vfgcatt(vgroup_id, attr_index, attr_values)

integer vgroup_id, attr_index

character*(*) attr_values
```

Vgetclass/vfgcls

int32 Vgetclass(int32 *vgroup_id*, char **vgroup_class*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_class	OUT:	Class name of the vgroup

Purpose Retrieves the class name of a vgroup.

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

Description **Vgetclass** retrieves the class name of the vgroup identified by the parameter *vgroup_id* in the buffer *vgroup_class*.

The maximum length of the name is defined by `VGNAMELENMAX (or 64)`.

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 current vgroup

Purpose Returns the reference number of the next vgroup.

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

Description **Vgetid** sequentially searches the file identified by the parameter *file_id* and returns the reference number of the vgroup following the vgroup that has the reference number specified by the parameter *vgroup_ref*.

The search is initiated by calling 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 cause **Vgetid** to return `FAIL` (or `-1`).

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:	Name of the vgroup

Purpose Retrieves the name of a vgroup.

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

Description **Vgetname** retrieves the name of the vgroup identified by the parameter *vgroup_id* into the buffer *vgroup_name*. The maximum length of the name is defined by `VGNAMELENMAX` (or `64`).

FORTRAN

```
integer function vfgnam(vgroup_id, vgroup_name)

integer vgroup_id

character*(*) vgroup_name
```

Vgetnext/vfgnxt

int32 Vgetnext(int32 *vgroup_id*, int32 *v_ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
v_ref	IN:	Reference number of the vgroup or vdata

Purpose Gets the reference number of the next member (vgroup or vdata only) of a vgroup.

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

Description **Vgetnext** searches in the vgroup identified by the parameter *vgroup_id* for the object following the object specified by its reference number *v_ref*. Either of the two objects can be a vgroup or a vdata. If *v_ref* is set to -1, the routine will return the reference number of the first vgroup or vdata in the vgroup.

Note that this routine only gets a vgroup or a vdata in a vgroup. **Vgettagrefs** gets any object in a vgroup.

FORTRAN

```
integer function vfgnxt(vgroup_id, v_ref)

integer vgroup_id, v_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 object in the vgroup
tag	OUT:	Tag of the object
ref	OUT:	Reference number of the object

Purpose Retrieves the tag/reference number pair of an object given its index within a vgroup.

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

Description **Vgettagref** retrieves the tag/reference number pair of the object specified by its index, *index*, within the vgroup identified by the parameter *vgroup_id*. Note that this routine is different from **Vgettagrefs**, which retrieves the tag/reference number pairs of a number of objects.

The valid values of *index* range from 0 to the total number of objects in the vgroup - 1. The total number of objects in the vgroup can be obtained using **Vinquire**.

The tag is stored in the buffer *tag* and the reference number is stored in the buffer *ref*.

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 *num_of_pairs*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag_array	OUT:	Array of tags
ref_array	OUT:	Array of reference numbers
num_of_pairs	IN:	Number of tag/reference number pairs

Purpose Retrieves the tag/reference number pairs of the HDF objects belonging to a vgroup.

Return value Returns the number of tag/reference number pairs obtained from a vgroup if successful and `FAIL` (or `-1`) otherwise.

Description **Vgettagrefs** retrieves at most *num_of_pairs* number of tag/reference number pairs belonging to the vgroup, *vgroup_id*, and stores them in the buffers *tag_array* and *ref_array*.

The input parameter *num_of_pairs* specifies the maximum number of tag/reference number pairs to be returned. The size of the arrays, *tag_array* and *ref_array*, must be at least *num_of_pairs*.

FORTRAN

```
integer function vfgttrs(vgroup_id, tag_array, ref_array,
                         num_of_pairs)

integer vgroup_id, num_of_pairs

integer tag_array(*), ref_array(*)
```

Vgetversion/vfgver

int32 Vgetversion(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Gets the version of a vgroup.

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

Description **Vgetversion** returns the version number of the vgroup identified by the parameter *vgroup_id*. There are three valid version numbers: `VSET_OLD_VERSION` (or `2`), `VSET_VERSION` (or `3`), and `VSET_NEW_VERSION` (or `4`).

`VSET_OLD_VERSION` is returned when the vgroup is of a version that corresponds to an HDF library version before version 3.2.

`VSET_VERSION` is returned when the vgroup is of a version that corresponds to an HDF library version between versions 3.2 and 4.0 release 2.

`VSET_NEW_VERSION` is returned when the vgroup is of the version that corresponds to an HDF library version of version 4.1 release 1 or higher.

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 the object
ref	IN:	Reference number of the object

Purpose Checks whether an object belongs to a vgroup.

Return value Returns `TRUE` (or `1`) if the object belongs to the vgroup, and `FALSE` (or `0`) otherwise.

Description **Vinqtagref** checks if the object identified by its tag, *tag*, and its reference number, *ref*, belongs to the vgroup identified by the parameter *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:	Number of entries in a vgroup
vgroup_name	OUT:	Name of a vgroup

Purpose Retrieves the number of entries in a vgroup and its name.

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

Description **Vinquire** retrieves the name of and the number of entries in the vgroup identified by the parameter *vgroup_id* into the buffer *vgroup_name* and the parameter *n_entries*, respectively.

The maximum length of the vgroup name is defined by `VGNAMELENMAX` (or `64`).

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 of the vdata or vgroup

Purpose Inserts a vdata or vgroup into a vgroup.

Return value Returns the position (*index*) of the inserted element within the vgroup if successful and `FAIL` (or `-1`) otherwise.

Description **Vinsert** inserts the vdata or vgroup identified by the parameter *v_id* into the vgroup identified by the parameter *vgroup_id*.

Essentially, **Vinsert** only inserts a vgroup or vdata. To insert any objects into a vgroup, use **Vaddtagref**.

The returned value, *index*, is either 0 or a positive value, which indicates the position of the inserted element in the vgroup.

FORTRAN

```
integer function vfinsrt(vgroup_id, v_id)

integer vgroup_id, v_id
```

Visvg/vfisvg

intn Visvg(int32 *vgroup_id*, int32 *obj_ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
obj_ref	IN:	Reference number of the object

Purpose Determines whether an element of a vgroup is a vgroup and a member of another vgroup.

Return value Returns `TRUE` (or `1`) if the object is a vgroup and `FALSE` (or `0`) otherwise.

Description **Visvg** determines if the object specified by the reference number, *obj_ref*, is a vgroup within the vgroup identified by the parameter *vgroup_id*.

FORTRAN

```
integer function vfisvg(vgroup_id, obj_ref)

integer vgroup_id, obj_ref
```

Visvs/vfisvs

intn Visvs(int32 *vgroup_id*, int32 *obj_ref*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
obj_ref	IN:	Reference number of the object

Purpose Determines whether a data object is a vdata within a vgroup.

Return value Returns `TRUE (or 1)` if the object is a vdata and `FALSE` (or `0`) otherwise.

Description **Visvs** determines if the object specified by the reference number, *obj_ref*, is a vdata within the vgroup identified by the parameter *vgroup_id*.

FORTRAN

```
integer function vfisvs(vgroup_id, obj_ref)

integer vgroup_id, obj_ref
```

Vlone/vflone

int32 Vlone(int32 *file_id*, int32 *ref_array*[], int32 *max_refs*)

file_id	IN:	File identifier returned by **Hopen**
ref_array	OUT:	Array of reference numbers
max_refs	IN:	Maximum number of lone vgroups to be retrieved

Purpose Retrieves the reference numbers of lone vgroups, i.e., vgroups that are at the top of the grouping hierarchy, in a file.

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

Description **Vlone** retrieves the reference numbers of lone vgroups in the file identified by the parameter *file_id*. Although **Vlone** returns the total number of lone vgroups in the file, only at most *max_refs* reference numbers are retrieved and stored in the buffer *ref_array*. The array must have at least *max_refs* elements.

An array size of 65,000 integers for *ref_array* is more than adequate if the user chooses to declare the array statically. However, the preferred method is to dynamically allocate memory instead; first call **Vlone** with a value of `0` for *max_refs*, and then use the returned value to allocate memory for *ref_array* before calling **Vlone** again.

FORTRAN

```
integer function vflone(file_id, ref_array, max_refs)

integer file_id, ref_array(*), max_refs
```

Vnattrs/vfnatts

intn Vnattrs(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Returns the number of attributes assigned to a vgroup.

Return value Returns the total number of attributes assigned to the specified vgroups if successful and `FAIL` (or `-1`) otherwise.

Description **Vnattrs** gets the number of attributes assigned to the vgroup identified by the parameter *vgroup_id*.

FORTRAN

```
integer function vfnatts(vgroup_id)

integer vgroup_id
```

Vnrefs/vnrefs

int32 Vnrefs(int32 *vgroup_id*, int32 *tag_type*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
tag_type	IN:	Type of the tag

Purpose Returns the number of tags of a given tag type in a vgroup.

Return value Returns 0 or the total number of tags if successful and `FAIL` (or `-1`) otherwise.

Description **Vnrefs** returns 0 or the number of tags having the type specified by the parameter *tag_type* in the vgroup identified by the parameter *vgroup_id*.

See Appendix A, *Reserved HDF Tags*, in the *HDF User's Guide*, for a discussion of tag types.

FORTRAN

```
integer function vnrefs(vgroup_id, tag_type)

integer vgroup_id, tag_type
```

Vntagrefs/vfntr

int32 Vntagrefs(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose
Returns the number of objects in a vgroup.

Return value
Returns `0` or a positive number representing the number of HDF objects linked to the vgroup if successful or `FAIL` (or `-1`) otherwise.

Description
Vntagrefs returns the number of objects in a vgroup identified by the parameter *vgroup_id*.

Vntagrefs is used together with **Vgettagrefs**, or with **Vgettagref** to look at the data objects linked to a given vgroup.

FORTRAN

```
integer function vfntr(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 **Vsetattr** attaches an attribute to the vgroup identified by the parameter *vgroup_id*. The attribute name is specified by the parameter *attr_name* and the attribute data type is specified by the parameter *data_type*. The values of the attribute are specified by the parameter *values*, and the number of values in the attribute is specified by the parameter *count*. Refer to Table 1A in Section I of this manual for a listing of all valid data types.

If the attribute already exists, the new values will replace the current ones, provided the data type and the number of attribute values have not been changed. If either the data type or the order have been changed, **Vsetattr** will return `FAIL` (or `-1`).

FORTRAN

```
integer vfsnatt(vgroup_id, attr_name, data_type, count, values)

integer vgroup_id, data_type, count

<valid numeric data 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*, char **vgroup_class*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_class	IN:	Class name of a vgroup

Purpose Sets the class name of a vgroup.

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

Description **Vsetclass** sets the class name specified by the parameter *vgroup_class* to the vgroup identified by the parameter *vgroup_id*.

A vgroup initially has 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.

The class name is limited to `VSNAMELENMAX` (or `64`) characters.

FORTRAN

```
integer function vfscls(vgroup_id, vgroup_class)

integer vgroup_id

character*(*) vgroup_class
```

Vsetname/vfsnam

int32 Vsetname(int32 *vgroup_id*, char **vgroup_name*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**
vgroup_name	IN:	Name of a vgroup

Purpose Sets the name of a vgroup.

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

Description **Vsetname** sets the name specified by the parameter *vgroup_name* for the vgroup identified by the parameter *vgroup_id*.

A vgroup initially has a name of `NULL`, and may be renamed more than once during the scope of the vgroup identifier (*vgroup_id*). 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 name length is limited to `VSNAMELENMAX` (or `64`) characters.

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 vdata and/or vgroup interface.

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

Description **Vstart** initializes the vdata and/or vgroup interfaces for the file identified by the parameter *file_id*.

Vstart 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
```

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
ref_array	IN:	Array of reference numbers
n_objects	IN:	Number of data objects to be stored
vgroup_name	IN:	Name of the vgroup
vgroup_class	IN:	Class of the vgroup

Purpose Creates a vgroup.

Return value Returns the reference number of the newly-created vgroup if successful, `FAIL` (or `-1`) otherwise.

Description **VHmakegroup** creates a vgroup with the name specified by the parameter *vgroup_name* and the class name specified by the parameter *vgroup_class* in the file identified by the parameter *file_id*. The routine inserts *n_objects* objects into the vgroup. The tag and reference numbers of the objects to be inserted are specified in the arrays *tag_array* and *ref_array*.

Creating empty vgroups with **VHmakegroup** is allowed. **VHmakegroup** does not check if the tag/reference number pair is valid, or if the corresponding data object exists. However, all of the tag/reference number pairs must be unique.

Vstart must precede any calls to **VHmakegroup**. It is not necessary, however, to call **Vattach** or **Vdetach** in conjunction with **VHmakegroup**.

The elements in the arrays *tag_array* and *ref_array* are the matching tag/reference number pairs of the objects to be inserted, that means *tag_array*[`0`] and *ref_array*[`0`] refer to one data object, and *tag_array*[`1`] and *ref_array*[`1`] to another, etc.

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(*)
```

VQueryref/vqref

int32 VQueryref(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Returns the reference number of a vgroup.

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

Description **VQueryref** returns the reference number of the vgroup identified by the parameter *vgroup_id*.

FORTRAN

```
integer function vqref(vgroup_id)

integer vgroup_id
```

VQuerytag/vqtag

int32 VQuerytag(int32 *vgroup_id*)

vgroup_id	IN:	Vgroup identifier returned by **Vattach**

Purpose Returns the tag of a vgroup.

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

Description **VQuerytag** returns the tag of the vgroup identified by the parameter *vgroup_id*.

FORTRAN

```
integer function vqtag(vgroup_id)

integer vgroup_id
```

VFfieldesize/vffesiz

int32 VFfieldesize(int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Vdata field index

Purpose Returns the size, as stored on disk, of a vdata field.

Return value Returns the vdata field size if successful and `FAIL` (or `-1`) otherwise.

Description **VFfieldesize** returns the size, as stored on disk, of a vdata field identified by the parameter *field_index* in the vdata identified by the parameter *vdata_id*.

The value of the parameter *field_index* ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by **VFnfields** function.

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 identifier returned by **VSattach**
field_index	IN:	Vdata field index

Purpose Returns the size, as stored in memory, of a vdata field.

Return value Returns the vdata field size if successful and `FAIL` (or `-1`) otherwise.

Description **VFfieldisize** returns the size, as stored in memory, of a vdata field identified by the parameter *field_index* in the vdata identified by the parameter *vdata_id*.

The value of the parameter *field_index* ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by **VFnfields** function.

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 identifier returned by **VSattach**
field_index	IN:	Vdata field index

Purpose Returns the name of a vdata field.

Return value Returns a pointer to the vdata field name if successful and `NULL` otherwise. The FORTRAN-77 version of this routine, **vffname**, returns `SUCCEED` (or `0`) or `FAIL` (or `-1`).

Description **VFfieldname** returns the name of the vdata field identified by the parameter *field_index* in the vdata identified by the parameter *vdata_id*.

The value of the parameter *field_index* ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by **VFnfields** function.

The FORTRAN-77 version of this routine, **vffname**, returns the field name in the parameter *fname*.

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 identifier returned by **VSattach**
field_index	IN:	Vdata field index

Purpose Returns the order of a vdata field.

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

Description **VFfieldorder** returns the order of the vdata field identified by its index, *field_index*, in the vdata identified by the parameter *vdata_id*.

The value of the parameter *field_index* ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by **VFnfields** function.

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 identifier returned by **VSattach**
field_index	IN:	Vdata field index

Purpose Returns the data type of a vdata field.

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

Description **VFfieldtype** returns the data type of the vdata field identified by its index, *field_index*, in the vdata identified by the parameter *vdata_id*.

The value of the parameter *field_index* ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by **VFnfields** function.

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 a vdata.

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

Description **VFnfields** returns the total number of fields in the vdata identified by the parameter *vdata_id*.

FORTRAN

```
integer function vfnflds(vdata_id)

integer vdata_id
```

VSQuerycount/vsqfnelt

intn VSQuerycount(int32 *vdata_id*, int32 **n_records*)

vdata_id	IN:	Vdata access identifier returned by **VSattach**
n_records	OUT:	Number of records in the vdata

Purpose Retrieves the number of records in a vdata.

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

Description **VSQuerycount** retrieves the number of records in the vdata identified by *vdata_id* in the parameter *n_records*.

FORTRAN

```
integer function vsqfnelt(vdata_id, n_records)

integer vdata_id, n_records
```

VSQueryfields/vsqfflds

intn VSQueryfields(int32 *vdata_id*, char **field_name_list*)

vdata_id	IN:	Vdata access identifier returned by **VSattach**
field_name_list	OUT:	List of field names

Purpose Retrieves the names of the fields in a vdata.

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

Description **VSQueryfields** retrieves the names of the fields in the vdata identified by the parameter *vdata_id* into the parameter *field_name_list*.

The parameter *field_name_list* is a comma-separated list of the fields in the vdata. (i.e., “PX,PY,PZ” in C and ’PX,PY,PZ’ in Fortran).

FORTRAN

```
integer function vsqfflds(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```

VSQueryinterlace/vsqfintr

intn VSQueryinterlace(int32 *vdata_id*, int32 **interlace_mode*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
interlace_mode	OUT:	Interlace mode

Purpose Retrieves the interlace mode of the vdata.

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

Description **VSQueryinterlace** retrieves the interlace mode of the vdata identified by the parameter *vdata_id* into the parameter *interlace_mode*.

Valid values for *interlace_mode* are `FULL_INTERLACE` (or `0`) and `NO_INTERLACE` (or `1`).

FORTRAN

```
integer function vsqfintr(vdata_id, interlace_mode)

integer vdata_id, interlace_mode
```

VSQueryname/vsqfname

intn VSQueryname(int32 *vdata_id*, char **vdata_name*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_name	OUT:	Name of the vdata

Purpose Retrieves the name of a vdata.

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

Description **VSQueryname** retrieves the name of the vdata identified by the parameter *vdata_id* into the buffer *vdata_name*.

The buffer *vdata_name* should be set to at least `VSNAMELENMAX` bytes. `VSNAMELENMAX` is defined by the HDF library.

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 identifier returned by **VSattach**

Purpose Returns the reference number of a vdata.

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

Description **VSQueryref** returns the reference number of the vdata identified by the parameter *vdata_id*.

FORTRAN

```
integer function vsqref(vdata_id)

integer vdata_id
```

VSQuerytag/vsqtag

int32 VSQuerytag(int32 *vdata_id*)

vdata_id IN: Vdata identifier returned by **VSattach**

Purpose Returns the tag of the specified vdata.

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

Description Returns the tag of the vdata identified by the parameter *vdata_id*.

FORTRAN

```
integer function vsqtag(vdata_id)

integer vdata_id
```

VSQueryvsize/vsqfvsiz

intn VSQueryvsize(int32 *vdata_id*, int32 **vdata_size*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_size	OUT:	Size of the vdata record

Purpose Retrieves the size of a record in a vdata.

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

Description **VSQueryvsize** retrieves the size, in bytes, of a record in the vdata identified by the parameter *vdata_id* into the parameter *vdata_size*. The returned size value is machine dependent.

FORTRAN

```
integer function vsqfvsiz(vdata_id, vdata_size)

integer vdata_id, vdata_size
```

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 containing the records to be stored
n_records	IN:	Number of records to be stored
data_type	IN:	Type of data to be stored
vdata_name	IN:	Name of the vdata to be created
vdata_class	IN	Class of the vdata to be created

Purpose Creates and writes to a single-field vdata.

Return value Returns reference number of the newly-created vdata if successful, and `FAIL` (or `-1`) otherwise.

Description **VHstoredata** creates a single-field vdata in the file, *file_id*, and stores data from the buffer *buf* in it. Vdata name, class name and data type are specified by the parameters *vdata_name*, *vdata_class*, and *data_type*, respectively. Number of records in a vdata is specified by the parameter *n_records*. Field name is specified by the parameter *fieldname*.

Vstart must precede **VHstoredata**. It is not necessary, however, to call **VSattach** or **VSdetach** in conjunction with **VHstoredata**.

This routine provides a high-level method for creating single-order, single-field vdatas.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (**vhfsd**) and the other for 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*(*) vdata_name, vdata_class, fieldname

<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*(*) vdata_name, vdata_class, fieldname

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
buf	IN:	Buffer containing the records to be stored
n_records	IN:	Number of records to be stored
data_type	IN:	Type of data to be stored
vdata_name	IN:	Name of the vdata to be created
vdata_class	IN:	Class of the vdata to be created
order	IN:	Field order

Purpose Creates and writes to a multi-order, single-field vdata.

Return value Returns the reference number of the newly created vdata if successful, and `FAIL` (or `-1`) otherwise.

Description **VHstoredatam** creates a vdata with the name specified by the parameter *vdata_name* and a class name specified by the parameter *vdata_class* in the file identified by the parameter *file_id*. The data type of the vdata is specified by the parameter *data_type*. The vdata contains one field with the name specified by the parameter *fieldname*. The order of the field, *order,* indicates the number of vdata values stored per field. The vdata contains the number of records specified by the parameter *n_records*. The *buf* parameter should contain *n_records* records that will be stored in the vdata.

Vstart must precede **VHstoredatam**. It is not necessary, however, to call **VSattach** or **VSdetach** in conjunction with **VHstoredatam**.

This routine provides a high-level method for creating multi-order, single-field vdatas.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (**vhfsdm**) and the other for character data (**vhfscdm**).

FORTRAN

```
integer function vhfsdm(file_id, fieldname, buf, n_records,

integer file_id, n_records, data_type, order

character*(*) vdata_name, vdata_class, fieldname

<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*(*) vdata_name, vdata_class, fieldname

character*(*) buf
```

VSappendable/vsapp (Obsolete)

int32 VSappendable(int32 *vdata_id*, int32 *block_size*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
block_size	IN:	Standard block size of appended data

Purpose Makes it possible to append to a vdata.

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

Description The HDF library makes all vdatas appendable upon creation. Therefore, this routine has been made 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 of the vdata
access	IN:	Access mode

Purpose Attaches to an existing vdata or creates a new vdata.

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

Description **VSattach** attaches to the vdata identified by the reference number, *vdata_ref*, in the file identified by the parameter *file_id*. Access to the vdata is specified by the parameter *access*. **VSattach** returns an identifier to the vdata, through which all further operations on that vdata are carried out.

An existing vdata may be multiply-attached for reads. Only one attach with write access to a vdata is allowed.

The default interlace mode for a new vdata is `FULL_INTERLACE` (or `0`). This may be changed using **VSsetinterlace**.

The value of the parameter *vdata_ref* may be -1. This is used to create a new vdata.

Valid values for *access* are "`r`" for read access and "`w`" for write access.

If *access* is "`r`", then *vdata_ref* must be the valid reference number of an existing vdata returned from any of the vdata and vgroup search routines (e.g., **Vgetnext** or **VSgetid**). It is an error to attach to a 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 or -1. An existing vdata is generally attached with "`w`" access to replace part of its data, or to append new data to it.

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,* intn *attr_index*, char **attr_name*, int32 **data_type*, int32 **count*, int32 **size*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Index of the field
attr_index	IN:	Index of the attribute
attr_name	OUT:	Name of the attribute
data_type	OUT:	Data type of the attribute
count	OUT:	Attribute value count
size	OUT:	Size of the attribute

Purpose Retrieves attribute information of a vdata or a vdata field.

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

Description **VSattrinfo** gets information on the attribute attached to the vdata, *vdata_id*, or to the vdata field. Vdata field is specified by its index, *field_index*. Attribute is specified by its index, *attr_index*. The attribute name is returned into the parameter *attr_name*, the data type is returned into the parameter *data_type*, the number of values of the attribute is returned into the parameter *count*, and the size of the attribute is returned into the parameter *size*.

The parameter *field_index* in **VSattrinfo** is the same as the parameter *field_index* in **VSsetattr**. It can be set to either an integer field index for the vdata field attribute, or `_HDF_VDATA` (or `-1`) to specify the vdata attribute.

In C the values of the parameters *attr_name*, *data_type*, *count* and *size* can be set to `NULL` if the information returned by these parameters is not needed.

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
```

VSdelete/vsfdlte

int32 VSdelete(int32 *file_id*, int32 *vdata_ref*)

file_id	IN:	File identifier returned by **Hopen**
vdata_ref	IN:	Vdata reference number returned by **VSattach**

Purpose Remove a vdata from a file.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) if not successful.

Description **VSdelete** removes the vdata identified by the parameter *vdata_ref* from the file identified by the parameter *file_id*.

FORTRAN

```
integer function vsfdlte(file_id, vdata_ref)

integer file_id, vdata_ref
```

VSdetach/vsfdtch

int32 VSdetach(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **VSattach**

Purpose Detaches from the current vdata, terminating further access to that vdata.

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

Description **VSdetach** detaches from the vdata identified by the parameter *vdata_id* and updates the vdata information in the file if there are any changes. All memory used for that vdata is freed.

The *vdata_id* identifier 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 identifier returned by **VSattach**

Purpose Determines the number of records in a vdata.

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

Description **VSelts** returns the number of records in the vdata identified by *vdata_id*.

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 identifier returned by **VSattach**
fieldname	IN:	Name of the 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** defines a field with the name specified by the parameter *fieldname*, of the data type specified by the parameter *data_type*, of the order specified by the parameter *order*, and within the vdata identified by the parameter *vdata_id*.

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 vdata fields may not be modified or deleted.

There are certain field names the HDF library recognizes as predefined. A list of these predefined field types can be found in the HDF User's Guide.

A field is defined by its name (*fieldname*), its type (*data_type*) and its order (*order*). A fieldname is any sequence of characters. By convention, fieldnames are usually a 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 data types listed in Table 1A in Section I of this manual.

The order of a field is the number of components in that field. A field containing the value of a simple variable, such a time or pressure, would have an order of 1. Compound variables have an order greater than 1. For example, a field containing the values associated with a variable for velocity in three dimensions would have an order of 3.

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 **field_name_list*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_name_list	IN:	List of field names

Purpose Checks to see if certain fields exist in the current vdata.

Return value Returns a value of 1 if all field(s) exist and `FAIL` (or `-1`) otherwise.

Description **VSfexist** checks if all fields with the names specified in the parameter *field_name_list* exist in the vdata identified by the parameter *vdata_id*.

The parameter *field_name_list* is a string of comma-separated fieldnames (e.g., “PX,PY,PZ” in C and ’PX,PY,PZ’ in Fortran).

FORTRAN

```
integer function vsfex(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```

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 Returns the reference number of a vdata, given its name.

Return value Returns the vdata reference number if successful and `0` if the vdata is not found or an error occurs.

Description **VSfind** returns the reference number of the vdata with the name specified by the parameter *vdata_name* in the file specified by the parameter *file_id*. 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
```

VSfindattr/vsffdat

intn VSfindattr(int32 *vdata_id*, int32 *field_index*, char **attr_name*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Field index
attr_name	IN:	Attribute name

Purpose Returns the index of an attribute of a vdata or vdata field.

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

Description **VSfindattr** returns the index of the attribute with the name specified by the parameter *attr_name* in the vdata identified by the parameter *vdata_id*.

To return the index of the attribute attached to the vdata , set the value of the parameter *field_index* to `_HDF_VDATA` (or `-1`). To return the index of the attribute of a field in the vdata , set the value of the parameter *field_index* to the field index. Valid values of *field_index* range from 0 to the total number of the vdata fields - 1. The number of the vdata fields is returned by **VFnfields**.

FORTRAN

```
integer function vsffdat(vdata_id, field_index, attr_name)

integer vdata_id, field_index

character*(*) attr_name
```

VSfindclass/vffcls

int32 VSfindclass(int32 *file_id*, 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 with a given vdata class name

Return value Returns the reference number of the vdata if successful and `0` if the vdata is not found or an error occurs.

Description **VSfindclass** returns the reference number of the vdata with the class name specified by the parameter *vdata_class* in the file identified by the parameter *file_id*.

FORTRAN

```
integer function vffcls(vdata_id, vdata_class)

integer vdata_id

character*(*) vdata_class
```

VSfindex/vsffidx

intn VSfindex(int32 *vdata_id*, char **fieldname*, int32 **field_index*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
fieldname	IN:	Name of the field
field_index	OUT:	Index of the field

Purpose Retrieves the index of a field within a vdata.

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

Description **VSfindex** retrieves the index, *field_index*, of the field with a name specified by the parameter *fieldname*, within the vdata identified by the parameter *vdata_id*.

FORTRAN

```
integer function vsffidx(vdata_id, fieldname, field_index)

integer vdata_id, field_index

character*(*) fieldname
```

VSfnattrs/vsffnas

int32 VSfnattrs (int32 *vdata_id*, int32 *field_index*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Index of the field

Purpose Returns the number of attributes attached to a vdata *or* the number of attributes attached to a vdata field.

Return value Returns the number of attributes assigned to this vdata or its fields when successful, and `FAIL` (or `-1`) otherwise.

Description **VSfnattrs** returns the number of attributes attached to a vdata specified by the parameter *vdata_id*, or the number of attributes attached to a vdata field, specified by the field index, *field_index.*

To return the number of attributes attached to the vdata , set the value of *field_index* to `_HDF_VDATA` (or `-1`). To return the number of attributes of a field in the vdata , set the value of *field_index* to the field index. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields is returned by **VFnfields**.

VSfnattrs is different from the **VSnattrs** routine, which returns the number of attributes of the specified vdata *and* the fields contained in it.

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 **field_name_list*, VOIDP *bufptrs*[])

vdata_id	IN:	Vdata identifier returned by **VSattach**
action	IN:	Action to be performed
fields_in_buf	IN:	Names of the fields in *buf*
buf	IN/OUT:	Buffer containing the values of the packed fields to write to or read from the vdata
buf_size	IN:	Buffer size in bytes
n_records	IN:	Number of records to pack or unpack
field_name_list	IN:	Names of the fields to be packed or unpacked
bufptrs	IN/OUT:	Array of pointers to the field buffers

Purpose Packs field data into a buffer or unpacks buffered field data into vdata field(s) for fully interlaced fields.

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

Description **VSfpack** packs or unpacks the field(s) listed in the parameter *field_name_list* to or from the buffer *buf* according to the specified action in the parameter *action*.

Valid values for *action* are `_HDF_VSPACK` (or `0`) which packs field values from *bufptrs* (the field buffers) to *buf*, or `_HDF_VSUNPACK` (or `1`) which unpacks vdata field values from *buf* into *bufptrs*.

When **VSfpack** is called to pack field values into *buf*, *fields_in_buf* must list all fields of the vdata. When **VSfpack** is called to unpack field values, *fields_in_buf* may be a subset of the vdata fields. To specify all vdata fields in *fields_in_buf*, `NULL` can be used in C and a blank character (" ") in Fortran.

The name(s) of the field(s) to be packed or unpacked are specified by the *field_name_list*. In C, the names in the parameter *field_name_list* can be a subset of or all field names listed in *fields_in_buf*. To specify all vdata fields, `NULL` can be used in C.

The FORTRAN-77 versions of this routine can pack or unpack only one field at a time. Therefore, *field_name_list* will contain the name of the field that will be packed or unpacked.

The calling program must allocate sufficient space for *buf* to hold all of the packed fields. The size of the *buf* buffer should be at least *n_records* * (the total size of all fields specified in *fields_in_buf*).

Note that there are two FORTRAN-77 versions of this routine: **vsfnpak** to pack or unpack a numeric field and **vsfcpak** to pack or unpack a character field.

Refer to the HDF User's Guide for an example on how to use this routine.

FORTRAN

```
integer function vsfnpak(vdata_id, action, fields_in_buf, buf,
                  buf_size, n_records, field_name_list, bufptrs)

integer vdata_id, action, buf(*), buf_size, n_records

character*(*) fields_in_buf, field_name_list

<valid numeric data type> bufptrs(*)


integer function vsfcpak(vdata_id, action, fields_in_buf, buf,
                  buf_size, n_records, field_name_list, bufptrs)

integer vdata_id, action, buf(*), buf_size, n_records

character*(*) fields_in_buf, field_name_list, bufptrs(*)
```

VSgetattr/vsfgnat/vsfgcat

intn VSgetattr(int32 *vdata_id*, intn *field_index,* int32 *attr_index*, VOIDP *values*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Index of the field
attr_index	IN:	Index of the attribute
values	OUT:	Buffer for the attribute values

Purpose Retrieves the attribute values of a vdata or vdata field.

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

Description **VSgetattr** retrieves the attribute values of the vdata identified by the parameter *vdata_id* or the vdata field specified by the field index, *field_index*, into the buffer *values*.

If *field_index* is set to `_HDF_VDATA` (or `-1`), the value of the attribute attached to the vdata is returned. If *field_index* is set to the field index, attribute attached to a vdata field is returned. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields is returned by **VFnfields**

Attribute to be retrieved is specified by its index, *attr_index*. Index is a nonnegative integer less than the total number of the vdata or vdata field attributes. Use **VSfnattrs** to find the number of the vdata or vdata field attributes.

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, field_index, attr_index

character*(*) values
```

VSgetblockinfo/vsfgetblinfo

intn VSgetblockinfo(int32 *vdata_id*, int32 **block_size*, int32 **num_blocks*)

vdata_id	IN:	Vdata identifier
block_size	OUT:	Block size in bytes
num_blocks	OUT:	Number of linked blocks

Purpose Retrieves the block size and the number of blocks for a linked-block Vdata element.

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

Description **VSgetblockinfo** retrieves the block size and the number of linked blocks for a linked-block Vdata element.

If no response is desired for either returned value, *block_size* and *num_blocks* may be set to `NULL`.

FORTRAN

```
integer function vsfgetblinfo(vdata_id, block_size, num_blocks)

integer vdata_id, num_blocks, block_size
```

VSgetclass/vsfgcls

int32 VSgetclass(int32 *vdata_id*, char **vdata_class*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_class	OUT:	Vdata class name

Purpose Retrieves the vdata class name, if any.

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

Description **VSgetclass** retrieves the class name of the vdata identified by the parameter *vdata_id* and places it in the buffer *vdata_class*.

Space for the buffer *vdata_class* must be allocated by the calling program before **VSgetclass** is called. The maximum length of the class name is defined by the macro `VSNAMELENMAX (or 64)`.

FORTRAN

```
integer function vsfgcls(vdata_id, vdata_class)

integer vdata_id

character*(*) vdata_class
```

VSgetfields/vsfgfld

int32 VSgetfields(int32 *vdata_id*, char **field_name_list*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_name_list	OUT:	Field name list

Purpose Retrieves the field names of all of the fields in a Vdata.

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

Description **VSgetfields** retrieves the names of the fields in the Vdata identified by the parameter *vdata_id* into the buffer *field_name_list*.

The parameter *field_name_list* is a character string containing a comma-separated list of names (e.g., “PX,PY,PZ” in C or ‘PX,PY,PZ’ in Fortran).

The user must allocate the memory space for the buffer *field_name_list* before calling **VSgetfields**.

If the Vdata does not have any fields, a null string is returned in the parameter *field_name_list*.

The maximum length of a Vdata name is defined by `VSNAMELENMAX` (or 64).

FORTRAN

```
integer function vsfgfld(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```

VSgetid/vsfgid

int32 VSgetid(int32 *file_id*, int32 *vdata_ref*)

file_id	IN:	File identifier returned by **Hopen**
vdata_ref	IN:	Vdata reference number

Purpose Sequentially searches through a file for vdatas.

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

Description **VSgetid** sequentially searches through a file identified by the parameter *file_id* 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. Searching past the last vdata in a file will result in an error condition.

To initiate a search, this routine must be called with the value of *vdata_ref* equal to -1. Doing so returns the reference number of the first vdata in the file.

FORTRAN

```
integer function vsfgid(file_id, vdata_ref)

integer file_id, vdata_ref
```

VSgetinterlace/vsfgint

int32 VSgetinterlace(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **VSattach**

Purpose Returns the interlace mode of a vdata.

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

Description **VSgetinterlace** returns the interlace mode of the vdata identified by the parameter *vdata_id*.

FORTRAN

```
integer function vsfgint(vdata_id)

integer vdata_id
```

VSgetname/vsfgnam

int32 VSgetname(int32 *vdata_id*, char **vdata_name*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_name	OUT:	Vdata name

Purpose Retrieves the name of a vdata.

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

Description **VSgetname** retrieves the name of the vdata identified by the parameter *vdata_id* into the buffer *vdata_name*.

The user must allocate the memory space for the buffer *vdata_name* before calling **VSgetname**. If the vdata does not have a name, a null string is returned in the parameter *vdata_name* . The maximum length of a vdata name is defined by `VSNAMELENMAX` (or 64)

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 returned by **VSattach**

Purpose Returns the version number of a vdata.

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

Description **VSgetversion** returns the version number of the vdata identified by the parameter *vdata_id*. There are three valid version numbers: `VSET_OLD_VERSION` (or `2`), `VSET_VERSION` (or `3`), and `VSET_NEW_VERSION` (or `4`).

`VSET_OLD_VERSION` is returned when the vdata is of a version that corresponds to an HDF library version before version 3.2.

`VSET_VERSION` is returned when the vdata is of a version that corresponds to an HDF library version between versions 3.2 and 4.0 release 2.

`VSET_NEW_VERSION` is returned when the vdata is of the version that corresponds to an HDF library version of version 4.1 release 1 or higher.

FORTRAN

```
integer vsgver(vdata_id)

integer vdata_id
```

VSinquire/vsfinq

intn VSinquire(int32 *vdata_id*, int32 **n_records*, int32 **interlace_mode*, char **field_name_list*, int32 **vdata_size*, char **vdata_name*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
n_records	OUT:	Number of records
interlace_mode	OUT:	Interlace mode of the data
field_name_list	OUT:	List of field names
vdata_size	OUT:	Size of a record
vdata_name	OUT:	Name of the vdata

Purpose Retrieves general information about a vdata.

Return value Returns `SUCCEED` (or `0`) if successful and `FAIL` (or `-1`) if it is unable to return any of the requested information.

Description **VSinquire** retrieves the number of records, the interlace mode of the data, the name of the fields, the size, and the name of the vdata, *vdata_id*, and stores them in the parameters *n_records*, *interlace_mode*, *field_name_list*, *vdata_size*, and *vdata_name*, respectively. In C, if any of the output parameters are `NULL`, the corresponding information will not be retrieved. Refer to the Reference Manual pages on **VSelts**, **VSgetfields**, **VSgetinterlace**, **VSsizeof** and **VSgetname** for other routines that can be used to retrieve specific information.

Possible returned values for *interlace_mode* are `FULL_INTERLACE` (or `0`) and `NO_INTERLACE` (or `1`). The returned value of *vdata_size* is the number of bytes in a record and is machine-dependent.

The parameter *field_name_list* is a character string that contains the names of all the vdata fields, separated by commas. (e.g., “PX,PY,PZ” in C and ’PX,PY,PZ’ in Fortran).

The user must allocate the memory space for the buffer *vdata_name* before calling **VSinquire**. If the vdata does not have a name, a null string is returned in the parameter *vdata_name* . The maximum length of a vdata name is defined by `VSNAMELENMAX` (or 64)

Note **VSinquire** will return `FAIL` if it is called before **VSdefine** and **VSsetfield** on the same vdata.

FORTRAN

```
integer function vsfinq(vdata_id, n_records, interlace,
                  field_name_list, vdata_size, vdata_name)

integer vdata_id, n_records, interlace, vdata_size

character*(*) field_name_list, vdata_name
```

VSisattr/vsfisat

intn VSisattr(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **VSattach**

Purpose Determines whether a vdata is an attribute.

Return value Returns `TRUE` (or `1`) if the vdata is an attribute, and `FALSE` (or `0`) otherwise.

Description **VSisattr** determines whether the vdata identified by the parameter *vdata_id* is an attribute.

As attributes are stored by the HDF library as vdatas, a means of testing whether or not a particular vdata is an attribute is needed, and is provided by this 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:	Array of reference numbers
max_refs	IN:	Maximum number of lone vdatas to be retrieved

Purpose Retrieves the reference numbers of all lone vdatas, i.e., vdatas that are not grouped with other objects, in a file.

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

Description **VSlone** retrieves the reference numbers of lone vgroups in the file identified by the parameter *file_id*. Although **VSlone** returns the number of lone vdatas in the file, only at most *max_refs* reference numbers are retrieved and stored in the buffer *ref_array.* The array must have at least *max_refs* elements.

An array size of 65,000 integers for *ref_array* is more than adequate if the user chooses to declare the array statically. However, the preferred method is to dynamically allocate memory instead; first call **VSlone** with a value of 0 for *max_refs* to return the total number of lone vdatas, then use the returned value to allocate memory for *ref_array* before calling **VSlone** again.

FORTRAN

```
integer function vsflone(file_id, ref_array, max_refs)

integer file_id, ref_array(*), max_refs
```

VSnattrs/vsfnats

intn VSnattrs(int32 *vdata_id*)

vdata_id	IN:	Vdata identifier returned by **VSattach**

Purpose Returns the total number of attributes of a vdata and of its fields.

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

Description **VSnattrs** returns the total number of attributes of the vdata, *vdata_id*, and of its fields.

VSnattrs is different from the **VSfnattrs** routine, which returns the number of attributes of a specified vdata *or* of a field contained in a specified vdata.

FORTRAN

```
integer function vsfnats(vdata_id)

integer vdata_id
```

VSread/vsfrd/vsfrdc/vsfread

int32 VSread(int32 *vdata_id*, uint8 **databuf*, int32 *n_records*, int32 *interlace_mode*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
databuf	OUT:	Buffer to store the retrieved data
n_records	IN:	Number of records to be retrieved
interlace_mode	IN:	Interlace mode of the data to be stored in the buffer

Purpose Retrieves data from a vdata.

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

Description **VSread** reads *n_records* records from the vdata identified by the parameter *vdata_id* and stores the data in the buffer *databuf* using the interlace mode specified by the parameter *interlace_mode*.

The user can specify the fields and the order in which they are to be read by calling **VSsetfields** prior to reading. **VSread** stores the requested fields in *databuf* in the specified order.

Valid values for *interlace_mode* are `FULL_INTERLACE` (or `1`) and `NO_INTERLACE` (or `0`). Selecting `FULL_INTERLACE` causes *databuf* to be filled by record and is recommended for speed and efficiency. Specifying `NO_INTERLACE` causes *databuf* to be filled by field, i.e., all values of a field in *n_records* records are filled before moving to the next field. Note that the default interlace mode of the buffer is `FULL_INTERLACE`.

As the data is stored contiguously in the vdata, **VSfpack** should be used to unpack the fields after reading. Refer to the discussion of **VSfpack** in the HDF User’s Guide for more information.

Note that there are three FORTRAN-77 versions of this routine: **vsfrd** is for buffered numeric data, **vsfrdc** is for buffered character data and **vsfread** is for generic packed data.

See the notes regarding the potential performance impact of appendable data sets in the *HDF User’s Guide* Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance."

FORTRAN On Windows systems, this function is available only for an integer data buffer.

```
integer function vsfrd(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

<valid numeric data type> databuf(*)
```

```
integer function vsfrdc(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

character*(*) databuf


integer function vsfread(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

integer databuf(*)
```

VSseek/vsfseek

int32 VSseek(int32 *vdata_id*, int32 *record_pos*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
record_pos	IN:	Position of the record

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 **VSseek** moves the access pointer within the vdata identified by the parameter *vdata_id* to the position of the record specified by the parameter *record_pos*. The next call to **VSread** or **VSwrite** will read from or write to the record where the access pointer has been moved to.

The value of *record_pos* is zero-based. For example, to seek to the third record in the vdata, set *record_pos* to 2. The first record position is specified by specifying a *record_pos* value of 0. Each seek is constrained to a record boundary within the vdata.

See the notes regarding the potential performance impact of appendable data sets in the *HDF User's Guide* Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance."

FORTRAN

```
integer function vsfseek(vdata_id, record_pos)

integer vdata_id, record_pos
```

VSsetattr/vsfsnat/vsfscat

intn VSsetattr(int32 *vdata_id*, int32 *field_index*, char **attr_name*, int32 *data_type*, int32 *count*, VOIDP *values*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_index	IN:	Index of the field
attr_name	IN:	Name of the attribute
data_type	IN:	Data type of the attribute
count	IN:	Number of attribute values
values	IN:	Buffer containing the attribute values

Purpose Sets an attribute of a vdata or a vdata field.

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

Description **VSsetattr** defines an attribute that has the name specified by the parameter *attr_name*, the data type specified by the parameter *data_type*, and the number of values specified by the parameter *count*, and that contains the values specified in the parameter *values*. The attribute is set for either the vdata or a vdata field depending on the value of the parameter *field_index*.

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 a value of `FAIL` (or `-1`) to be returned.

If *field_index* value is set to `_HDF_VDATA` (or `-1`), the attribute will be set for the vdata. If *field_index* is set to the field index, attribute will be set for the vdata field. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields can be obtained using **VFnfields**.

The value of the parameter *data_type* can be any one of the data types listed in Table 1A in Section I of this manual.

FORTRAN

```
integer function vsfsnat(vdata_id, field_index, attr_name,
                    data_type, count, values)

integer vdata_id, field_index, data_type, count, values(*)

character*(*) attr_name


integer function vsfscat(vdata_id, field_index, attr_name,
                    data_type, count, values)

integer vdata_id, field_index, data_type, count

character*(*) attr_name, values(*)
```

VSsetblocksize/vsfsetblsz

intn VSsetblocksize(int32 *vdata_id*, int32 *block_size*)

vdata_id	IN:	Vdata identifier
block_size	IN:	Size of each block in bytes

Purpose Sets linked-block Vdata element block size.

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

Description **VSsetblocksize** sets the block size for linked-block elements that will be used to store Vdatas.

The default block size is `HDF_APPENDABLE_BLOCK_LEN`, which is set to `4096` in the library as it is distributed. **VSsetblocksize** modifies that default value and must be called before the first write to the Vdata. Once the linked-block element is created, the block size cannot be changed.

The following note may be of interest to users who must pay *very* close attention to performance issues: **VSsetblocksize** sets the block size only for blocks following the first block. The first block can be arbitrarily large; the library continues to write to it until it encounters an obstacle, at which point the linked block mechanism is invoked. For example, a Vdata A that is the last item in a file can continue to grow, simply extending the file. If a new Vdata B is then written, that new object is (normally) placed at the end of the file, blocking off extension of the prior Vdata, A. At this point, new writes to A will write data to linked blocks per the *block_size* and *num_blocks* settings.

FORTRAN

```
integer function vsfsetblsz(vdata_id, block_size)

integer vdata_id, block_size
```

VSsetclass/vsfscls

int32 VSsetclass(int32 *vdata_id*, char **vdata_class*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_class	IN:	Name of the vdata class

Purpose Sets the class name of a vdata.

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

Description **VSsetclass** sets the class name of the vdata identified by the parameter *vdata_id* to the value of the parameter *vdata_class*.

At creation, the class name of a vdata is `NULL`. The class name may be reset more than once. Class names, like vdata names, can be any character string. They exist solely as meaningful labels to user applications and are not used by the HDF library in any way. Class names will be truncated to `VSNAMELENMAX` (or `64`) characters.

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 by **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 **VSsetexternalfile** writes data in the vdata identified by the parameter *vdata_id* in the file named *filename*, at the byte offset specified by the parameter *offset*.

Only the data will be stored externally. Attributes and all metadata will remain in the primary HDF file.

IMPORTANT: The user 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 **field_name_list*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_name_list	IN:	List of the field names to be accessed

Purpose Specifies the fields to be accessed.

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

Description **VSsetfields** specifies that the fields, whose names are listed in the parameter *field_name_list*, of the vdata identified by the parameter *vdata_id* will be accessed by the next call to **VSread** or **VSwrite**. **VSsetfields** must be called before any call to **VSread** or **VSwrite**.

For reading from a vdata, a call to **VSsetfields** sets up the fields that are to be retrieved from the records in the vdata. If the vdata is empty, **VSsetfields** will return `FAIL` (or `-1`).

For writing to a vdata, **VSsetfields** can only be called once, to set up the fields in a vdata. Once the vdata fields are set, they may not be changed. Thus, to update some fields of a record after the first write, the user must read all the fields to a buffer, update the buffer, then write the entire record back to the vdata.

The parameter *field_name_list* is a character string that contains a comma-separated list of fieldnames (i.e., "PX,PY,PZ" in C and 'PX,PY,PZ' in Fortran). The combined width of the fields in a vdata must be less than `MAX_FIELD_SIZE` (or `65535`) bytes. If an attempt to create a larger record is made, **VSsetfields** will return `FAIL` (or `-1`).

If the vdata is attached with an "`r`" access mode, the parameter *field_name_list* must contain only the fields that already exist in the vdata. If the vdata is attached with a "`w`" access mode, *field_name_list* can contain the names of any fields that have been defined by **VSfdefine** or any predefined fields.

FORTRAN

```
integer function vsfsfld(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```

VSsetinterlace/vsfsint

intn VSsetinterlace(int32 *vdata_id*, int32 *interlace_mode*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
interlace_mode	IN:	Interlace mode of the data to be stored in the vdata

Purpose Sets the interlace mode of a vdata.

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

Description **VSsetinterlace** sets the interlace mode of the vdata, *vdata_id*, to that specified by the parameter *interlace_mode*. This routine can only be used when creating new vdatas with write access.

The value of *interlace_mode* may be either `FULL_INTERLACE` (or `0`) or `NO_INTERLACE` (or `1`). If this routine is not called, the default interlace mode of the vdata is `FULL_INTERLACE`. The `FULL_INTERLACE` option is more efficient than `NO_INTERLACE` although both require the same amount of disk space.

Specifying `FULL_INTERLACE` accesses the vdata by record; in other words, all values of all fields in a record are accessed before moving to the next record. Specifying `NO_INTERLACE` accesses the vdata by field; in other words, all field values are accessed before moving to the next field. Thus, for writing data, all record data must be available before the write operation is invoked.

Note that the interlace mode of the data to be written is specified by a parameter of the **VSwrite** routine.

FORTRAN

```
integer function vsfsint(vdata_id, interlace_mode)

integer vdata_id, interlace_mode
```

VSsetname/vsfsnam

int32 VSsetname(int32 *vdata_id*, char **vdata_name*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
vdata_name	IN:	Name of the vdata

Purpose Assigns a name to a vdata.

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

Description **VSsetname** sets the name of the vdata identified by the parameter *vdata_id* to the value of the parameter *vdata_name*.

At creation, the name of the vdata is `NULL`. The name may be reset more than once. Vdata names, like class names, can be any character string. They exist solely as a meaningful label for user applications and are not used by the HDF library in any way. Vdata names will be truncated to `VSNAMELENMAX` (or `64`) characters.

FORTRAN

```
integer function vsfsnam(vdata_id, vdata_name)

integer vdata_id

character*(*) vdata_name
```

VSsetnumblocks/vsfsetnmbl

intn VSsetnumblocks(int32 *vdata_id*, int32 *num_blocks*)

vdata_id	IN:	Vdata identifier
num_blocks	IN:	Number of blocks to be used for the linked-block element

Purpose Sets the number of blocks for a linked-block Vdata element.

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

Description **VSsetnumblocks** sets the number of blocks in linked-block elements that will be used to store Vdatas.

The default number of blocks is `HDF_APPENDABLE_BLOCK_NUM`, which is set to `16` in the library as it is distributed. **VSsetnumblocks** modifies that default value and must be called before the first write to the Vdata. Once the linked-block element is created, the number of blocks cannot be changed.

FORTRAN

```
integer function vsfsetnmbl(vdata_id, num_blocks)

integer vdata_id, num_blocks
```

VSsizeof/vsfsiz

int32 VSsizeof(int32 *vdata_id*, char **field_name_list*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
field_name_list	IN:	Name(s) of the fields to check

Purpose Computes the size, in bytes, of the given field(s) for the local machine.

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

Description **VSsizeof** computes the size, in bytes, of the fields specified in the parameter *field_name_list* in the vdata identified by the parameter *vdata_id*.

The parameter *field_name_list* specifies a single field or several comma-separated fields. The field or fields should already exist in the vdata. If more than one field is specified, **VSsizeof** will return the total sizes of all of the fields.

FORTRAN

```
integer function vsfsiz(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```

VSwrite/vsfwrt/vsfwrtc/vsfwrit

int32 VSwrite(int32 *vdata_id*, uint8 **databuf*, int32 *n_records*, int32 *interlace_mode*)

vdata_id	IN:	Vdata identifier returned by **VSattach**
databuf	IN:	Buffer of records to be written to the vdata
n_records	IN:	Number of records to be written
interlace_mode	IN:	Interlace mode of the buffer in memory

Purpose Writes data to a vdata.

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

Description **VSwrite** writes the data stored in the buffer *databuf* into the vdata identified by the parameter *vdata_id*. The parameter *n_records* specifies the number of records to be written. The parameter *interlace_mode* defines the interlace mode of the vdata fields stored in the buffer *databuf*.

Valid values for *interlace_mode* are `FULL_INTERLACE` (or `0`) and `NO_INTERLACE` (or `1`). Selecting `FULL_INTERLACE` fills *databuf* by record and is recommended for speed and efficiency. Specifying `NO_INTERLACE` causes *databuf* to be filled by field, i.e., all values of a field in all records must be written before moving to the next field. Thus, all data must be available before writing. If the data is to be written to the vdata with an interlace mode different from that of the buffer, **VSsetinterlace** must be called prior to **VSwrite**. Note that the default interlace mode of a vdata is `FULL_INTERLACE`.

It is assumed that the data in *databuf* is organized as specified by the parameter *interlace_mode*. The number and order of the fields organized in the buffer must correspond with the number and order of the fields specified in the call to **VSsetfields**, which finalizes the vdata fields definition. Since **VSwrite** writes the data in *databuf* contiguously to the vdata, **VSfpack** must be used to remove any "padding", or non-data spaces, used for vdata field alignment. This process is called packing. Refer to the discussion of **VSfpack** in the HDF User's Guide for more information.

Before writing data to a newly-created vdata, **VSdefine** and **VSsetfields** must be called to define the fields to be written.

Note that there are three FORTRAN-77 versions of this routine: **vsfwrt** is for buffered numeric data, **vsfwrtc** is for buffered character data and **vsfwrit** is for generic packed data.

FORTRAN On Windows systems, this function is available only for an integer data buffer.

```
integer function vsfwrt(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

<valid numeric data type> databuf(*)
```

```
integer function vsfwrtc(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

character*(*) databuf


integer function vsfwrit(vdata_id, databuf, n_records,
                  interlace_mode)

integer vdata_id, n_records, interlace_mode

character*(*) databuf
```

DF24addimage/d2aimg

intn DF24addimage(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 (char **filename*, int32 **width*, int32 **height*, intn **interlace_mode*)

filename	IN:	Name of the file
width	OUT:	Width of the image
height	OUT:	Height of the image
interlace_mode	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, interlace_mode)

character*(*) filename

integer width, height, interlace_mode
```

DF24getimage/d2gimg

intn DF24getimage(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(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(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(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
```

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.

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
```

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 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 from an RIS8.

Return value Returns a non-zero reference number if successful and `FAIL` (or `0`) 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 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
```

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.

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
```

DFPaddpal/dpapal

intn DFPaddpal(char **filename*, 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(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(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 (char **filename*, VOIDP *palette*, intn *overwrite*, 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 control of 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(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(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
```

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.

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 transformation 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
```

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.

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 annotation 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.

Valid 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.

Valid 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
```

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.

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.

As an 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. **DFSDrestart** can be used to override this convention.

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 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.

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 accommodate 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.

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.

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 with earlier versions of the library.

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* parameter

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_erruncalibrated offset error
data_typedata 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.

As an 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.

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.

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.

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.

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

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.

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 Specifies the reference number, *ref*, of the dataset to be overwritten next by **DFSDputdata** or **DFSDadddata**.

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

Description **DFSDwriteref** verifies the refence number's existence before returning. 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* parameter, 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
```

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.

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 for flush with zeros, 1 for flush with ones and -1 for 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.

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.

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.

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.

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/reference 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 in 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 access 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( )

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 all errors are processed or when desired.

HEpush

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

error_code	IN:	HDF error code corresponding to the error
func_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 will be ignored. **HEpush** assumes that the character strings *func_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**.

HEvalue

int16 HEvalue(int32 *level*)

level	IN:	Level of the error stack to be returned

Purpose Returns an error code from the specified level of the error stack.

Return value The error code if successful or `DFE_NONE` otherwise.

Description **HEvalue** returns the error code at the top of the stack, when *level* is 1. Refer to Table 1B of Section 1 in this reference manual for a complete list of HDF4 error codes.

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 1A

*_INTERLACE - Interlace Mode Codes

Definition Name	Definition Value
FULL_INTERLACE	0
NO_INTERLACE	1

TABLE 1B

*_WILDCARD - Wildcard Code

Definition Name	Definition Value
DFREF_WILDCARD	0

TABLE 1C

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 1D

COMP_* - Raster Image Compression Codes

Definition Name	Definition Value
COMP_NONE	0
COMP_RLE	11
COMP_IMCOMP	12
COMP_JPEG	2

TABLE 1E

COMP_CODE_* - General 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
COMP_CODE_JPEG	6

TABLE 1F

DF_* - Maximum Length Codes

Definition Name	Definition Value
DF_MAXFNLEN	256

TABLE 1G

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

TABLE 1H

DFE_* - Error Codes

Definition Name	Definition Value
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_NOENCODER	88
DFE_NOSZLIB	89
DFE_COMPVERSION	90
DFE_READCOMP	91
DFE_NODIM	92
DFE_BADRIG	93
DFE_RINOTFOUND	94
DFE_BADATTR	95
DFE_LUTNOTFOUND	96
DFE_BADTABLE	97
DFE_BADSDG	98
DFE_BADNDG	99
DFE_VGSIZE	100
DFE_VTAB	101
DFE_CANTADDELEM	102
DFE_BADVGNAME	103
DFE_BADVGCLASS	104
DFE_BADFIELDS	105
DFE_NOVS	106
DFE_SYMSIZE	107
DFE_BADATTACH	108
DFE_BADVSNAME	109
DFE_BADVSCLASS	110
DFE_VSWRITE	111
DFE_VSREAD	112
DFE_BADVH	113

DFE_FIELDSSET	114
DFE_VSCANTCREATE	115
DFE_VGCANTCREATE	116
DFE_CANTATTACH	117
DFE_CANTDETACH	118
DFE_BITREAD	119
DFE_BITWRITE	120
DFE_BITSEEK	121
DFE_TBBTINS	122
DFE_BVNEW	123
DFE_BVSET	124
DFE_BVGET	125
DFE_BVFIND	126

TABLE 1I

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 1J

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 1K

DFTAG_* - Object Tags

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 1L

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

TABLE 1M

MFGR_* - Interlace Mode Codes

Definition Name	Definition Value
MFGR_INTERLACE_PIXEL	0
MFGR_INTERLACE_LINE	1
MFGR_INTERLACE_COMPONENT	2

TABLE 1N

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 1O

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 1P

SUCCEED/FAIL - Routine Return Status Codes

Definition Name	Definition Value
SUCCEED	0
FAIL	-1