Chapter 3

Scientific Data Sets (SD API)

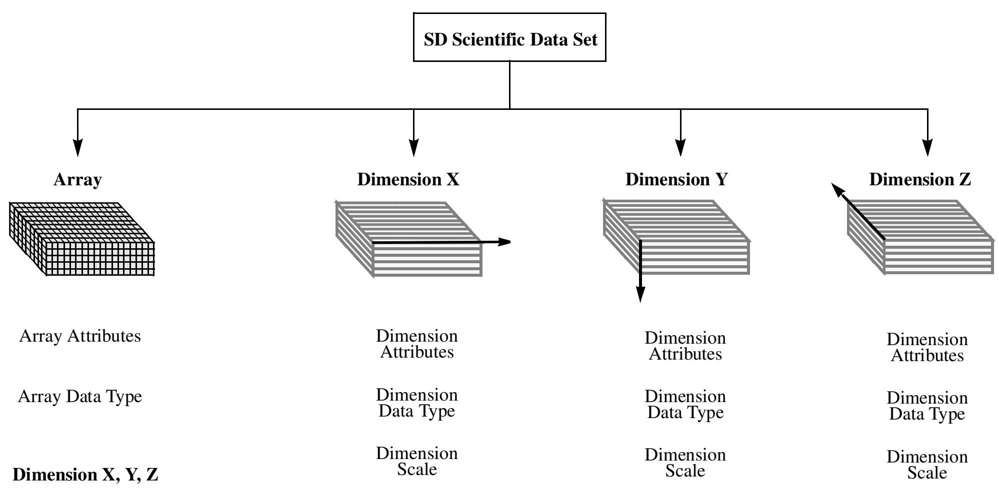
3.1 Chapter Overview

This chapter describes the routines available for storing and retrieving multidimensional arrays containing scientific data.

3.2 The SD Scientific Data Set Data Model

In HDF, any multi-dimensional array qualifies as a ***scientific data set*** or ***SDS*** if it's associated with a dimension record and a data type. In addition to providing a framework for storing arrays of arbitrary dimensions and data type, the SDS data model supports ***dimension scales***, ***user-defined attributes*** and ***predefined attributes***. (See Figure 3a.)

FIGURE 3a **The Contents of a Three-Dimensional SD Scientific Data Set**



Scientific data sets consist of required and optional objects. The required objects will be covered first. Please note that in this chapter the terms "SDS" and "data set" are used interchangeably.

3.2.1 Required SD SDS Objects

Every SD scientific data set must contain three objects. These objects include the ***SDS array***, ***data type*** and ***dimension records***. Required objects are automatically created from the information provided at the time the SDS is defined.

3.2.1.1 SDS Array

An ***SDS array*** is an n-dimensional data structure that serves as the basic building block of an SDS. When an SDS array is created, the number and size of the dimensions that define its shape are specified, as is its data type. SDS arrays are conceptually equivalent to ***variables*** in the netCDF data model.

3.2.1.2 SDS Array Name

An SDS array has an ***SDS name*** consisting of a sequence of case-sensitive alphanumeric characters. A name can be assigned to an SDS by a calling program, but if a name is not provided by the calling program one will be assigned by the HDF library. Names are assigned when the data set is created and cannot be changed. SDS names do not have to be unique within a file, but if they are not it can be difficult to distinguish between the scientific data sets in the file.

3.2.1.3 Data Type

The standard data types supported by the SD API are 32- and 64-bit floating-point numbers, 8-, 16- and 32-bit signed integers and 8-, 16- and 32-bit unsigned integers. The SD interface also includes a routine that allows SD data sets with variable bit lengths (1 to 32 bits) to be created.

Before writing an SDS to a file, HDF normally converts its elements from the native format of the host machine to a standard HDF format. The standard representations used by HDF for floating-point numbers are the IEEE 32- and 64-bit floating-point formats. For integers, HDF uses big-endian byte ordering. For signed integers HDF uses twos-complement representation. Converting to and from the standard formats can result in low-order inaccuracies in the data. For example, data converted from 64-bit to 32-bit floating-point representation is accurate to about 10^{-7}.

Sometimes users prefer not to have their data automatically converted, either because the conversion slows down processing or because it introduces intolerable inaccuracies. For those instances, HDF provides a “native format” option, whereby numbers are stored "as is" in the file and are tagged accordingly. HDF also provides a “little-endian” option to suppress any rearranging of byte ordering from little- to big-endian. This is primarily for users of Intel-based machines who do not want to incur the cost of reordering data when writing to an HDF file. Because HDF does not support direct conversion between many machine architectures, using a native format can diminish the portability of HDF files. However, note that direct conversions are supported between little-endian and all other byte-order formats supported by HDF.

3.2.1.4 Dimensions

SDS ***dimensions*** specify the shape and size of an SDS array. The number of dimensions of an array is known as the ***rank*** of the array. Dimension names are not treated in the same way as array names. For example, if a name assigned to a dimension was previously assigned to another dimension the SD interface treats both dimensions as the same data object and any changes made to one will be reflected in the other. The size of a dimension is a positive integer.

Also, one dimension of an SDS array can be assigned the predefined size `SD_UNLIMITED`. This dimension is referred to as an ***unlimited dimension*** - which, as the name suggests, can grow to any length.

3.2.2 Optional SD SDS Objects

There are two types of optional objects available for inclusion in an SDS: ***dimension scales*** and ***attributes***. Attributes are either predefined or defined by the user. Optional objects are only created when specifically requested by the calling program.

Dimension Scales

A dimension scale is a sequence of numbers placed along a dimension to demarcate intervals along it. Dimension scales are described in Section 3.9 on page 71.

User-Defined Attributes

Attributes are alphanumeric strings describing the nature and/or intended usage of the file, SDS or dimension they're attached to. ***User-defined attributes*** are attributes defined by the calling program containing auxiliary information about a file, SDS array or dimension. They are more fully described in Section 3.10 on page 80.

Predefined Attributes

Predefined attributes are attributes that have reserved labels and in some cases predefined data types. Predefined attributes are useful because they establish conventions that applications can depend on. They are further described in Section 3.11 on page 87.

3.2.3 Annotations and the SD Data Model

With the expansion of the SD interface to include user-defined attributes, annotations should no longer be used in conjunction with scientific data sets. In fact, metadata once stored as an annotation is now more conveniently stored as an attribute. However, to insure backward compatibility with scientific data sets and applications relying on annotations, the DFAN annotation API, described in Chapter 10, titled *Annotations (DFAN API)*, can be used to annotate SDSs. There is no cross-compatibility between attributes and annotations; creating one does not automatically create the other.

3.3 The SD Scientific Data Set API

Unlike the DFSD SDS interface, the SD interface supports simultaneous access to more than one SDS in more than one HDF file. Although it is fully compatible with the DFSD data model, the SD interface also supports a more powerful and more general scientific data model; one that is very similar to the netCDF data model developed by the Unidata Program Center.

All valid operations involving an SDS can be done by calling routines in the SD interface. Except in rare instances, the physical file format used to store HDF data is transparent to HDF users.

3.3.1 SD Library Routines

All C routines in the SD scientific data library begin with the prefix "SD". The equivalent Fortran-77 functions use the prefix "sf". These routines are categorized as follows:

- ***Access routines*** initialize and terminate access to HDF files and data sets.
- ***Read and write routines*** read and write data sets by manipulating their dimensions, rank and data type.
- ***General inquiry routines*** return information about the location, contents and description of the scientific data sets in an HDF file.
- ***Dimension scale routines*** define and access dimension scales within a data set.
- ***User-defined attribute routines*** describe and access characteristics of an HDF file, data set or dimension defined by the HDF user.
- ***Predefined attribute routines*** access previously-defined characteristics of an HDF file, data set, or dimension.

- ***Compression routines*** determine the compression method for SDS data.
- ***Chunking/tiling routines*** determine the chunking configuration for SDS data.
- ***N-bit data set routines*** determine the non-standard data bit length configuration for SDS data.

The SD routines are described in the following table and in the *HDF Reference Guide*.

TABLE 3A

SD Library Routines

Category	Routine Name		Description
	C	Fortran-77	
Access	SDend	sfend	Closes the file and clean up memory.
	SDendaccess	sfendacc	Disposes of a data set identifier, flush out metadata and order information.
	SDselect	sfselect	Returns the identifier of the specified data set.
	SDstart	sfstart	Initializes the SD interface.
Read and Write	SDcreate	sfcreate	Creates a new data set.
	SDreaddata	sfrdata/ sfrcdata	Reads data from a chunked or non-chunked data set.
	SDsetexternalfile	sfsextf	Defines the data type to be stored in an external file.
	SDwritedata	sfwdata/ sfwcdata	Writes data to a chunked or non-chunked data set.
General Inquiry	SDfileinfo	sffinfo	Returns information about the contents of a file.
	SDgetinfo	sfginfo	Returns information about a data set.
	SDidtoref	sfid2ref	Returns a reference number for a named data set.
	SDiscoordvar	sfiscvar	Distinguishes data sets from dimension scales.
	SDnametoindex	sfn2index	Returns an index of a specified data set.
	SDreftoindex	sfref2index	Returns the index of a data set corresponding to a given reference number.
Dimension Scales	SDdiminfo	sfgdinfo	Gets information about a dimension.
	SDgetdimid	sfdimid	Retrieves the identifier of a dimension.
	SDsetdimname	sfsdimname	Associates a name with a dimension.
	SDgetdimscale	sfgdscale	Returns scale values for a dimension.
	SDsetdimscale	sfsdscale	Defines the values of this dimension.
User-defined Attributes	SDattrinfo	sfgainfo	Gets information about an attribute.
	SDfindattr	sffattr	Returns the index of the specified attribute.
	SDreadattr	sfrnatt/ sfrcatt	Reads the values of the specified attribute.
	SDsetattr	sfsnatt/ sfscatt	Creates and defines a new attribute.

Predefined Attributes	`SDgetcal`	`sfgcal`	Returns calibration information.
	`SDgetdatastrs`	`sfgdtstr`	Returns the label, limit, format and coordinate system of a data set.
	`SDgetdimstrs`	`sfgdmstr`	Returns the attribute strings for a dimension.
	`SDgetfillvalue`	`sfgfill/ sfgcfill`	Reads the fill value if it exists.
	`SDgetrange`	`sfgrange`	Returns the range of values of the specified data set.
	`SDsetcal`	`sfscal`	Defines the calibration information.
	`SDsetdatastrs`	`sfsdtstr`	Defines the attribute strings of the specified data set.
	`SDsetdimstrs`	`sfsdmstr`	Defines the attribute strings of the specified dimension.
	`SDsetfillvalue`	`sfsfill/ sfscfill`	Defines the fill value of the current data set.
	`SDsetfillmode`	`sfsflmd`	Defines the file mode to be applied to all SDSs in the specified file.
	`SDsetrange`	`sfsrange`	Defines the maximum and minimum values of the valid range.
Compression	`SDsetcompress`	`None`	Defines the compression method to be applied to data set data.
Chunking/ Tiling	`SDgetchunkinfo`	`None`	Obtains information about a chunked SDS.
	`SDsetchunkcache`	`None`	Sets the size of the chunk cache.
	`SDsetchunk`	`None`	Makes a non-chunked SDS a chunked SDS.
	`SDwritechunk`	`None`	Writes data to a chunked SDS.
	`SDreadchunk`	`None`	Reads data from a chunked SDS.
N-bit Data Length	`SDsetnbitdataset`	`sfsnbit`	Defines the non-standard bit length of the data set data.

3.3.2 SDS Identifiers in the SD Interface

In the SD interface, scientific data sets are identified in several ways. Before a data set is accessible, it is identified by ***index***, ***name*** and ***reference number***. After it is opened, it is identified by a ***data set identifier*** or ***SDS id***.

The index describes the relative position of the data set in the file. The name is a unique character string and the reference number is a unique integer which are both assigned to the data set when it is created. The index, name, and reference number are needed to obtain a data set id.

The SDS id is the data set equivalent of a file identifier and uniquely identifies an SDS data set within a file. It is created when an existing SDS is selected for use or when a new SDS is created and is thereafter used to access the SDS.

3.3.3 Tags in the SD Interface

A complete list of SDS tags and their descriptions is in Table D in the User's Guide Appendix.

3.4 Programming Model for the SD Interface

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

1. Open a file and initialize the SD interface.
2. Open an existing SDS by obtaining an SDS id from an SDS index **OR** create a new SDS by obtaining an SDS id from the SDS name, rank and dimensions.
3. Perform desired operations on the SDS.
4. Terminate access to the data set.
5. Terminate access to the SD interface and close the file.

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

C:
```
sd_id = SDstart(filename, access_mode);
sds_id = SDselect(sd_id, sds_index);
<Optional operations>
status = SDendaccess(sds_id);
status = SDend(sd_id);
```

FORTRAN:
```
sd_id = sfstart(filename_1, access_mode)
sds_id = sfselect(sd_id, sds_index)
<Optional operations>
status = sfendaccess(sds_id)
status = sfend(sd_id)
```

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

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

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

As with file identifiers, SD ids can be obtained and discarded in any order and all SD ids must be individually discarded before termination of the calling program.

3.4.1 Establishing Access to Files and Data Sets: SDstart and SDselect

In the SD interface, **SDstart** is used to open files rather than **Hopen**. **SDstart** takes two arguments; `filename` and `access_mode`, and returns the file id `sd_id`. The argument `filename` is the name of an HDF or netCDF file as it is stored on disk. All other functions in the SD interface accept only `sd_id` for file operations. The argument `access_mode` specifies the type of access required for operations on the file. The access mode tags passed in the `access_mode` parameter have names prefaced by "DFACC".

Although it is possible to open a file more than once, it is better to select the most appropriate access mode and call **SDstart** only once. Repeatedly calling **SDstart** may cause unexpected results and is not recommended.

If `access_mode` is set to `DFACC_RDONLY`, the specified file will not be created if it doesn’t exist.

SDselect takes two arguments: `sd_id` and `sds_index` and returns the data set id `sds_id`. The argument `sd_id` is the file identifier returned by **SDstart**, and `sds_index` is the position of the data set relative to the beginning of the file. The argument `sds_index` is zero-based, meaning that the index of first SDS in the file is 0.

The parameters of **SDstart** and **SDselect** are further defined below. (See Table 3B.)

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

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

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

The parameters of **SDendaccess** and **SDend** are further defined in the following table.

TABLE 3B **SDstart, SDselect, SDend and SDendaccess Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDstart (sfstart)	filename	char *	character* (*)	Name of the HDF or netCDF file.
	access_mode	int32	integer	Type of access.
SDselect (sfselect)	file_id	int32	integer	HDF file identifier.
	sds_index	int32	integer	Position of the data set within the file.
SDend (sfend)	file_id	int32	integer	HDF file identifier.
SDendaccess (sfendacc)	sds_id	int32	integer	Data set identifier.

EXAMPLE 1. **Accessing and Closing an SDS**

The following examples are a code template on how to access the first data set in an HDF file, then detach from the data set and the file. The file name "Dummy_HDF_File.hdf" represents a preexisting HDF file - replace it with the name of your target file.

This example assumes that the "Dummy_HDF_File.hdf" file contains one SDS.

C:

```
#include "mfhdf.h"

main( )
{
int32 sd_id, sds_id, sds_index, status;
int32 rank, num_type, attributes;

/* Open the HDF file. DFACC_CREATE is defined in hdf.h. */
sd_id = SDstart("Dummy_HDF_File.hdf", DFACC_CREATE);

/* Get the identifier of the first data set. */
sds_index = 0;
sds_id = SDselect(sd_id, sds_index);
```

```
/* Dispose of the data set identifier to terminate access. */
status = SDendaccess(sds_id);

/* Dispose of the file identifier to close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM SDSDATA ACCESS

       integer*4 sds_id, sd_id, sds_index, status
       integer sfstart, sfselect, sfendacc, sfend

C      DFACC_CREATE is defined in hdf.h.
       parameter (DFACC_CREATE = 4)

C      Open the HDF file.
       sd_id = sfstart('Dummy_HDF_File.hdf', DFACC_CREATE)

C      Get the identifier of the first data set.
       sds_index = 0
       sds_id = sfselect(sd_id, sds_index)

C      Dispose of the data set identifier to terminate access.
       status = sfendacc(sds_id)

C      Dispose of the file identifier to close the file.
       status = sfend(sd_id)

       end
```

3.5 Creating and Writing to Simple Scientific Data Sets

This section describes the routines needed to create and write simple SDSs. A “simple” SDS is defined here as one with no attributes or user-defined dimension scales.

In the SD interface, creating and writing data to an SDS are separate operations - each performed by one SD routine. The SD interface retains no definitions about the size, contents or rank of an SDS from one SDS to the next or from one file to the next.

3.5.1 Creating Scientific Data Sets: SDcreate

Creating a simple SDS or an SDS without attributes or scales involves the following steps:

1. Open a file and initialize the SD interface.
2. Define the characteristics of the SDS.
3. Perform optional operations on the SDS.
4. Terminate access to the data set.
5. Terminate access to the SD interface and close the file.

To create an SDS, the calling program must contain the following sequence of routine calls:

C:

```
sd_id = SDstart(filename, access_mode);
sds_id = SDcreate(sd_id, name, number_type, rank, dim_sizes);
```

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

FORTRAN:
```
sd_id = sfstart(filename, access_mode)
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
<Optional operations>
status = sfendacc(sds_id)
status = sfend(sd_id)
```

SDcreate defines a new SDS using the arguments `name`, `number_type`, `rank` and `dim_sizes`. Once a data set is created, you cannot change its name, data type or rank. **SDcreate** does not actually perform the write; it occurs only when **SDend** is called.

The SD interface will assign a name if one is not provided. In this situation, the `filename` parameter must be a null character string. The maximum length of an SDS name is defined by `MAX_NC_NAME` and the maximum rank of an SDS array is defined by `MAX_VAR_DIMS`. Both are defined in the "netcdf.h" header file. Tag names passed in the `number_type` parameter are prefaced by "DFNT".

When creating an SDS, it is necessary to specify the data type of the array data contained in the SDS. The "hntdefs.h" header file contains definitions of all valid data types, which are described in Chapter 2, titled *HDF Fundamentals*.

TABLE 3C

SDcreate Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDcreate (sfcreate)	file_id	int32	integer	File identifier.
	name	char *	character* (*)	ASCII string containing the name of the data set.
	data_type	int32	integer	Data type of the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 []	integer (*)	Array defining the size of each dimension.

EXAMPLE 2.

Creating an Empty SDS

If **SDcreate** is called but not written, an empty array is created. An "empty array" is an array that has been defined but not yet initialized with data. All array information passed into **SDcreate** through its parameters are stored in the file.

C:

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

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id, status;
int32 dimsizes[2], rank;

/* Create and open the file and initiate the SD interface. */
```

```
sd_id = SDstart("Example2.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 2;
dimsizes[0] = Y_LENGTH;
dimsizes[1] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_1", DFNT_INT16, rank, dimsizes);

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

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

}
```

FORTRAN:

```
      PROGRAM EMPTY ARRAY

       integer*4 sd_id, sds_id, dimsizes(2), rank
       integer sfstart, sfcreate, sfendacc, sfend
       integer*4 X_LENGTH, Y_LENGTH
       parameter (X_LENGTH = 16, Y_LENGTH = 5)

C      DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
       integer*4 DFACC_CREATE, DFNT_INT16
       parameter (DFACC_CREATE = 4, DFNT_INT16 = 22)

C      Create and open the file and initiate the SD interface.
       sd_id = sfstart('Example2.hdf', DFACC_CREATE)

C      Define the rank and dimensions of the array to be created.
       rank = 2
       dimsizes(1) = Y_LENGTH
       dimsizes(2) = X_LENGTH

C      Create the array.
       sds_id = sfcreate(sd_id, 'Ex_array_1', DFNT_INT16, rank,
      +                  dimsizes)

C      Terminate access to the array.
       status = sfendacc(sds_id)

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

       end
```

3.5.2 Writing Data to an SDS Array: SDwritedata

SDwritedata is the only routine that is needed to write data to an SDS array. It can completely or partially fill an SDS n-dimensional array or append data along one dimension defined to be of unlimited length. It can also skip a specified number of SDS array elements between write operations along each dimension.

Creating an SDS and writing one or more slabs to it involves the following steps:

1. Create an SDS.

2. Write a slab or series of slabs.

To do this, the calling program must contain the following sequence of routine calls:

C:
```
sds_id = SDcreate(sd_id, name, number_type, rank, dim_sizes);
status = SDwritedata(sds_id, start, stride, edge, data);
```

FORTRAN:
```
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
status = sfwdata(sds_id, start, stride, edge, data)
```

A ***slab*** is an n-dimensional array whose dimensions are smaller than or equal to those of the SDS array into which it will be written. A slab is defined by the n-dimensional coordinate of its initial vertex and the lengths of each dimension.

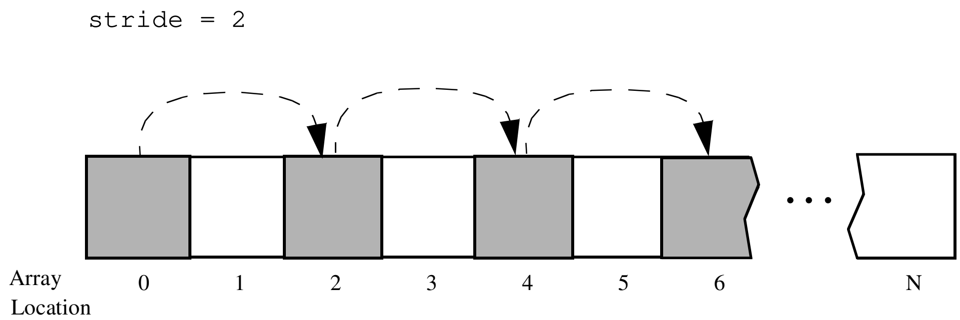
SDwritedata takes five arguments: `sds_id`, `start`, `stride`, `edge`, and `data`. The `sds_id` argument is the SDS identifier returned by **SDcreate** or **SDselect**. The arguments `start`, `stride`, and `edge` respectively describe the n-dimensional coordinates the SD interface will begin the write operation in the data set, the number of locations the current SDS location will be moved forward after each write, and the length of each dimension of the n-dimensional slab to be written. If the SDS array is smaller than the `data` argument array, the amount of data written will be limited to the maximum size of the SDS array.

The argument `start` is an array specifying the location in the data set array the write operation will begin. The indices are relative to 0 so the first data value of an array would have index `{0,0, ... 0}`. The size of `start` must be the same as the number of dimensions in the SDS array. In addition, each value in `start` must be smaller than its corresponding SDS array dimension unless the dimension is unlimited. Violating any of these conditions causes a termination of the write operation and causes an error condition to be generated.

The argument `stride` is an array specifying, for each dimension, the interval between values to be written. For example, setting `stride[0]=2` writes data to every other location along the first dimension.(See Figure 3b on page 30.) Setting `stride[0]=1` writes data to every location along the first dimension. If `stride` is defined as `NULL` in C or `0` in Fortran-77, it is assumed to contain all ones. For better performance, is it advised that you define the value of `stride` as `NULL` rather than setting it equal to 1. The length of the `stride` array must be the same as the number of dimensions in the SDS array. Also, each value in `stride` must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. A violation of any of these conditions terminates the write operation and causes an error condition to be generated.

FIGURE 3b

Description of "Strides"



The argument `edge` is an array specifying the length of each dimension of the slab to be written. If the rank of the slab is less than that of the SDS data set, all additional dimensions must be specified as 1. Each value in the `edge` array must not be larger than the length of the corresponding dimension in the SDS data set. Attempting to read or write slabs larger than the size of the SDS data set will result in an error condition. The size of `edge` must be equal to the number of dimensions in the SDS array. In addition, the sum of each value in the `edge` array and the corresponding value in the `start` array must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. A violation of any of these conditions terminates the write operation and results in an error condition.

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

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

The parameters of **SDwritedata** are described in the following table. Note that, because there are two Fortran-77 versions of **SDwritedata**, there are correspondingly two entries in the “Data Type” field of the *data* parameter.

TABLE 3D

SDwritedata Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDwritedata (sfwdata/ sfwcdata)	sds_id	int32	integer	Data set identifier.
	start	int32 []	integer (*)	Array containing the position the write will start for each dimension.
	stride	int32 []	integer (*)	Array containing the number of data locations the current location is to be moved forward before the next write.
	edge	int32 []	integer (*)	Array containing the number of data elements that will be written along each dimension.
	data	VOIDP	<valid numeric data type>	Buffer for the data to be written.

3.5.2.1 Filling an Entire Array

Filling an array is a simple slab operation where the slab begins at the origin of the SDS array and fills every location in the array. **SDwritedata** will fill an SDS array with data when given the origin (`start={0,0, ... 0}`) as its starting coordinates, a stride value of `NULL`, and edge dimensions equal to the size of the SDS array (`edge={dim_sizes[0], dim_sizes[1], ... dim_size[rank-1]}`).

EXAMPLE 3. **Creating and Writing to an SDS**

These examples use **SDcreate** under the C interface and **sfcreate** under the Fortran-77 interface. The only difference between writing array data to an existing SDS and writing it to a newly created array is the use of **SDselect** instead of **SDcreate**.

C:

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

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{
int32 sd_id, sds_id, status;
int32 dims[2], start[2], edges[2], rank;
int16 array_data[Y_LENGTH][X_LENGTH];
intn i, j;

/* Create and open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the data set to be created. */
rank = 2;
dims[0] = Y_LENGTH;
dims[1] = X_LENGTH;

/* Create the array data set. */
sds_id = SDcreate(sd_id, "Ex_array_3", DFNT_INT16, rank, dims);

/* Fill the stored-data array with values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        array_data[j][i] = (i + j) + 1;
}

/* Define the location, pattern, and size of the data set */
for (i = 0; i < rank; i++) {
    start[i] = 0;
    edges[i] = dims[i];
}

/* Write the stored data to the "Ex_Array_3" data set. The fifth
* argument must be explicitly cast to a generic pointer to conform
* to the HDF API definition for SDwritedata. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)array_data);

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

/* Terminate access to the SD interface and close the file. */
```

```
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM FILLED ARRAY

      integer*4 sd_id, sds_id, rank
      integer dims(2), start(2), edges(2), stride(2), status
      integer*2 i, j
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22, X_LENGTH = 5,
     +           Y_LENGTH = 16)
      integer*2 array_data(X_LENGTH, Y_LENGTH)

C     Create and open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the data set to be created.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the data set.
      sds_id = sfcreate(sd_id, 'Ex_array_3', DFNT_INT16, rank, dims)

C     Fill the stored-data array with values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j - 1
10         continue
20    continue

C     Define the location, pattern, and size of the data set
C     that will be written to.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the stored data to the "Ex_array_3" data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

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

      end
```

3.5.2.2 Writing Slabs to an SDS Array

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

EXAMPLE 4.

Writing a Slab of Data to an SDS

These examples show how to write a series of slabs to a data set. The programs create a three-dimensional array with the size of the x-dimension being four elements, the y-dimension five elements and the z-dimension six elements. As the data set elements are written, the slabs are "sliced" along the y-dimension.

Note that the *start*, *edge* and *stride* arguments in **SDwritedata** refer to the coordinate locations in the file representation of the data set, not the location within the `write_data` array. Therefore, the actual slab to be written is first buffered in the `zx_data` array.

In most real-world HDF-based applications that perform slab writes, the `write_data` array would be a buffer area for data previously read in by **SDreaddata**. In this example, this data read step has been omitted so that this example can focus on the slab write procedure.

C:

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

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{
int32 sd_id, sds_id, rank, status;
int32 dims[3], start[3], edges[3];
int16 write_data[Z_LENGTH][Y_LENGTH][X_LENGTH];
int16 zx_data[Z_LENGTH][X_LENGTH];
intn i, j, k;

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 3;
dims[0] = Z_LENGTH;
dims[1] = Y_LENGTH;
dims[2] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_4", DFNT_INT16, rank, dims);

/* Compute and store the values that will be written to the data
   set. Fill the write_data array along the x-axis first. */
for (k = 0; k < Z_LENGTH; k++)
    for (j = 0; j < Y_LENGTH; j++)
        for (i = 0; i < X_LENGTH; i++)
            write_data[k][j][i] = (i + 1) + (j + 1) + (k + 1);

/* Within each for loop, set the start and edge parameters to write \
a 4-by-6 element slab of stored data from the write_data array to \
the data set. */
edges[0] = Z_LENGTH;
edges[1] = 1;
edges[2] = X_LENGTH;
start[0] = start[2] = 0;
```

```
for (j = Y_LENGTH; j > 0; j--) {
    start[1] = j - 1;
    for ( k = 0; k < Z_LENGTH; k++) {
        for ( i = 0; i < X_LENGTH; i++) {
            zx_data[k][i] = write_data[k][j-1][i];
            status = SDwritedata(sds_id, start, NULL, edges,
                         (VOIDP)zx_data);
                }
            }
}

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

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

}
```

FORTRAN:

```
     PROGRAM WRITE SLAB

      integer*4 sd_id, sds_id, rank
      integer dims(3), start(3), edges(3), stride(3)
      integer i, j, k, status
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH, Z_LENGTH
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22, X_LENGTH = 4,
     +           Y_LENGTH = 5, Z_LENGTH = 6)

      integer*2 write_data(X_LENGTH, Y_LENGTH, Z_LENGTH)
      integer*2 zx_data(X_LENGTH, Z_LENGTH)

C     Create the file.
      sd_id = sfstart('Example4.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the array to be created.
      rank = 3
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH
      dims(3) = Z_LENGTH

C     Create the array.
      sds_id = sfcreate(sd_id, 'Ex_array_4', DFNT_INT16, rank, dims)

C     Compute and store the values that will be later written to the
C     selected array data set. Fill the array_data array along the
C     x-axis first.
      do 30 k = 1, Z_LENGTH
         do 20 j = 1, Y_LENGTH
            do 10 i = 1, X_LENGTH
               write_data(i, j, k) = i + j + k
10          continue
20       continue
30    continue

C     Within each do loop, set the start and edge parameters so that
C     a 4-by-6 element slab of stored data will be written from the
```

```
C     array_data array to the data set.
      edges(1) = X_LENGTH
      edges(2) = 1
      edges(3) = Z_LENGTH
      start(1) = 0
      start(3) = 0
      stride(1) = 1
      stride(2) = 1
      stride(3) = 1

      do 60 j = Y_LENGTH, 0, -1
       start(2) = j - 1
       do 50 k = 1, Z_LENGTH
        do 40 i = 1, X_LENGTH
         zx_data(i, k) = write_data(i, j, k)
         status = sfwdata(sds_id, start, stride, edges, zx_data)
40      continue
50     continue
60    continue

C     Terminate access to the data set.
      status = sfendacc(sds_id)

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

      end
```

EXAMPLE 5.

Altering Values Within an Array Data Set

These examples demonstrate the procedure for changing the value of one element, located in the second column and tenth row, of a data set.

This procedure can be used to alter the values of a group of elements within the data set by changing the `start` and `edge` arguments of the second call of the **SDwritedata** function. Notice that a `NULL` value is passed into **SDwritedata** in the C program instead of the `stride` array.

C:

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

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{
int32 sd_id, sds_id, status;
int32 start[2], edges[2];
int16 write_data[Y_LENGTH][X_LENGTH], intval;
intn i, j;

/*  Open the file. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

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

/* Compute and store the values that will be written to the
   selected array data set. */
for (j = 0; j < Y_LENGTH; j++)
```

```
    for (i = 0; i < X_LENGTH; i++)
        write_data[j][i] = (i + 1) + (j + 1) * 10;

/* Set up the start and edge parameters to write the buffered
   data to the entire array data set. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Write the buffered data to the "Ex_array_3" data set. The fifth
   argument must be explicitly cast to a generic pointer to conform
   to the HDF API definition of SDwritedata.*/
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)write_data);

/* Alter a value within this data set. */
intval = 15;
start[0] = 10;
start[1] = 1;
edges[0] = 1;
edges[1] = 1;
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)&intval);

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

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

}
```

FORTRAN:

```
     PROGRAM ALTER DATA

      integer*4 sd_id, sds_id
      integer start(2), edges(2), stride(2)
      integer i, j, status
      integer sfstart, sfselect, sfwdata, sfendacc, sfend

C     DFACC_RDWR is defined in hdf.h.
      integer*4 DFACC_RDWR
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDWR = 3, X_LENGTH = 5, Y_LENGTH = 16)

      integer*2 array_data(X_LENGTH, Y_LENGTH), intval

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

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

C     Compute and store the values that will be later written to
C     the data set.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j * 10
10         continue
20    continue

C     Initialize the start, edge and stride parameters to write the
C     stored data to the data set.
      start(1) = 0
```

```
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the stored data to the data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

C     Alter the value of the data set element in the first column,
C     tenth row to be 15.
      intval = 15
      start(1) = 1
      start(2) = 10
      edges(1) = 1
      edges(2) = 1
      status = sfwdata(sds_id, start, stride, edges, intval)

C     Terminate access to the data set.
      status = sfendacc(sds_id)

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

      end
```

3.5.2.3 Appending Data to an SDS Array Using the Unlimited Dimension

If one dimension of an SDS array must be extendable, it is possible to create an ***appendable SDS array***. An SDS array is appendable if one dimension is specified as an "unlimited" when the array is created.

In C, if the **SDcreate** parameter `dim_sizes[0]` is assigned the value `SD_UNLIMITED` it is considered to be an unlimited dimension. This is the only dimension that may be specified as unlimited in a C program and can also be understood as the *first* dimension or the dimension of the *lowest* rank value, of the SDS. In Fortran-77, only the *last* dimension or the dimension of the *highest* rank value, can be unlimited. In other words, in Fortran-77 `dim_sizes(rank)` must be set to the value `SD_UNLIMITED` to make the array appendable.

To append data to an extensible data set without overwriting data, specify the appropriate `start` coordinates in the **SDwritedata** routine. For example, if the current coordinate boundary of an unlimited dimension is defined as 15, appending data to the array without overwriting existing data requires a `start` coordinate of 16. To append data by overwriting data, specify a `start` coordinate less than the current boundary of the unlimited dimension. In either case, all but the unlimited coordinate in `start` must fall within the boundaries of the original array definition.

Any time a write operation is attempted beyond the current boundary, the HDF library will automatically adjust the dimension record to the new length. If the new data both begins and ends past the boundary of the array, locations between the existing boundary and the beginning of the new data stream are initialized to the assigned fill value if there is one or the default fill value if none is assigned.

EXAMPLE 6. **Appending Data to an SDS Array Using an Unlimited Dimension**

In the C example, the length of the SDS array's y-dimension (or the first dimension) is set to the value `SD_UNLIMITED`, which defines the x-dimension (or the second and last dimension) as appendable. In the Fortran-77 version, the length of the y-dimension (or the second and last dimension) is set to the value SD_UNLIMITED. Again, this is because only the dimension of the highest rank (in this case the x-dimension is of rank 1 and the y-dimension is of rank 2) can be defined as appendable under the Fortran-77 interface.

C:

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

#define X_LENGTH 10
#define Y_LENGTH 10

main( )
{

int32 sd_id, sds_id, sds_idx;
int32 dims[2], rank;
int16 array_data[Y_LENGTH][X_LENGTH], append_data[X_LENGTH];
int32 start[2], edges[2];
intn i = 0, j, status;

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

/* Define the rank and dimensions of the array. Make the first \
   array dimension appendable by defining it's length to be \
   unlimited.*/
rank = 2;
dims[0] = SD_UNLIMITED;
dims[1] = X_LENGTH;

/* Create the array data set. */
sds_id = SDcreate(sd_id, "Ex_File_6", DFNT_INT16, rank, dims);

/* Store the array values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        array_data[j][i] = (i + 1) + (j + 1);
}

/* Write the data to the array. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Perform the initial write to the array data set. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)array_data);

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

/* Reopen the file and initiate the SD interface in preparation
for appending data to the data set. Then select the first
data set.*/
sd_id = SDstart("Example3.hdf", DFACC_RDWR);
sds_idx = SDnametoindex(sd_id, "Ex_File_6");
sds_id = SDselect(sd_id, sds_idx);

/* Store the array values to be appended to the data set. */
for (i = 0; i < X_LENGTH; i++)
    append_data[i] = i + 1;
```

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

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

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

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

}
```

FORTRAN:

```
     PROGRAM APPEND DATA

      integer*4 sd_id, sds_id, sds_idx
      integer dims(2), rank
      integer start(2), edges(2), stride(2)
      integer status
      integer sfstart, sfcreate, sfwdata, sfn2index, sfselect
      integer sfendacc, sfend

C     DFACC_RDWR and DFNT_INT16 are defined in hdf.h, SD_UNLIMITED
C     is defined in hdf.h.
      integer*4 DFACC_RDWR, DFNT_INT16, SD_UNLIMITED
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDWR = 3, DFNT_INT16 = 22, SD_UNLIMITED = 0,
     +           X_LENGTH = 10, Y_LENGTH = 10)

      integer*2 array_data(X_LENGTH, Y_LENGTH)
      integer*2 append_data(X_LENGTH)
      integer*2 i, j

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

C     Define the rank and dimensions of the array. Make the
C     last dimension appendable by defining it's length as unlimited.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = SD_UNLIMITED

C     Create the array data set.
      sds_id = sfcreate(sd_id, 'Ex_File_6', DFNT_INT16, rank, dims)

C     Store the array values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j
10       continue
20    continue

C     Write the data to the array.
      start(1) = 0
```

```
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Perform the initial write to the data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

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

C     Reopen the file and initiate the SD interface in preparation
C     for appending data. Then select the first data set.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)
      sds_idx = sfn2index(sd_id, 'Ex_File_6')
      sds_id = sfselect(sd_id, sds_idx)

C     Store the array values to be appended to the data set.
      do 30 i = 1, X_LENGTH
         append_data(i) = i
30    continue

C     Define the location of the append to start at the first
C     column of the third row and to stop at the end of the third row.
      start(1) = 0
      start(2) = Y_LENGTH
      edges(1) = X_LENGTH
      edges(2) = 1

C     Append the stored data to the data set.
      status = sfwdata(sds_id, start, stride, edges, append_data)

C     Terminate access to the array data set.
      status = sfendacc(sds_id)

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

      end
```

3.5.3 Compressing SD SDS Array Data: SDsetcompress

Uncompressed SDS array data is compressed, or new compressed data sets are created, by calling the **SDsetcompress** routine. **SDsetcompress** compresses the data set data at the time it is called (not during the next call to **SDwritedata**)and supports all standard HDF compression algorithms. These algorithms are:

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

In the future, the following algorithms may be included: Lempel/Ziv-78 dictionary coding, an arithmetic coder, and a faster Huffman algorithm.

The **SDsetcompress** routine is a simplified interface to the **HCcreate** routine, and should be used instead of **HCcreate** unless the user is familiar with working with the lower-level routines. All essential compression functionality is provided by **SDsetcompress**.

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

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

The `comp_type` parameter is the compression type definition and is set to `COMP_CODE_JPEG` for JPEG compression, `COMP_CODE_RLE` for run-length encoding, `COMP_CODE_DEFLATE` for Gnu ZIP (or GZIP) compression, `COMP_CODE_SKPHUFF` for skipping Huffman or `COMP_CODE_NONE` for no compression.

The `c_info` parameter is a pointer to a union structure of type `coder_info`.If `comp_type` is set to `COMP_CODE_NONE` or `COMP_CODE_RLE`, this is unused and can be set to `NULL`. If it's set to `COMP_CODE_SKPHUFF`, the `skphuff` structure in the `coder_info` union must be provided information about the size, in bytes, of the data elements. If it's set to `COMP_CODE_DEFLATE`, the deflate structure in the `coder_info` union must be provided information about the compression "effort".

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

```
coder_info c_info;
c_info.skphuff.skp_size = sizeof(uint16);
status = SDsetcompress(sds_id, COMP_CODE_SKPHUFF, &c_info);
```

To compress a data set using the gzip deflation algorithm, with the maximum "effort" level designated, the following definition and **SDsetcompress** call is used.

```
coder_info c_info;
c_info.deflate_level = 9;
status = SDsetcompress(sds_id, COMP_CODE_DEFLATE, &c_info);
```

To compress a data set using the JPEG algorithm, with the quality level of 80 designated and the "force baseline" parameter enabled, the following definition and **SDsetcompress** call is used.

```
coder_info c_info;
c_info.quality = 80;
c_info.force_baseline = TRUE;
status = SDsetcompress(sds_id, COMP_CODE_JPEG, &c_info);
```

SDsetcompress functionality is currently limited to creating new datasets or appending new slabs onto existing datasets. Overwriting existing dataset data will be supported in the future.

Note that there is currently no Fortran-77 version of the **SDsetcompress** routine. It will be included in a future release.

TABLE 3E

SDsetcompress Parameter List

Routine Name	Parameter	Data Type	Description
		C	

SDsetcompress	sds_id	int32	Data set identifier.
	comp_type	char *	Compression method.
	cinfo	comp_info *	Pointer to compression information structure.

3.5.3.1 Rules for Writing to a Compressed Data Set

Due to certain limitations in the way that compressed data sets are stored, they aren't writable in the way that uncompressed data sets are. The "rules" for writing to a compressed data set are the following:

1. Write the compressed data, in its entirety, to the data set. Build the data set in-core, then write it to the data set in a single write operation.
2. Append to a compressed data set. In other words, write to a compressed data set along it's unlimited dimension. If an unlimited dimension hasn't been defined for the data set, it can't be appended to in this way.
3. Write the compressed data, in its entirety, to any chunk in a chunked SDS.

These rules imply that it is impossible to overwrite subsets of non-chunked data sets. This is because the existing compression algorithms supported by HDF don't allow partial modification to a compressed datastream.

3.5.4 External File Operations

An ***external SDS array*** is an array stored in a file separate from the file containing the metadata for the array. With external arrays, it is possible to link data sets in the same HDF file to multiple external files or data sets in different HDF files to the same external file. Routines for writing external SDS arrays are only available in the C interface and can only be used with HDF files. Unidata-formatted netCDF files are not supported by these routines.

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

There are two methods for creating external SDS arrays. The user can create a new data set in an external file or move data from an existing internal data set to an external file. In either case, only the array values are stored externally, all other data set information remains in the primary HDF file. When an external array is created, a sufficient amount of space is reserved in the external file for the entire data set. The data set will begin at the specified byte offset and extend the length of the data set. The write operation will overwrite the target locations in the external file. The external file may be of any format, provided the data types, byte ordering, and dimension ordering are supported by HDF. The primary file must be an HDF file.

3.5.4.1 Specifying the Directory Search Path of an External File: HXsetdir

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

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

Until the **HXsetdir** routine is called, either the directories specified by the $HDFEXTDIR environment variable are searched if the variable has been set, or Item 3 will be the search method used. If the **HXsetdir** routine hasn't been called and the $HDFEXTDIR variable hasn't been set, Item 3 will again be the search method used. Setting the $HDFEXTDIR environment variable effectively ensures that Item 3 will never be used as a default method as it can't be unset within a calling program.

HXsetdir has one argument, a string specifying the directory list to be searched. This list can consist of one directory name or a set of directory names separated by colons. If **HXsetdir** is passed a null string and the $HDFEXTDIR environment variable has been set, the directories specified by $HDFEXTDIR are searched, or Item 3 will be the search method used if $HDFEXTDIR hasn't been set.

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

The parameters of **HXsetdir** are described further in the following table.

TABLE 3F **HXsetdir Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
HXsetdir (**hxsdir**)	dir_list	char *	character* (*)	Directory list to be searched.
	dir_length	None	integer	Length of the `dir_list` string.

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

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

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

HXsetcreatedir has one argument: the directory location of the next external file to be created. If an error is encountered, the directory location is left unchanged.

The parameters of **HXsetcreatedir** are described further in the following table.

TABLE 3G **HXsetcreatedir Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
HXsetcreatedir (**hxsdir**)	dir	char *	character* (*)	Directory location of the next external file to be created.
	dir_length	None	integer	Length of the `dir` string.

3.5.4.3 Creating a Data Set in an External File: SDsetexternalfile

Creating a data set using external data involves the following steps:

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

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

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

FORTRAN:
```
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
status = sfsextf(sds_id, filename, offset)
status = sfwdata(sds_id, start, stride, edge, data)
status = sfendacc(sds_id)
```

SDsetexternalfile marks the SDS identified by `sds_id` as one whose data is to be written to an external file. The parameter `filename` is the name of the external data file, and `offset` is the number of bytes from the beginning of the external file to the location where the first byte of data should be written.

When used in conjunction with **SDcreate**, **SDsetexternalfile** does not actually write data to an external file. Instead it marks the data set as an external data set for all subsequent **SDwritedata** operations. **SDsetexternalfile** can only be called once after a data set has been created.

If a file with the same name as `filename` exists in the current directory search path, HDF will access it as the external file. If the file does not exist, HDF will create one in the directory named in the last call to **HXsetcreatefile**. If an absolute pathname is specified, the external file will be created at the location specified by the pathname, overriding the location specified by the last call to **HXsetcreatefile**.

Once the name of an external file is established, it is impossible to change it without breaking the association between the data set's metadata and the data it describes.

Use caution when writing to existing external or primary files as the HDF library starts the write operation at the specified offset without checking if data is being overwritten.

For more information on the parameters used in **SDsetexternalfile** refer to the following table.

TABLE 3H **SDsetexternalfile Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetexternalfile (sfsextf)	sds_id	int32	integer	Data set identifier.
	filename	char *	character* (*)	Name of the external data set.
	offset	int32	integer	Offset in bytes from the beginning of the external file to the SDS data.

3.5.4.4 Moving Data to an External File

Data can be moved from a primary file to an external file. To do so requires the following steps:

1. Select the array.
2. Specify the external data file.

3. Terminate access to the data set.

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

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

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

When **SDsetexternalfile** is used in conjunction with **SDselect**, it will immediately write the existing data to the external file. Any data in the external file that occupies the space reserved for the external array will be overwritten as a result of this operation. Data can only be moved to an external file once after the parent data set has been created and the data set must first exist in the primary HDF file. During the operation, the data is written to the external file as a contiguous stream regardless of how it is stored in the primary file. Because data is moved “as is”, any unwritten locations in the data set are preserved in the external file. Subsequent read and write operations performed on the data set will access the external file.

EXAMPLE 7.

Writing SDS Data to an HDF File Starting at an Offset

A subset of values stored in an SDS can be moved from an HDF file to an external file, as shown in the following examples.

C:

```
#include "hdf.h"

main( )
{

int32 sd_id, sds_id, offset, status;

/* Open the file. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

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

/* Create a file named "subfile1" and move the data set values into
 *  it, starting at byte location 24. */
offset = 24;
status = SDsetexternalfile(sds_id, "subfile1", offset);

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

}
```

FORTRAN:

```
      PROGRAM WRITE EXTFILE

       integer*4 sd_id, sds_id, offset, DFACC_RDWR
       integer status
       integer sfstart, sfselect, sfsextf, sfendacc, sfend
```

```
C     DFACC_RDWR is defined in hdf.h.
      parameter (DFACC_RDWR = 3)

C     Open the HDF file.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

C     Get the identifier of the first data set.
      sds_id = sfselect(sd_id, 0)

C     Create a file named "subfile1" and move the data set
C     into it, starting at byte location 24.
      offset = 24
      status = sfsextf(sds_id, 'subfile1', offset)

C     Dispose of the data set identifier to terminate access.
      status = sfendacc(sds_id)

C     Dispose of the file identifier to close the file.
      status = sfend(sd_id)

      end
```

3.6 Reading Data from an SDS Array: SDreaddata

Selecting an SD data set and reading one or more slabs from it involves the following steps:

1. Select an SDS.
2. Read a slab or series of slabs.

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

C:

```
sds_id = SDselect(sd_id, sds_index);
status = SDreaddata(sds_id, start, stride, edge, data);
```

FORTRAN:

```
sds_id = sfselect(sd_id, sds_index)
status = sfrdata(sds_id, start, stride, edge, data)
```

SDreaddata reads an entire array, slabs of an array or it can be used to read a subset of the array. For more information on reading attributes or scales see Section 3.10.3 on page 84 and Section 3.9.4.3 on page 75.

The `sds_id` argument is the SDS id returned by **SDcreate** or **SDselect**. As with **SDwritedata,** the arguments `start`, `stride`, and `edge` respectively describe the n-dimensional coordinate of the slab the SD interface will begin the read operation, number of locations the current SDS location will be moved forward after each read, and the length of each dimension of the subset to be read. If the SDS array is smaller than the `data` argument array, the amount of data read will be truncated to the size of the SDS array. For additional information on the `start`, `stride` and `edge` parameters refer to Section 3.5.2 on page 28.

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.

The parameters of **SDreaddata** are further described in the following table. Note that, because there are two Fortran-77 versions of **SDreaddata**, there are correspondingly two entries in the "Data Type" field of the *data* parameter.

TABLE 3I

SDreaddata Parameter List

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

EXAMPLE 8.

Reading an Entire SDS

When **SDreaddata** is used to read an entire n-dimensional SDS array, the coordinates for the start position must begin at 0 for each dimension (`start={0,0, ... 0}`), the interval between each read must equal 1 for each dimension (`stride=NULL or stride={1,1, ... 1}`), and the size of each dimension must equal the size of the array itself (`edge={dim_size_1, dim_size_2, ... dim_size_n}`). The `data` buffer must have enough preallocated space to hold the data.

C:

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

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{
int32 sd_id, sds_id, status;
int32 start[3], edges[3];
int16 array_data[Z_LENGTH][Y_LENGTH][X_LENGTH];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Select the first (and in this case, only) data set in the file. */
sds_id = SDselect(sd_id, 0);

/* Define the location, pattern, and size of the data to read. */
start[0] = start[1] = start[2] = 0;
edges[0] = Z_LENGTH;
edges[1] = Y_LENGTH;
edges[2] = X_LENGTH;

/* Read the array. */
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array */
status = SDendaccess(sds_id);

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

}
```

FORTRAN:

```
      PROGRAM SEARCH INDEX

       integer*4 gr_id, gr_index, ri_id, file_id, status
       integer*4 start(2), edges(2), stride(2)
       integer mgstart, mgn2ndx, mgrdimg, mgendac, mgselct
       integer mgend, hopen, hclose

C      DFACC_RDONLY is defined in hdf.h.
       integer*4 DFACC_RDONLY
       integer*4 X_LENGTH, Y_LENGTH
       parameter (DFACC_RDONLY = 1, X_LENGTH = 10, Y_LENGTH = 20)
       integer*2 image_data(2, X_LENGTH, Y_LENGTH)

C      Open the file.
       file_id = hopen('Example5.hdf', DFACC_RDONLY, 0)

C      Initialize the GR interface.
       gr_id = mgstart(file_id)

C      Search for the index of a non-existent image.
       gr_index = mgn2ndx(gr_id, 'Invalid_Data_Set_Name')

C      Error condition: gr_index contains the value -1.
C      Search for the index of the image named 'Image_array_5'.
       gr_index = mgn2ndx(gr_id, 'Image_array_5')

C      Select the image corresponding to the returned index.
       ri_id = mgselct(gr_id, gr_index)

C      Read the image data into the image_data array.
       start(1) = 0
       start(2) = 0
       edges(1) = X_LENGTH
       edges(2) = Y_LENGTH
       stride(1) = 1
       stride(2) = 1
       status = mgrdimg(ri_id, start, stride, edges, image_data)

C      Terminate access to the array.
       status = mgendac(ri_id)

C      Terminate access to the GR interface and the file.
       status = mgend(gr_id)

C      Close the file.
       status = hclose(file_id)

       end
```

EXAMPLE 9.

Reading a Subset of an SDS Array

The `start` and `edges` arguments of the **SDreaddata** function can be used to read a subset of the entire data set, as shown in the following examples.

C:

```
#include "hdf.h"
```

```
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id;
int32 start[2], edges[2];
int16 all_data[Y_LENGTH][X_LENGTH], subset_data[7][3], status;

/* Open the file for read-only access. */
sd_id = SDstart("Example3.hdf", DFACC_RDONLY);

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

/* First, read the entire data set. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)all_data);

/* Read a subset of the data set. */
start[0] = 1;
start[1] = 1;
edges[0] = 7;
edges[1] = 3;
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)subset_data);

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

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

}
```

FORTRAN:

```
     PROGRAM READ SUBSET

      integer*4 sd_id, sds_id
      integer start(2), edges(2), stride(2), status
      integer sfstart, sfselect, sfrdata, sfendacc, sfend

C     DFACC_RDONLY is defined in hdf.h.
      integer*4 DFACC_RDONLY
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDONLY = 1, X_LENGTH = 5, Y_LENGTH = 16)
      integer*2 all_data(X_LENGTH, Y_LENGTH)
      integer*2 subset_data(3, 7)

C     Open the file.
      sd_id = sfstart('Example3.hdf', DFACC_RDONLY)

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

C     Read the entire data set.
      start(1) = 0
      start(2) = 0
```

```
      edges(1) = 5
      edges(2) = 16
      stride(1) = 1
      stride(2) = 1
      status = sfrdata(sds_id, start, stride, edges, all_data)

C     Read a subset from the middle of the data set.
      start(1) = 1
      start(2) = 1
      edges(1) = 3
      edges(2) = 7
      status = sfrdata(sds_id, start, stride, edges, subset_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

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

      end
```

EXAMPLE 10.

Sampling SDS Data

These examples demonstrate how samples of data set elements can be read by using the `stride` argument of the **SDreaddata** function. Here we read every tenth row and every other column.

C:

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

#define X_LENGTH 10
#define Y_LENGTH 20

main( )
{

int32 sd_id, sds_id, rank, start[2], edges[2], stride[2], dims[2];
int16 all_data[Y_LENGTH][X_LENGTH], sample_data[5][5];
intn i, j;
int16 status;

/* Open the file. */
sd_id = SDstart("Example10.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 2;
dims[0] = Y_LENGTH;
dims[1] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_10", DFNT_INT16, rank, dims);

/* Compute and store the data values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        all_data[j][i] = i + j * 10;
}

/* Define the start and edge parameters. */
```

```
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Write the buffered data in all_data to the data set. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)all_data);

/* Close the SD interface and the file, then re-open both and \
select the first and only data set in the file. */

status = SDendaccess(sds_id);
status = SDend(sd_id);

sd_id = SDstart("Example10.hdf", DFACC_RDONLY);
sds_id = SDselect(sd_id, 0);

/* Read the data into the sample_data array, skipping every fourth \
row and every other column. */
start[0] = start[1] = 0;
edges[0] = 5;
edges[1] = 5;
stride[0] = 4;
stride[1] = 2;

status = SDreaddata(sds_id, start, stride, edges, (VOIDP)sample_data);

/* Terminate access to the array */
status = SDendaccess(sds_id);

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

}
```

FORTRAN:

```
     PROGRAM READ STRIDES

      integer*4 sd_id, sds_id, rank
      integer*4 start(2), edges(2), stride(2), dims(2)
      integer i, j, status
      integer sfstart, sfcreate, sfwdata, sfrdata, sfendacc
      integer sfselect, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFACC_RDONLY, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_CREATE = 4, DFACC_RDONLY = 1, DFNT_INT16 = 22,
     +            X_LENGTH = 10, Y_LENGTH = 20)
      integer*2 all_data(X_LENGTH, Y_LENGTH), sample_data(5, 5)

C     Create the file.
      sd_id = sfstart('Example10.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the array to be created.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the array.
      sds_id = sfcreate(sd_id, 'Ex_array_10', DFNT_INT16, rank, dims)

C     Compute and store the data values.
```

```
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            all_data(i, j) = i + j * 10
10          continue
20    continue

C     Set up the start and edge parameters to write the buffered
C     data to the entire array data set.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the buffered data in wrt_data to the data set.
      status = sfwdata(sds_id, start, stride, edges, all_data)

C     Close the SD interface and the file, then re-open both and
C     select the first and only data set in the file.
      status = sfendacc(sds_id)
      status = sfend(sd_id)
      sd_id = sfstart('Example10.hdf', DFACC_RDONLY)
      sds_id = sfselect(sd_id, 0)

C     Read the data into the sample_data array, skipping every fourth
C     row and every other column.
      edges(1) = 5
      edges(2) = 5
      stride(1) = 2
      stride(2) = 4
      status = sfrdata(sds_id, start, stride, edges, sample_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

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

      end
```

3.6.1 Reading Data from an External File

SDS data is read from an external file in the same way that it is read from a primary file. Whether the SDS data is or is not stored in an external file is transparent to the user.

3.7 Obtaining Information About SD Data Sets

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

SDfileinfo obtains the number of SDSs and file attributes in a file, and **SDgetinfo** provides information about individual SDSs. To cycle through the data sets in a file, a calling program must use **SDfileinfo** to determine the number of data sets, followed by repeated calls to **SDgetinfo** to view them. The parameters of these two routines are described below. (See Table 3J on page 53.)

At times you might need to search through a file for an SDS for a given name or reference number. The **SDnametoindex** routine determines the index of an SDS from its name, and **SDreftoindex** determines the index of an SDS from its reference number. The parameters of these two routines are described below. (See Table 3K on page 56.)

3.7.1 Obtaining Information About the SDSs in a File: SDfileinfo

It is often useful to determine the number of scientific data sets and global SDS attributes contained in a file before executing a series of read or write operations. The **SDfileinfo** routine is designed for this purpose. To determine the contents of a file, the calling program must contain the following calls:

C:
```
sd_id = SDstart(filename, access_mode);
status = SDfileinfo(sd_id, n_datasets, n_file_attr);
status = SDend(sd_id);
```

FORTRAN:
```
sd_id = sfstart(filename, access_mode)
status = sffinfo(sd_id, n_datasets, n_file_attr)
status = sfend(sd_id)
```

SDfileinfo uses `n_datasets` to return the number of scientific data sets in the file, and `n_file_attr` to return the number of file attributes in the file.

3.7.2 Obtaining Information About a Specific SDS: SDgetinfo

Some information may be needed before reading and working with SDS arrays. For instance, if the rank, dimension sizes and/or data type of an array are unknown, it may be impossible to allocate the proper amount of memory to work with the array. The **SDgetinfo** routine provides basic information about SDS arrays.

SDgetinfo takes an SDS id as input, and returns the name, rank, dimensions, data type, and number of attributes for the corresponding SDS. The attribute count will only reflect the number of attributes assigned to the SDS array; file attributes are not included.

TABLE 3J **SDfileinfo and SDgetinfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDfileinfo (sffinfo)	file_id	int32	integer	File identifier.
	n_datasets	int32 *	integer	Number of data sets in the file.
	n_file_attr	int32 *	integer	Number of global attributes in the file.
SDgetinfo (sfginfo)	sds_id	int32	integer	Data set identifier
	sds_name	char*	character* (*)	Space to put the name of the data set.
	rank	int32 *	integer	Space to put the number of dimensions in the array.
	dim_sizes	int32 []	integer (*)	Space to put the size of each dimension in the array.
	data_type	int32 *	integer	Space to put the data type for the data in the array.
	nattrs	int32 *	integer	Space to put the number of attributes in the data set.

EXAMPLE 11. **Printing Data Set Names**

The **SDgetinfo** function can be called within a loop to retrieve the names of all data sets in an HDF file, as shown in the following examples.

C:

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

#include <stdio.h>

main( )
{

int32 sd_id, sds_id, n_datasets, n_file_attrs, index, status;
int32 dim_sizes[MAX_VAR_DIMS];
int32 rank, num_type, attributes;
char name[MAX_NC_NAME];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Determine the contents of the file. */
status = SDfileinfo(sd_id, &n_datasets, &n_file_attrs);

/* Access and print the name of every data set in the file. */
for (index = 0; index < n_datasets; index++) {
    sds_id = SDselect(sd_id, index);
    status = SDgetinfo(sds_id, name, &rank, dim_sizes, \
                &num_type, &attributes);

    printf("name = %s\n", name);
    status = SDendaccess(sds_id);
}

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

}
```

FORTRAN:

```
     PROGRAM PRINT NAMES

      integer*4 sd_id, sds_id
      integer*4 n_datasets, n_file_attrs, index
      integer status, attributes
      integer sfstart, sffinfo, sfselect, sfginfo
      integer sfendacc, sfend

C     DFACC_RDONLY is defined in hdf.h. MAX_NC_NAME and MAX_VAR_DIMS
C     are defined in netcdf.h.
      integer*4 DFACC_RDONLY, MAX_NC_NAME, MAX_VAR_DIMS
      parameter (DFACC_RDONLY = 1, MAX_NC_NAME = 256,
     +            MAX_VAR_DIMS = 32)

      integer*4 dim_sizes(MAX_VAR_DIMS)
      character name *(MAX_NC_NAME)

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C     Determine the contents of the file.
      status = sffinfo(sd_id, n_datasets, n_file_attrs)

C     Access and print the names of every data set in the file.
```

```
      do 10 index = 0, n_datasets - 1
         sds_id = sfselect(sd_id, index)
         status = sfginfo(sds_id, name, rank, dim_sizes, num_type,
     +                    attributes)
         print *, "name = ", name
         status = sfendacc(sds_id)
10    continue

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

      end
```

3.7.3 Locating a SDS Data Set by Name: SDnametoindex

When an SDS is created, a unique number is assigned to it by the SD library so that it can be located in the file for future access. This number is referred to as the ***index*** of an SDS.

If the index of an SDS in an HDF file is known, **SDselect** can be used immediately. If not, it is necessary to determine the index by some other means. **SDnametoindex** determines the index of an SDS from its name. Selecting a data set by name involves the following steps:

1. Convert the SDS name into a valid index number.
2. Select the SDS by obtaining its identifier from its index number.

To access a SDS via its name, the calling program must contain the following function calls:

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

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

SDnametoindex returns the index specified by the `sds_index` parameter for the first data set in the file with the name `sds_name`. If two data sets have the same name, **SDnametoindex** returns the index of the first data set. The index `sds_index` can then be used by **SDselect** to obtain an SDS id for the specified data set.

The **SDnametoindex** routine is case-sensitive and does not accept wildcards.

3.7.4 Locating an SDS by Reference Number: SDreftoindex

In addition to an index and name, data sets are also assigned a tag and reference number. **SDreftoindex** determines the index of the SDS from its reference number. Selecting a data set by reference number involves the following steps:

1. Convert the reference number for the SDS into a valid index number.
2. Select the SDS by obtaining its identifier from its index number.

To select a data set by reference number, the calling program must execute the following calls:

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

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

SDreftoindex returns the index specified by the `sds_index` parameter for the SDS in the file with the reference number `ref`. The index `sds_index` can then be passed to **SDselect** to obtain an SDS id for the SDS.

Because the HDF library assigns reference numbers in unpredictable ways, reference numbers should be used to identify SDSs only when no other means are available. Reference numbers do not necessarily adhere to any ordering scheme.

TABLE 3K

SDnametoindex and SDreftoindex Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDnametoindex (sfn2index)	sd_id	int32	integer	SD interface identifier.
	sds_name	char *	character* (*)	Name of the data set to index.
SDreftoindex (sfref2index)	sd_id	int32	integer	SD interface identifier.
	ref	int32	integer	IN: Reference number for the specified data set.

EXAMPLE 12.

Searching for the Index of an SDS

In these examples, an invalid data set name is specified which results in **SDnametoindex** returning a value of `-1`. A valid data set name is then provided and the returned index is used to read the contents of the corresponding data set.

C:

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

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{

int32 sd_id, sds_index, sds_id, status;
int32 start[3], edges[3];
int16 array_data[Z_LENGTH][Y_LENGTH][X_LENGTH];

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Search for the index of a non-existent array data set. */
sds_index = SDnametoindex(sd_id, "Invalid_Data_Set_Name");

/* Error condition: sds_index contains the value -1. */

/* Search for the index of a "Ex_array_4" array data set. */
sds_index = SDnametoindex(sd_id, "Ex_array_4");

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

/* Read the data set data into the array_data array. */
start[0] = start[1] = start[2] = 0;
edges[0] = Z_LENGTH;
edges[1] = Y_LENGTH;
edges[2] = X_LENGTH;

status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array */
```

```
status = SDendaccess(sds_id);

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

}
```

FORTRAN:

```
      PROGRAM SEARCH INDEX

       integer*4 sd_id, sds_index, sds_id, status
       integer*4 start(3), edges(3), stride(3)
       integer sfstart, sfn2index, sfrdata, sfendacc, sfselect

C      DFACC_RDONLY is defined in hdf.h.
       integer*4 DFACC_RDONLY
       integer*4 X_LENGTH, Y_LENGTH, Z_LENGTH
       parameter (DFACC_RDONLY = 1, X_LENGTH = 4, Y_LENGTH = 5,
      +           Z_LENGTH = 6)
       integer*2 array_data(X_LENGTH, Y_LENGTH, Z_LENGTH)

C      Open the file.
       sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C      Search for the index of a non-existent array data set.
       sds_index = sfn2index(sd_id, 'Invalid_Data_Set_Name')

C      Error condition: sds_index contains the value -1.

C      Search for the index of a 'Ex_array_4' array data set.
       sds_index = sfn2index(sd_id, 'Ex_array_4')

C      Select the data set corresponding to the returned index.
       sds_id = sfselect(sd_id, sds_index)

C      Read the data set data into the rd_data array.
       start(1) = 0
       start(2) = 0
       start(3) = 0
       edges(1) = X_LENGTH
       edges(2) = Y_LENGTH
       edges(3) = Z_LENGTH
       stride(1) = 1
       stride(2) = 1
       stride(3) = 1
       status = sfrdata(sds_id, start, stride, edges, array_data)

C      Terminate access to the array
       status = sfendacc(sds_id)

       end
```

3.7.5 Creating SDS Arrays Containing Variable-Length Data: SDsetnbitdataset

Version 4.0r1 of HDF provided the **SDsetnbitdataset** routine, allowing the HDF user to specify that a particular SDS array contains data of a non-standard length. Any length between 1 and 32 bits can be specified. After **SDsetnbitdataset** has been called for the SDS array, any read or write operations will involve a conversion between the new data length of the SDS array and the data length of the read or write buffer.

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

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

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

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 value of `1` would denote 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 is set to either `TRUE` or `FALSE` - specify `TRUE` to sign-extend.

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

The "background" bits of a variable-length 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.

SDsetnbitdataset returns `SUCCEED` (or `0`) upon successful completion and `FAIL` (or `-1`) otherwise.

TABLE 3L

SDsetnbitdataset Parameter List

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

3.8 Chunked (or Tiled) Scientific Data Sets

NOTE: It is strongly encouraged that HDF users who wish to use the SD chunking routines first read the section on SD chunking in Chapter 13, titled *HDF Performance Issues*. In that section the concepts of chunking are explained, as well as their use in relation to HDF. As the ability to work with chunked data has been added to HDF functionality for the purpose of addressing specific performance-related issues, the user should first have the necessary background knowledge to correctly determine how chunking will positively or adversely affect their application.

This chapter will refer to both "tiled" and "chunked" SDSs as simply "chunked SDSs", as tiled SDSs are the two-dimensional case of chunked SDSs.

3.8.1 Making a Non-Chunked SDS a Chunked SDS: SDsetchunk

In HDF, an SDS must first be created as a generic SDS through the **SDcreate** routine, then "promoted" to be a chunked SDS through the use of **SDsetchunk**. **SDsetchunk** determines the chunk size and the compression method, if any, to be applied when accessing the chunks.

SDsetchunk receives its information on how to create the chunks, as well as compression information, from an `HDF_CHUNK_DEF` union passed in as its second argument. This union structure is defined in the HDF library as follows:

```
typedef union hdf_chunk_def_u {
    int32 chunk_lengths[MAX_VAR_DIMS];
    struct {
        int32 chunk_lengths[MAX_VAR_DIMS];
        int32 comp_type;
        comp_info cinfo;
    } comp;
}HDF_CHUNK_DEF
```

Note that there is currently no Fortran-77 version of **SDsetchunk**.

TABLE 3M

SDsetchunk Parameter List

Routine Name	Parameter	Data Type	Description
		C	
SDsetchunk	sds_id	int32	SD identifier.
	c_def	HDF_CHUNK_DEF	Union containing information on how the chunks are to be defined.
	flags	int32	Flags determining the behavior of the routine.

The `flags` parameter can either be set to `HDF_CHUNK` if the SDS is to be uncompressed, or to the bitwise-OR'ed values of `HDF_CHUNK` and `HDF_COMP` (`HDF_CHUNK | HDF_COMP`) if a compression method is to be applied. In the former case the first definition of the `chunk_lengths` array should be set to the dimensions of the chunks. For example, given the definition of `HDF_CHUNK_DEF` stated above, a `flags` parameter value of `HDF_CHUNK` and the following definition of a three-dimensional chunked SDS

```
HDF_CHUNK_DEF current_chunk_def;

current_chunk_def.chunk_lengths[0] = 2;
current_chunk_def.chunk_lengths[1] = 3;
current_chunk_def.chunk_lengths[2] = 4;
```

the size of the chunk passed to **SDsetchunk** will be 2 bytes by 3 bytes by 4 bytes.

In the latter case (where a compression method is specified), the `chunk_lengths` array definition within the `comp` structure is initialized in the way described above and the `cinfo` structure is initialized in the same way as for the **SDsetcompress** routine. (See Figure 3.5.3 on page 40.)

There are two restrictions that apply to chunked SDSs. The maximum number of chunks in a single HDF file is 65,535, and a chunked SDS cannot contain an unlimited dimension.

SDsetchunk returns either a value of `SUCCEED` or `FAIL`.

3.8.2 Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache

To maximize the performance of the HDF library routines when working with chunked SDSs, the library maintains a separate area of memory specifically for cached data chunks. **SDsetchunkcache** determines the maximum number of chunks of the specified SDS that are cached into this segment of memory.

When the chunk cache has been filled, any additional chunks written to cache memory are cached according to the LRU, or Least-Recently-Used, algorithm. This means that the chunk that has resided in the cache the longest without being reread or rewritten will be written over with the new chunk.

Note that there is currently no Fortran-77 equivalent of **SDsetchunkcache**.

TABLE 3N **SDsetchunkcache Parameter List**

Routine Name	Parameter	Data Type C	Description
SDsetchunkcache	sds_id	int32	SD identifier.
	maxcache	int32	Maximum number of chunks to cache.
	flags	int32	Flags determining the default caching behavior.

By default, when a generic SDS is promoted to be a chunked SDS, the `maxcache` parameter is set to the number of chunks along the last dimension.

If the chunk cache is full and the value of the `maxcache` parameter is larger than the currently allowed maximum number of cached chunks, then the maximum number of cached chunks is reset to the value of `maxcache`. If the chunk cache is not full, then the size of the chunk cache is reset to the value of `maxcache` only if it is greater than the current number of chunks in the cache.

Currently the only allowed value of the flag parameter is `0`, which designates default operation. In the near future, the value `HDF_CACHEALL` will be supported to be used to specify that the whole SDS object is to be cached.

SDsetchunkcache returns either the set value of the maximum number of cacheable chunks or `FAIL`.

3.8.3 Writing Data to Chunked SDSs: SDwritechunk and SDwritedata

If an SDS has been created as a chunked SDS (i.e., has been created through calls to **SDcreate** and **SDsetchunk**), both **SDwritedata** and **SDwritechunk** can be used to write data to it. There are situations where **SDwritechunk** may be a more appropriate routine to use than **SDwritedata**, but both routines essentially achieve the same results.

The location of data in a chunked SDS can be referenced in two ways. The first is the standard method used in the SD routines that access both chunked and non-chunked SDSs, and refers to the starting location, or the *origin*, as an offset in bytes from the origin of the SD array itself. The second method is used by the SD routines that only access chunked SDSs, and refers to the origin of the chunk as an offset in chunks from the origin of the SD array itself. See the section on SD chunking in Chapter 13, titled *HDF Performance Issues*, for an illustration of this.

SDwritechunk is used when an entire chunk is to be written and requires that the chunk offset be known. **SDwritedata** is used when the write operation is to be done regardless of the chunking scheme used in the SDS and the byte offset of the target chunk is known. Also, as **SDwritechunk** is written specifically for chunked SDSs and doesn't have the overhead of the additional functionality supported by the **SDwritedata** routine, it is much faster than **SDwritedata**.

The parameters of **SDwritedata** are listed above. (See Table 3D on page 30.) The parameters of **SDwritechunk** are as follows. Note that there currently is no Fortran-77 equivalent of **SDwritechunk**.

TABLE 3O **SDwritechunk Parameter List**

Routine Name	Parameter	Data Type	Description
		C	
SDwritechunk	sds_id	int32	SD identifier.
	origin	int32 *	Origin of the chunk to be written.
	datap	const VOID *	Buffer containing the data to be written.

The `datap` parameter must point to an array containing an entire chunk of data - in other words, the size of the array must be the same as the chunk size of the SDS to be written to. An error condition will result if this is not the case.

SDwritechunk returns either a value of `SUCCEED` or `FAIL`.

3.8.4 Reading Data From Chunked SDSs: SDreadchunk and SDreaddata

As both **SDwritedata** and **SDwritechunk** can be used to write data to chunked SDSs, both **SDreaddata** and **SDreadchunk** can be used to read data from chunked SDSs.

SDreadchunk is used when an entire chunk of data is to be read. **SDreaddata** is used when the read operation is to be done regardless of the chunking scheme used in the SDS. Also, **SDreadchunk** is written specifically for chunked SDSs and doesn't have the overhead of the additional functionality supported by the **SDreaddata** routine - therefore, it is much faster than **SDreaddata**.

SDreadchunk returns either a value of `SUCCEED` or `FAIL`.

The parameters of **SDreaddata** are listed above. (See Table 3I on page 47.) The parameters of **SDreadchunk** are as follows. Note that there is currently no Fortran-77 equivalent of **SDreadchunk**.

TABLE 3P

SDreadchunk Parameter List

Routine Name	Parameter	Data Type	Description
		C	
SDreadchunk	sds_id	int32	SD identifier.
	origin	int32 *	Origin of the chunk to be read.
	datap	VOID *	Buffer for the returned chunk data.

As with **SDwritechunk**, the `datap` parameter must point to an array containing enough space for an entire chunk of data. In other words, the size of the array must be the same as the chunk size of the SDS to be written to. An error condition will result if this is not the case.

3.8.5 Obtaining Information About a Chunked SDS: SDgetchunkinfo

SDgetchunkinfo is used to determine how the chunks in a chunked SDS are defined and whether the SDS is chunked or not.

Information about the chunks is returned in the `HDF_CHUNK_DEF` union provided as the second parameter - therefore, it will return the same information that can be set in the **SDsetchunk** routine. For example, the `flags` argument will return either `HDF_CHUNK` for an uncompressed chunked SDS, `HDF_CHUNK` bitwise-OR'ed with `HDF_COMP` for a compressed and chunked SDS or `HDF_NONE` for a non-chunked SDS. A pointer to a `chunk_lengths` array containing chunk dimension size information will be returned if the returned `flags` param value is `HDF_CHUNK`, a pointer to a `comp` structure containing chunk dimension size and compression information will be returned if the returned `flags` param value is `HDF_CHUNK` bitwise-OR'ed with `HDF_COMP`.

A `NULL` value can also be passed in as the `c_def` param if chunking information is not desired.

Note that there is currently no Fortran-77 equivalent of **SDgetchunkinfo**.

TABLE 3Q

SDgetchunkinfo Parameter List

Routine Name	Parameter	Data Type	Description
		C	
SDgetchunkinfo	sds_id	int32	SD identifier.
	c_def	HDF_CHUNK_DEF *	Union structure containing information about the chunks in the SDS.
	flags	int32 *	Flags determining the behavior of the routine.

SDgetchunkinfo returns either a value of `SUCCEED` or `FAIL`.

EXAMPLE 13.

Writing and Reading Chunked Data Using SDwritechunk and SDreaddata

This example creates a 9-by-4 integer chunked uncompressed SDS in a file named "Example13.hdf", then writes six 3-by-2 byte chunks of 16-bit unsigned integers to it. It then reads one 5-by-2 16-bit unsigned integer subset. It uses **SDwritechunk** to write the chunks and **SDreaddata** to read the subset.

C:

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

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  chunk1_2u16[6] = {11, 21,
                                 12, 22,
                                 13, 23};

static uint16  chunk2_2u16[6] = {31, 41,
                                 32, 42,
                                 33, 43};

static uint16  chunk3_2u16[6] = {14, 24,
                                 15, 25,
                                 16, 26};

static uint16  chunk4_2u16[6] = {34, 44,
                                 35, 45,
                                 36, 46};

static uint16  chunk5_2u16[6] = {17, 27,
                                 18, 28,
                                 19, 29};

static uint16  chunk6_2u16[6] = {37, 47,
                                 38, 48,
                                 39, 49};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_2u16[5][2];/* Data array read */
uint16  fill_u16 = 0;/* Fill value */
int32 c_flags;

HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example13.hdf", DFACC_CREATE);

/* Create a  9x4 SDS of uint16 in file 1. */
d_dims[0] = 9;
d_dims[1] = 4;

sdsid = SDcreate(f1, "DataSetChunked_1", DFNT_UINT16, 2, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS chunk with 3x2 chunks which will create
```

```
6 chunks. */

c_def.chunk_lengths[0] = 3;
c_def.chunk_lengths[1] = 2;

status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 3 chunks */
status = SDsetchunkcache(sdsid, 3, 0);

/* Write the data chunks. */

/* Write chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk1_2u16);

/* Write chunk 4. */
start_dims[0] = 1;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk4_2u16);

/* Write chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk2_2u16);

/* Write chunk 5. */
start_dims[0] = 2;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk5_2u16);

/* Write chunk 3. */
start_dims[0] = 1;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk3_2u16);

/* Write chunk 6. */
start_dims[0] = 2;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk6_2u16);

/* Read a subset of the data back in using SDreaddata
       i.e  5x2 subset of the whole array. */
start_dims[0] = 2;
start_dims[1] = 1;
edge_dims[0] = 5;
edge_dims[1] = 2;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_2u16);

/* This 5x2 array should look like this
         {{23, 24, 25, 26, 27},
          {33, 34, 35, 36, 37}} */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);
```

```
}
```

EXAMPLE 14.

Writing and Reading Chunked Data Using SDwritedata and SDreaddata

This example creates a chunked SDS with a size of 2-by-3-by-4 16-bit unsigned integers in a file named "Example14.hdf", then writes two 2-by-3-by-2 16-bit unsigned integer chunks to it. It then reads one 2-by-3-by-4 16-bit unsigned integer chunk. It uses **SDwritedata** to write the chunks and **SDreaddata** to read the subset.

C:

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

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  u16_3data[2][3][4] =
{
{
    { 0, 1, 2, 3},
    { 10, 11, 12, 13},
    { 20, 21, 22, 23}},
{
    { 100, 101, 102, 103},
    { 110, 111, 112, 113},
    { 120, 121, 122, 123}}};

main( )
{

int32 f1;/* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_3u16[2][3][4];/* Data array read */
uint16  fill_u16 = 0;        /* Fill value */
int32 c_flags;

HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example14.hdf", DFACC_CREATE);

/* Create a new 2x3x4 SDS of uint16 in the file. */
d_dims[0] = 2;
d_dims[1] = 3;
d_dims[2] = 4;
sdsid = SDcreate(f1, "DataSetChunked_2", DFNT_UINT16, 3, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS - chunk is 2x3x2 which will create 2 chunks */
c_def.chunk_lengths[0] = 2;
```

```
c_def.chunk_lengths[1] = 3;
c_def.chunk_lengths[2] = 2;
status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 2 chunks. */
status = SDsetchunkcache(sdsid, 2, 0);

/* Write data using SDwritedata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDwritedata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) u16_3data);

/* Read data using SDreaddata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_3u16);

/* Verify the data in inbuf_3u16 against u16_3data[]. */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down SDS interface. */
status = SDend(f1);

}
```

EXAMPLE 15.

Writing and Reading Chunked Data Using SDwritedata and SDreadchunk

This example creates a chunked SDS with a size of 2-by-3-by-4 16-bit unsigned integers in a file named "Example15.hdf", then writes one 2-by-3-by-4 16-bit unsigned integer chunk to it. It then reads six 1-by-1-by-4 16-bit unsigned integer subsets. It uses **SDwritedata** to write the chunks and **SDreadchunk** to read the subsets.

C:

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

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

static uint16  u16_3data[2][3][4] =
{
```

```
{
    { 0, 1, 2, 3},
    { 10, 11, 12, 13},
    { 20, 21, 22, 23}},
{
    { 100, 101, 102, 103},
    { 110, 111, 112, 113},
    { 120, 121, 122, 123}}};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  ru16_3data[4];/* Whole chunk input buffer */
int32   rcdims[3];/* For SDgetchunkinfo() */
uint16  fill_u16 = 0;/* Fill value */
int32 c_flags;
HDF_CHUNK_DEF c_def, r_def;

int32 status;

/* Create the HDF file. */
f1 = SDstart("Example15.hdf", DFACC_CREATE);

/* Create a new 2x3x4 SDS of uint16 in the file. */
d_dims[0] = 2;
d_dims[1] = 3;
d_dims[2] = 4;
sdsid = SDcreate(f1, "DataSetChunked_4", DFNT_UINT16, 3, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS - chunk is 1x1x4 which will create 6 chunks. */
c_def.chunk_lengths[0] = 1;
c_def.chunk_lengths[1] = 1;
c_def.chunk_lengths[2] = 4;
status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 4 chunks. */
status = SDsetchunkcache(sdsid, 4, 0);

/* Write data using SDwritedata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDwritedata(sdsid, start_dims, NULL, edge_dims, \
                       (VOIDP) u16_3data);

/* Read data using SDreadchunk and verify against
       the chunk arrays chunk1_3u16[] ... chunk6_3u16[]. */

/* Read chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);
```

```
/* Read chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 3. */
start_dims[0] = 0;
start_dims[1] = 2;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 4. */
start_dims[0] = 1;
start_dims[1] = 0;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 5. */
start_dims[0] = 1;
start_dims[1] = 1;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 6. */
start_dims[0] = 1;
start_dims[1] = 2;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);

}
```

EXAMPLE 16.

Writing and Reading Compressed Chunked Data Using SDwritechunk and SDreaddata

This example uses **SDwritechunk** to write the chunks and **SDreaddata** to read the subset, like Example 13 (See page 62.). However, it also compresses the chunks using the GZIP (skipping Huffman) algorithm.

C:

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

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  chunk1_2u16[6] = {11, 21,
```

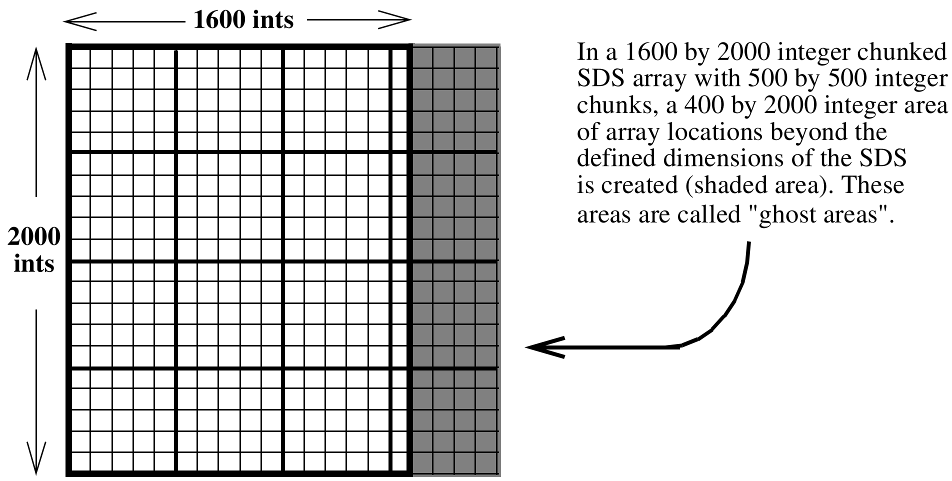
```
                                 12, 22,
                                 13, 23};

static uint16  chunk2_2u16[6] = {31, 41,
                                 32, 42,
                                 33, 43};

static uint16  chunk3_2u16[6] = {14, 24,
                                 15, 25,
                                 16, 26};

static uint16  chunk4_2u16[6] = {34, 44,
                                 35, 45,
                                 36, 46};

static uint16  chunk5_2u16[6] = {17, 27,
                                 18, 28,
                                 19, 29};

static uint16  chunk6_2u16[6] = {37, 47,
                                 38, 48,
                                 39, 49};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_2u16[5][2];    /* Data array read */
uint16  fill_u16 = 0;        /* Fill value */
int32 c_flags;
HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example16.hdf", DFACC_CREATE);

/* Create a 9x4 SDS of uint16 in file 1. */
d_dims[0] = 9;
d_dims[1] = 4;
sdsid = SDcreate(f1, "DataSetChunked_1", DFNT_UINT16, 2, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS chunk with 3x2 chunks which will create
    6 chunks. */
c_def.chunk_lengths[0] = c_def.comp.chunk_lengths[0] = 3;
c_def.chunk_lengths[1] = c_def.comp.chunk_lengths[1] = 2;

c_def.comp.comp_type = COMP_CODE_DEFLATE; /* GZIP */
c_def.comp.cinfo.deflate.level = 6;

status = SDsetchunk(sdsid, c_def, HDF_CHUNK | HDF_COMP);

/* Set chunk cache to hold a maximum of 3 chunks */
status = SDsetchunkcache(sdsid, 3, 0);

/* Write the data chunks. */

/* Write chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
```

```
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk1_2u16);

/* Write chunk 4. */
start_dims[0] = 1;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk4_2u16);

/* Write chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk2_2u16);

/* Write chunk 5. */
start_dims[0] = 2;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk5_2u16);

/* Write chunk 3. */
start_dims[0] = 1;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk3_2u16);

/* Write chunk 6. */
start_dims[0] = 2;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk6_2u16);

/* Read a subset of the data back in using SDreaddata
       i.e  5x2 subset of the whole array. */
start_dims[0] = 2;
start_dims[1] = 1;
edge_dims[0] = 5;
edge_dims[1] = 2;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_2u16);

/* This 5x2 array should look like this
         {{23, 24, 25, 26, 27},
          {33, 34, 35, 36, 37}} */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);

}
```

3.8.6 Ghost Areas

In cases where the size of the SDS array is not an even multiple of the chunk size, regions of excess array space beyond the defined dimensions of the SDS will be created. Refer to the following illustration.

FIGURE 3c **Array Locations Created Beyond the Defined Dimensions of an SDS**



These regions are called "ghost areas". "Ghost areas" can be accessed only by **SDreadchunk** and **SDwritechunk** - they cannot be written to or accessed by either **SDreaddata** or **SDwritedata**. If the fill value has been set, the values in these array locations will be initialized to the fill value. It is highly recommended that users set the fill value before writing to chunked SDSs so that garbage values won't be read from these locations.

Also, if **SDreadchunk** and **SDwritechunk** are used, it is not recommended that valid data be written to the "ghost areas" as these areas won't be accessible by **SDreaddata** and **SDwritedata**. Moreover, the ability to write to these areas may not be supported in future versions of the HDF library.

3.9 SD Dimension and Dimension Scale Operations

A description of dimensions is in Section 3.2.1.4 on page 20.

3.9.1 Selecting a Dimension: SDgetdimid

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

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

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

SDgetdimid has two arguments; `sds_id` and `dim_index`, and returns a dimension identifier, `dim_id`. The argument `dim_index` is a zero-based integer indicating the location of the dimension in the data set.

The number of dimensions in a data set is specified at the time the data set is created. Specifying a dimension index larger than the number of dimensions in the data set returns an error.

Unlike file and data set identifiers, dimension identifiers do not require explicit disposal.

3.9.2 Naming a Dimension: SDsetdimname

SDsetdimname assigns a name to the selected dimension. The name of the dimension will also appear as the name of the dimension scale. If the dimension name is not unique, it is assumed that both dimensions refer to the same object and changes to one will be reflected in the other. Naming dimensions is optional but encouraged. Dimensions that are not explicitly named by the user will have names generated by the HDF library. Use **SDdiminfo** to read existing dimension names.

The following steps are required to name a dimension and its scale:

1. Get the identifier of the dimension.
2. Assign a name to the dimension - the dimension scale will be set automatically.

The following routine calls are required to do this:

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

FORTRAN:
```
dim_id = sfdimid(dim_id, dim_index)
status = sfsdimname(dim_id, dim_name)
```

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

What should be remembered when naming dimensions is that the name or a particular dimension *must* be set before attributes are assigned - and, once the attributes have been set, the name should not be changed. In other words, **SDsetdimname** should only be called before any calls to **SDsetdimscale** (described in Section 3.9.4.1 on page 74), **SDsetattr** (described in Section 3.10.1 on page 81) or **SDsetdimstrs** (described in Section 3.11.2.1 on page 89).

TABLE 3R **SDsetdimname Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetdimname **(sfsdimname)**	dim_id	int32	integer	Dimension identifier.
	dim_name	char *	character* (*)	Dimension name.

3.9.3 Old and New Dimension Implementations

Up to and including HDF version 4.0 beta1, dimensions were vgroup objects containing a single field vdata with a class name of "DimVal0.0". The vdata had the same number of records as the size of the dimension, which consisted of the values 0, 1, 2, . . . n - 1, where n is the size of the dimension. These values weren't strictly necessary and for applications that create large one dimensional array datasets the disk space taken by these unnecessary values would nearly double the size of the HDF file. In order to avoid these situations, a new representation of dimensions was implemented for HDF version 4.0 beta 2 and later versions.

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

Until HDF version 4.1, the old and new dimension representation will be written by default for each dimension created, and both representations will be recognized by routines that operate on dimensions. HDF version 4.1 routines will only recognize the new representation. During the transitional period, two routines will be provided to allow HDF programs to distinguish between the two dimension representations, or *compatibility modes*: - **SDsetdimval_comp** and **SDsetdimval_bwcomp**.

3.9.3.1 Setting the Future Compatibility Mode of a Dimension: SDsetdimval_comp

SDsetdimval_comp determines whether the specified dimension *will have* the old and new representations or the new representation only, by setting the compatibility mode for the specified dimension. The routine's syntax is the following:

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

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

The `comp_mode` parameter determines the compatibility mode. It can be set to either `SD_DIMVAL_BW_COMP`, which specifies compatible mode and that the old and new dimension representations will be written to file , or `SD_DIMVAL_BW_INCOMP`, which specifies incompatible mode and that only the new dimension representation will be written to file.

Unlimited dimensions are always backward compatible. Therefore, **SDsetdimval_comp** takes no action on these dimensions.

SDsetdimval_comp returns either `SUCCEED` on successful completion, or `FAIL` otherwise.

TABLE 3S **SDsetdimval_comp Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetdimval_comp (sfsdmvc)	dim_id	int32	integer	Dimension identifier.
	comp_mode	intn	integer	Compatibility mode.

3.9.3.2 Setting the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp

SDisdimval_bwcomp determines whether the specified dimension *has* the old and new representations or the new representation only. The function's syntax is the following:

C: `comp_mode = SDisdimval_bwcomp(dim_id);`

FORTRAN: `comp_mode = sfisdmvc(dim_id)`

SDisdimval_bwcomp returns one of three values: either `SD_DIMVAL_BW_COMP`, `SD_DIMVAL_BW_INCOMP` or `ERROR`. `SD_DIMVAL_BW_COMP` and `SD_DIMVAL_BW_INCOMP` are interpreted as they are by **SDisdimval_comp**.

TABLE 3T **SDisdimval_bwcomp Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDisdimval_bwcomp (sfisdmvc)	dim_id	int32	integer	Dimension identifier.

3.9.4 Dimension Scales

A dimension scale is a series of numbers placed along a dimension to demarcate intervals in a data set. One scale is assigned per dimension. In the SDS data model, each dimension scale is a one-dimensional array with size and name equal to its assigned dimension name and size. For example, if a dimension of length 6 named "depth" is assigned a dimension scale, its scale is a one-dimensional array of length 6 and is also assigned the name "depth".

Although dimension scales are conceptually different from SDS arrays, they are both arrays and are often treated in the same way by the SDS API. For example, when the **SDfileinfo** routine returns the number of data sets in a file, it includes dimension scales in that number. The **SDiscoordvar** routine (described in Section 3.9.5 on page 78) distinguishes data sets from dimension scales.

3.9.4.1 Writing Dimension Scales: SDsetdimscale

Selecting a dimension with **SDgetdimid** assigns a default dimension scale. Default dimension scales have the data type of the corresponding SDS array. To assign a non-default scale with a non-default data type, use **SDsetdimscale**.

To create a non-default scale, the following steps are required:

1. Get the identifier of the dimension.
2. Create the dimension scale, setting the data type and size to the desired values.

To do this, the calling program must execute the following calls:

C:
```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDsetdimscale(dim_id, count, data_type, data);
```

FORTRAN:
```
dim_id = sfdimid(sds_id, dim_index)
status = sfsdscale(dim_id, count, data_type, data)
```

The argument `count` is the size of the scale, `data_type` defines the data type for the scale values and `data` is an array containing the scale values. Assigning a value to `count` is optional: it exists to insure backward compatibility.

3.9.4.2 Obtaining Dimension Scale and Other Dimension Information: SDdiminfo

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

To obtain dimension information the following steps are required:

1. Get the identifier of the dimension.
2. Retrieve the dimension information.

The calling program must call the following routines in order:

C:

```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDdiminfo(dim_id, name, count, data_type, nattrs);
```

FORTRAN:

```
dim_id = sfdimid(sds_id, dim_index)
gstatus = sfgdinfo(dim_id, name, count, data_type, nattrs)
```

In **SDdiminfo** the arguments `name`, `count`, `data_type` and `nattrs` define buffers allocated to respectively hold the dimension name and size, the data type for the scale values and the number of attributes assigned to the dimension.

Dimensions are always named. However, if you don't wish to explicitly provide a name, `NULL` can be passed as the `name` parameter to **SDdiminfo** and a default name will be assigned by the SD API. If scale information is available for the dimension, `data_type` will contain the data type of the scale values, otherwise `data_type` will be `0`.

3.9.4.3 Reading Dimension Scales: SDgetdimscale

To read a scale, the following steps are required:

1. Get the identifier of the dimension.
2. Read the scale.

The calling program must contain the following sequence of calls:

C:

```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDgetdimscale(dim_id, data);
```

FORTRAN:

```
dim_id = sfdimid(sds_id, dim_index)
status = sfgdscale(dim_id, data)
```

In **SDgetdimscale** the argument `data` is the buffer allocated to hold the scale values. As **SDgetdimscale** returns all of the values associated with a given scale, it is assumed that the size of the scale buffer is greater than or equal to the dimension size.

TABLE 3U **SDgetdimid, SDsetdimname, SDsetdimscale, SDdiminfo and SDgetdimscale Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDgetdimid (sfdimid)	sds_id	int32	integer	Data set identifier.
	dim_index	intn	integer	Index of the dimension.
SDsetdimname (sfsdmname)	dim_id	int32	integer	Dimension identifier.
	dim_name	char *	character* (*)	Dimension name.
SDsetdimscale (sfsdscale)	dim_id	int32	integer	Dimension identifier.
	count	int32 *	integer	Number of scale values.
	data_type	int32 *	integer	Data type of the scale values.
	data	VOIDP	<valid numeric data type>	Buffer for the scale values.
SDdiminfo (sfgdinfo)	dim_id	int32	integer	Dimension identifier.
	name	char *	character* (*)	Buffer for the dimension name.
	count	int32 *	integer	Buffer for the dimension size.
	number_type	int32 *	integer	Buffer for the scale data type.
	nattrs	int32 *	integer	Buffer for the attribute count.
SDgetdimscale (sfgdscale)	dim_id	int32	integer	Dimension identifier.
	data	VOIDP	<valid numeric data type>	Buffer for the scale values.

EXAMPLE 17.

Writing Dimension Information

The dimensions of the SDS created in Example 4 are created and scales assigned to them in these examples.

C:

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

main( )
{
int32 sd_id, sds_id, dim_index, dim_id, sds_index, status;
int32 count, num_type, num_attrs;
int16 dim_scale[] = {6,5,4,3,2,1};
char dim_name[MAX_NC_NAME];

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDWR);

/* Get the index of the "Ex_array_4" array data set. */
sds_index = SDnametoindex(sd_id, "Ex_array_4");

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

/* For each dimension of the "Ex_array_4" array data set, */
for (dim_index = 0; dim_index < 3; dim_index++) {

/* - select the dimension id, */
    dim_id = SDgetdimid(sds_id, dim_index);

/* - get the information about the selected dimension, */
    status = SDdiminfo(dim_id, dim_name, &count, &num_type, \
                       &num_attrs);
    num_type = DFNT_INT16;

/* - alter the dimension names, */
    switch(dim_index) {
        case 0:SDsetdimname(dim_id, "Z_Axis");
                break;
        case 1:SDsetdimname(dim_id, "Y_Axis");
                break;
        case 2:SDsetdimname(dim_id, "X_Axis");
                break;
        default:break;
    }

/* - then alter the dimension scale and write it to the data set. */
    dim_scale[0] = 3;
    dim_scale[1] = 2;
    dim_scale[2] = 1;
    status = SDsetdimscale(dim_id, count, num_type, (VOIDP)dim_scale);
}

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

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

}
```

FORTRAN:

```
      PROGRAM ALTER DIMENSION

       integer*4 sd_id, sds_id, dim_index, dim_id, status
       integer*4 count, num_attrs
       integer sfstart, sfn2index, sfdimid, sfgdinfo
       integer sfsdscale, sfsdmname, sfendacc
       integer sfend, sfselect, num_type, i, dim_scale(6)
       integer DFACC_RDWR, MAX_NC_NAME, DFNT_INT16
       parameter (DFACC_RDWR = 3, MAX_NC_NAME = 256, DFNT_INT16 = 22)
       character dim_name(MAX_NC_NAME)

C      For each dimension of the 'Ex_array_4' array data set,
       do 5 i = 1, 6
          dim_scale(i) = i
5      continue

C      Open the file.
       sd_id = sfstart('Example4.hdf', DFACC_RDWR)

C      Get the index of the 'Ex_array_4' array data set.
       sds_index = sfn2index(sd_id, 'Ex_array_4')

C      Select the data set corresponding to the returned index.
       sds_id = sfselect(sd_id, sds_index)

C      For each dimension of the 'Ex_array_4' array data set,
       do 10 dim_index = 1, 3

C       - select the dimension id,
          dim_id = sfdimid(sds_id, dim_index-1)

C       - get the information about the selected dimension,
          status = sfgdinfo(dim_id, dim_name, count, num_type,
     +                      num_attrs)

C       - alter the dimension names.
          if (dim_index .eq. 1) then
          status = sfsdmname(dim_id, 'Z_Axis')
          end if
          if (dim_index .eq. 2) then
          status = sfsdmname(dim_id, 'Y_Axis')
          end if
          if (dim_index .eq. 3) then
          status = sfsdmname(dim_id, 'X_Axis')
          end if

          num_type = DFNT_INT16

C      - and, alter the dimension scale, write it to the data set,
          dim_scale(1) = 3
          dim_scale(2) = 2
          dim_scale(3) = 1
          dim_scale(4) = 0
          dim_scale(5) = -1
          dim_scale(6) = -2
          status = sfsdscale(dim_id, count, num_type, dim_scale)
10     continue

C      Terminate access to the array.
       status = sfendacc(sds_id)
```

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

3.9.5 Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar

Although dimension scales, or ***coordinate variables*** in netCDF, can for the most part be ignored, it is important to note that, in HDF, they are also SDSs and are assigned the same tag (`DFTAG_SD`) as other SDSs. As a result, dimension scales are treated as SDSs, and are included in the SDS count returned by **SDfileinfo**. The function **SDiscoordvar** is available to determine whether or not a given SDS is a dimension scale.

SDiscoordvar takes an SDS id as its only argument and returns `TRUE` if the data set is a dimension scale, and `FALSE` if it isn't. If **SDiscoordvar** returns `TRUE`, a subsequent call to **SDgetinfo** will fill the specified locations with information about the dimension scale, rather than the data set array.

TABLE 3V

SDiscoordvar Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDiscoordvar (sfiscvar)	sds_id	int32	integer	Data set identifier.

EXAMPLE 18.

Retrieving SDS Information from an HDF File

SDgetinfo and **SDiscoordvar** provide information that may be needed in order to read in an array successfully, as the following examples show.

C:

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

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id, status;
int32 rank, nt, dims[MAX_VAR_DIMS], nattrs;
int32 start[2], edges[2];
int16 array_data[Y_LENGTH][X_LENGTH];
char name[MAX_NC_NAME];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_RDONLY);

/* Select the first (and in this case, only) data set in the file. */
sds_id = SDselect(sd_id, 0);

/* Confirm that the data set is not a coordinate variable. */
if (FALSE == SDiscoordvar(sds_id)) {

/* Verify the characteristics of the array. */
    status = SDgetinfo(sds_id, name, &rank, dims, &nt, &nattrs);
```

```
/* Define the location, pattern, and size of the data to read from \
    the data set. */
    start[0] = start[1] = 0;
    edges[0] = dims[0];
    edges[1] = dims[1];

/* Read the array data set created in Example 3. */
    status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

}

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

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

}
```

FORTRAN:

```
      PROGRAM CONFIRM FILE

       integer*4 sd_id, sds_id, rank, nt, nattrs, status
       integer start(2), edge(2), stride(2)
       integer datavar
       integer sfstart, sfselect, sfiscvar, sfginfo
       integer sfrdata, sfendacc, sfend

C      DFACC_RDONLY is defined in hdf.h. MAX_NC_NAME and MAX_VAR_DIMS
C      are defined in netcdf.h.
       integer*4 DFACC_RDONLY, MAX_NC_NAME, MAX_VAR_DIMS
       integer*4 X_LENGTH, Y_LENGTH
       parameter (DFACC_RDONLY = 1, MAX_NC_NAME = 256,
      +           MAX_VAR_DIMS = 32, X_LENGTH = 4, Y_LENGTH = 15)
       integer*2 array_data(X_LENGTH, Y_LENGTH)
       character name(MAX_NC_NAME)
       integer dims(MAX_VAR_DIMS)

C      Open the file and initiate the SD interface.
       sd_id = sfstart('Example3.hdf', DFACC_RDONLY)

C      Select the first (and in this case, only) data set in the file.
       sds_id = sfselect(sd_id, 0)

C      Confirm that the data set is not a coordinate variable.
       datavar = sfiscvar(sds_id)
       if (datavar .eq. 0) then

C         Verify the characteristics of the array.
          status = sfginfo(sds_id, name, rank, dims, nt, nattrs)

C         Define the location, pattern, and size of the data to read
C         from the data set.
          start(1) = 0
          start(2) = 0
          stride(1) = 1
          stride(2) = 1
          edge(1) = dims(1)
          edge(2) = dims(2)

C         Read the array data set.
```

```
          status = sfrdata(sds_id, start, stride, edge, array_data)
       endif

C      Terminate access to the array data set.
       status = sfendacc(sds_id)

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

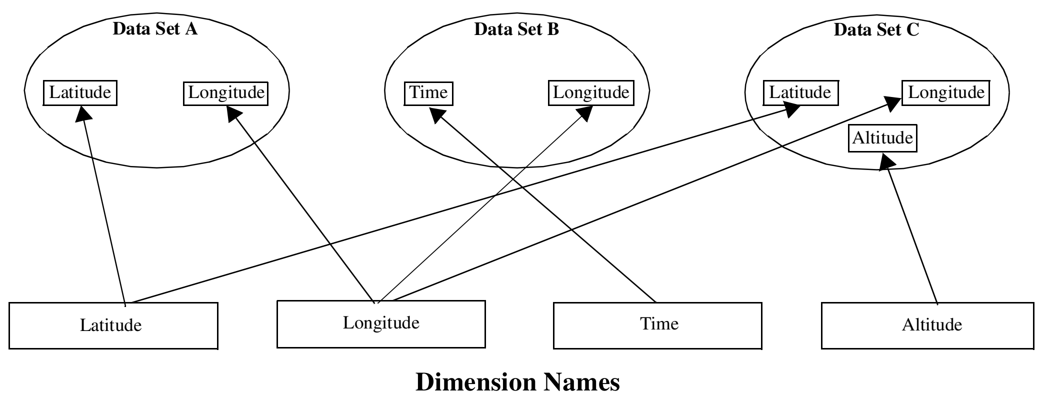
3.9.6 Dimension Scales for Multiple Data Sets

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

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

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

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



3.10 User-defined Attributes

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

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

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

For each object, a separate ***attribute count*** is maintained that identifies the number of attributes associated with the object. The attribute count begins at zero and is increased by one for every new attribute assigned to an object. Each attribute associated with an object has a unique ***attribute index***, a value between 1 and the total number of attributes. The attribute index is used to retrieve an attribute's value or information about an attribute.

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

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

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

3.10.1 Writing User-defined Attributes: SDsetattr

Attributes are not actually written out to a file until access to the object is terminated. Creating an attribute increases the attribute count by one for the given object. Writing an attribute involves the following steps:

1. Obtain the appropriate identifiers:
2. Create the attribute.
3. Terminate access by disposing of any existing identifiers:

To assign an attribute to a file, the calling program must contain a call to **SDsetattr**:

C:
```
status = SDsetattr(sd_id, attr_name, data_type, count, value);
```

FORTRAN:
```
status = sfsnatt(sd_id, attr_name, data_type, count, value)
```

In **SDsetattr** the argument `sd_id` is the identifier for the HDF object to be assigned the attribute and it can be a file id, SDS id or dimension id. The argument `attr_name` is an ASCII string containing the name of the attribute. It represents the label in the `label=value` equation and can be no more than `MAX_NC_NAME` characters. If this is set to the name of an existing attribute the value portion of the attribute will be overwritten. Do not use **SDsetattr** to assign a name to a dimension, use **SDsetdimname** instead.

The arguments, `data_type`, `count` and `value` describe the right side of the `label=value` equation. The `value` argument contain one or more values of the same data type. The `data_type` argument describes the data type for the attribute values and `count` defines the total number of values in the attribute.

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.

The parameters of **SDsetattr** are further described below. (See Table 3W on page 84.) Note that, because there are two Fortran-77 versions of **SDsetattr**, there are correspondingly two entries in the "Data Type" field of the *values* parameter.

EXAMPLE 19.

Setting Attribute Values

The following examples call **SDsetattr** with an SDS id as the first parameter, which assigns an SDS attribute to the selected data set. If a file id were passed instead, **SDsetattr** would assign a file attribute to the entire file.

C:

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

main( )
{

int32 sd_id, sds_id, dim_id, dim_index, status;
int32 num_values[2];

/* Open the file and get the identifier for the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDWR);

/* Set an attribute that describes the file contents. */
status = SDsetattr(sd_id, "file_contents", DFNT_CHAR8, 16, \
                (VOIDP)"storm_track_data");

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

/* Set an attribute the specifies a valid range of values. */
num_values[0] = 2;
num_values[1] = 10;
status = SDsetattr(sds_id, "valid_range", DFNT_INT32, 2, \
                (VOIDP)num_values);

/* Get the identifier for the first dimension. */
dim_id = SDgetdimid(sds_id, 0);

/* Set an attribute that specifies the dimension metric. */
status = SDsetattr(dim_id, "dim_metric", DFNT_CHAR8, 9,
                (VOIDP)"millibars");
```

```
/* Terminate access to the array */
status = SDendaccess(sds_id);

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

}
```

FORTRAN:

```
      PROGRAM SET ATTRIBS

       integer*4 sd_id, sds_id, dim_id, status
       integer num_values(2)
       integer sfstart, sfsnatt, sfselect, sfdimid
       integer sfendacc, sfend
       integer DFACC_RDWR, DFNT_CHAR8, DFNT_INT32
       parameter (DFACC_RDWR = 3, DFNT_CHAR8 = 4, DFNT_INT32 = 24)

C      Open the file and get the identifier for the file.
       sd_id = sfstart('Example4.hdf', DFACC_RDWR)

C      Set an attribute that describes the file contents.
       status = sfsnatt(sd_id, 'file_contents', DFNT_CHAR8, 16,
      +                 'storm_track_data')
C      Get the identifier for the first data set.
       sds_id = sfselect(sd_id, 0)

C      Set an attribute the specifies a valid range of values.
       num_values(1) = 2
       num_values(2) = 10
       status = sfsnatt(sds_id, 'valid_range', DFNT_INT32, 2,
      +  num_values)

C      Get the identifier for the first dimension.
       dim_id = sfdimid(sds_id, 0)

C      Set an attribute that specifies the dimension metric.
       status = sfsnatt(dim_id, 'dim_metric', DFNT_CHAR8, 9,
      +  'millibars')

C      Terminate access to the array
       status = sfendacc(sds_id)

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

3.10.2 Querying User-defined Attributes: SDfindattr and SDattrinfo

Given a file, array or dimension id and an attribute name, **SDfindattr** will return a valid attribute index if the corresponding attribute exists. The attribute index can then be used to retrieve information about an attribute or its values. Given a file, array or dimension id and a valid attribute index **SDattrinfo** returns the name, data type and count for the corresponding attribute if it exists.

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

C:

```
attr_index = SDfindattr(id, attr_name);
status = SDattrinfo(id, attr_index, attr_name, num_type, count);
```

```
FORTRAN: attr_index = sffattr(id, attr_name)
         status = sfgainfo(id, attr_index, attr, name, number_type,
                           count)
```

The parameters of **SDfindattr** and **SDattrinfo** are further described below. (See Table 3W.)

An attribute's index may also be determined by keeping track of the number and order of attributes as they are written or dumping the contents of a file using a dumping utility.

3.10.3 Reading User-defined Attributes: SDreadattr

SDreadattr reads the value or values of an attribute. The syntax for **SDreadattr** is as follows:

```
C:       status = SDreadattr(sds_id, attr_index, data);

FORTRAN: status = sfrattr(sd_id, attr_index, data)
```

SDreadattr takes a file, array, or dimension identifier and an attribute index specified in the `attr_index` parameter as input parameters and returns the attribute values in the buffer `data`. **SDreadaddr** will also read attributes and annotations created by the DFSD interface.

It's assumed that the buffer `data`, allocated to hold the attribute values, is large enough to hold the data. The size of the buffer must be at least `count*DFKNTsize(number_type)` bytes long. It is not possible to read a subset of values.

There are two Fortran-77 versions of this routine: **sfrnatt** and **sfrcatt**. The **sfrnatt** routine reads numeric attribute data and **sfrcatt** reads character attribute dataset.

The parameters of **SDreadattr** are further described in Table 3W. Note that, because there are two Fortran-77 versions of **SDreadattr**, there are correspondingly two entries in the "Data Type" field of the *data* parameter.

TABLE 3W **SDsetattr, SDfindattr, SDattrinfo and SDreadattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetattr (sfsnatt/ sfscatt)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_name	char *	character* (*)	Attribute name.
	data_type	int32	integer	Data type of the attribute.
	count	int32	integer	Number of values in the attribute.
	values	VOIDP	<valid numeric data type>	Buffer for the data to be written.
SDfindattr (sffattr)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_name	char *	character* (*)	Attribute name.

SDattrinfo (sfgainfo)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_index	int32	integer	Index of the attribute to be read.
	attr_name	char *	character* (*)	Buffer for the name of the dimension attribute.
	data_type	int32 *	integer	Buffer for the data type of the values in the attribute.
	count	int32 *	integer	Buffer for the total number of values in the attribute.
SDreadattr (sfrnatt/ sfrcatt)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_index	int32	integer	Index for the attribute to be read.
	data	VOIDP	<valid numeric data type>	Buffer for the attribute values.

EXAMPLE 20.

Retrieving Attribute Information

The attribute information stored in the "Example4.hdf" HDF file in Example 15 are read from the file in these examples.

C:

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

main( )
{

int32 sd_id, sds_id, status, *buffer1;
int32 attr_index, data_type, count;
char attr_name[MAX_NC_NAME];
int8 *buffer;

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Find the file attribute named "file_contents". */
attr_index = SDfindattr(sd_id, "file_contents");

/* Get information about the file attribute. */
status = SDattrinfo(sd_id, attr_index, attr_name, &data_type, &count);

/* Allocate a buffer to hold the attribute data. */
buffer = (int8 *)malloc(count * DFKNTsize(data_type));

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

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

/* Find the data set attribute named "valid_range". */
attr_index = SDfindattr(sds_id, "valid_range");

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

/* Allocate a buffer to hold the attribute data. */
buffer1 = (int32 *)malloc(count * DFKNTsize(data_type));
```

```
/* Read the attribute data. */
status = SDreadattr(sd_id, attr_index, buffer1);

/* Terminate access to the array */
status = SDendaccess(sds_id);

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

}
```

FORTRAN:

```
     PROGRAM ATTRIB INFO

      integer*4 sd_id, sds_id, range_buffer(2)
      integer attr_index, data_type, count, status
      character attr_name *13
      character char_buffer *20
      integer sfstart, sffattr, sfgainfo, sfrattr, sfselect
      integer sfendacc, sfend

C     DFACC_RDWR is defined in hdf.h.
      integer DFACC_RDWR
      parameter (DFACC_RDWR = 3)

C     Open the file.
      sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C     Find the file attribute named 'file_contents'.
      attr_index = sffattr(sd_id, 'file_contents')

C     Get information about the file attribute.
      status = sfgainfo(sd_id, attr_index, attr_name, data_type,
     +  count)

C     Read the attribute data.
      status = sfrattr(sd_id, attr_index, char_buffer)

C     Get the identifier for the first data set.
      sds_id = sfselect(sd_id, 0)

C     Find the data set attribute named 'valid_range'.
      attr_index = sffattr(sds_id, 'valid_range')

C     Get information about the data set attribute.
      status = sfgainfo(sds_id, attr_index, attr_name, data_type,
     +  count)

C     Read the attribute data.
      status = sfrattr(sd_id, attr_index, range_buffer)

C     Terminate access to the array
      status = sfendacc(sds_id)

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

      end
```

3.11 Predefined Attributes

Predefined attributes are attributes that use reserved labels and in some cases Predefined data types. Predefined attributes are categorized as follows:

- ***Labels*** can be thought of as independent variable names and dimension names and as such are used as primary search keys.
- ***Units*** are a means of declaring the units pertinent to a specific discipline. Unidata has developed a freely-available library of routines to convert between character string and binary forms of unit specifications and to perform useful operations on the binary forms. This library is used in some netCDF applications and is recommended for use with HDF applications. For more information, refer to the *netCDF User's Guide*.
- ***Formats*** describe the form numeric values will be printed and/or displayed. The recommended convention is to use standard Fortran-77 notation for describing the data format. For example, "F7.2" means to display seven digits with two digits to the right of the decimal point.
- ***Coordinate systems*** contain information that should be used when interpreting or displaying the data. For example, the text strings "cartesian", "polar" and "spherical" are recommended coordinate system descriptions.
- ***Ranges*** define the maximum and minimum values of a selected valid range. The range may cover the entire data set, values outside the data set or a subset of values within a data set. Because the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate as long as they are of the same data type as the SDS array.
- A ***fill value*** is the value used to fill the areas between non-contiguous writes to SDS arrays. For more information about fill values, refer to Section 3.11.5 on page 91.
- ***Calibration*** stores scale and offset values used to create calibrated data in SDS arrays. When data are calibrated, they are typically reduced from floats, doubles or large integers into 8-bit or 16-bit integers and "packed" into an appropriately sized array. After the scale and offset values are applied, the packed array will return to its original form.

Predefined attributes are useful because they establish conventions that applications can depend on and because they are understood by the HDF library without users having to define them. Predefined attributes also insure backward compatibility with earlier versions of the HDF library. They can be assigned to two types of HDF objects: data sets and dimensions. (See Table 3X.)

TABLE 3X **Predefined Attribute List**

Object Type	Attribute	Reserved Label	Description
SDS Array or Dimension	Label	long_name	Name of the array.
	Unit	units	Units used for all dimensions and data.
	Format	format	Format for displaying dim scales and array values.
SDS Array Only	Coordinate System	cordsys	Coordinate system used to interpret the SDS array.
	Range	valid_range	Maximum and minimum values within a selected data range.
	Fill Value	_FillValue	Value used to fill empty locations in SDS array.
	Calibration	scale_factor	Value by which each array value is to be multiplied.
		scale_factor_err	Error introduced by scaling SDS array data.
		add_offset	Value to which each array value is to be added.
		add_offset_err	Error introduced by offsetting the SDS array data.
		calibrated_nt	Data type of the calibrated data.

The following naming conventions taken from the *netCDF User's Guide* are meant to promote consistency of information-sharing among generic applications. These naming conventions are not specifically required by the HDF library, but are highly recommended.

- **missing_value**: An attribute containing a value use to fill areas of an array not intended to contain valid data or a fill value. The scope of this attribute is local to the array. An example of this would be a region where information is unavailable, as in a geographical grid containing ocean data. The part of the grid where there is land might not have any data associated with it and in such a case the `missing_value` value could be supplied. The `missing_value` attribute is different from the `_FillValue` attribute in that fill values are intended to indicate data that was expected but did not appear, whereas missing values are used to indicate data that were never expected.
- **title**: A global file attribute containing a description of the contents of a file.
- **history**: A global file attribute containing the name of a program and the arguments used to derive the file. Well-behaved generic filters (programs that take HDF or netCDF files as input and produce HDF or netCDF files as output) would be expected to automatically append their name and the parameters with which they were invoked to the history attribute of an input file.

3.11.1 Accessing Predefined Attributes

The SD interface provides two methods for accessing predefined attributes. The first method uses the general attribute routines for user-defined attributes described in Section 3.11 on page 87 and the second employs routines specifically designed for each attribute. Although the general attribute routines work well and are recommended in most cases, the specialized attribute routines are sometimes easier to use, especially when reading or writing related predefined attributes. This is true for two reasons. First, because predefined attributes are guaranteed unique names, the attribute index is unnecessary. Second, attributes with several components may be read as a group. For example, using the SD routine designed to read the predefined calibration attribute returns all five components with a single call, rather than five separate calls.

There is one exception: unlike predefined array attributes, predefined dimension attributes should be read or written using the specialized attribute routines only.

The predefined attribute parameters are described in Table 3Y. Creating a predefined attribute with parameters different from these will produce unpredictable results when the attribute is read using the corresponding predefined-attribute routine.

TABLE 3Y **Predefined Attribute Parameter List**

Category	attr_name (Attribute type)	num_type (Data type)	count (Number of Values)	value (Attribute value)
Label	long_name	DFNT_CHAR8	string length	Pointer to string.
Unit	units	DFNT_CHAR8	string length	Pointer to string.
Format	format	DFNT_CHAR8	string length	Pointer to string.
Coordinate System	cordsys	DFNT_CHAR8	string length	Pointer to string.
Range	valid_range	<valid type>	2	Pointer to array.
Fill Value	_FillValue	<valid type>	1	Pointer to fill value.

Calibration	scale_factor	DFNT_FLOAT64	1	Pointer to scale.
	scale_factor_err	DFNT_FLOAT64	1	Pointer to scale error.
	add_offset	DFNT_FLOAT64	1	Pointer to offset.
	add_offset_err	DFNT_FLOAT64	1	Pointer to offset error.
	calibrated_nt	DFNT_INT32	1	Pointer to data type.

In addition to **SDreadattr**, **SDfindattr** and **SDattrinfo** are also valid general attribute routines to use when reading a predefined attribute. **SDattrinfo** is always useful for determining the size of an attribute whose value contains a string.

3.11.2 SDS String Attributes

Predefined string attributes for an SDS array include the ***label***, ***unit***, ***format*** and ***coordinate system*** of an SDS.

3.11.2.1 Writing String Attributes: SDsetdatastrs

The following function assigns label, unit, format and coordinate system string attributes to an SDS array:

C:
```
status = SDsetdatastrs(sds_id, label, unit, format, coordsys);
```

FORTRAN:
```
status = sfsdtstr(sds_id, label, unit, format, coordsys)
```

SDsetdatastrs assigns a predefined attribute to an SDS. The arguments for **SDsetdatastrs** are described in Table 3Z. To avoid creating one or more attributes, pass `NULL` as the appropriate argument.

3.11.2.2 Reading String Attributes: SDgetdatastrs

The following function reads the label, unit, format and coordinate system string attributes of an SDS:

C:
```
status = SDgetdatastrs(sds_id, label, unit, format, coordsys,
                       len);
```

FORTRAN:
```
status = sfgdtstr(sds_id, label, unit, format, coordsys, len)
```

SDgetdatastrs reads the predefined attributes of an SDS array. The arguments `label`, `unit`, `format` and `coordsys` are string buffers. If a particular attribute does not exist, the first character of the returned string will be `NULL`. Each string buffer is assumed to be at least `len` characters long, including the space to hold the `NULL` termination character. To avoid reading a particular attribute, pass `NULL` in the corresponding argument.

Keep in mind that the value returned by `label` is the value of the attribute named "long_name" and that the value returned by `coordsys` is the value of the attribute named "cordsys". The reasons for this are explained in User's Guide Appendix F.

The parameters of **SDgetdatastrs** are described in Table 3Z.

TABLE 3Z **SDsetdatastrs and SDgetdatastrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

SDsetdatastrs (sfsdtstr)	sds_id	int32	integer	Data set identifier.
	label	char *	character* (*)	Label for the data.
	unit	char *	character* (*)	Definition of the units.
	format	char *	character* (*)	Description of the data format.
	coordsys	char *	character* (*)	Description of the coordinate system.
SDgetdatastrs (sfgdtstr)	sds_id	int32	integer	Data set identifier.
	label	char *	character* (*)	Buffer for the label.
	unit	char *	character* (*)	Buffer for the description of the units.
	format	char *	character* (*)	Buffer for the description of the data format.
	coordsys	char *	character* (*)	Buffer for the description of the coordinate system.
	len	integer	intn	Maximum length of the attributes.

3.11.3 Dimension String Attributes

Dimension string attributes include a ***label***, ***unit*** and ***format*** string which describe a dimension. They adhere to the same definitions as those of the label, unit and format strings for SDS attributes.

3.11.3.1 Writing a Dimension String Attribute: SDsetdimstrs

C: `status = SDsetdimstrs(dim_id, label, unit, format);`

FORTRAN: `status = sfsdmstr(dim_id, label, unit, format)`

SDsetdimstrs assigns label, unit and format attributes to an SDS dimension. The `dim_id` argument is the dimension identifier returned by the call to **SDgetdimid**. It identifies the dimension to which the attribute will be assigned. The remaining arguments are described below. (See Table 3AA on page 90.)

3.11.3.2 Reading a Dimension String Attribute: SDgetdimstrs

C: `status = SDgetdimstrs(dim_id, label, unit, format, len);`

FORTRAN: `status = sfgdmstr(dim_id, label, unit, format, len)`

SDgetdimstrs reads the attributes specified by the `label`, `unit` and `format` parameters to an SDS dimension. The arguments `label`, `unit`, and `format` are buffers to hold the label, unit and format strings as defined in their respective attributes. If a particular attribute does not exist, the first character of the returned string will be `NULL`. Each buffer is assumed to be at least `len` characters long including the space to hold the `NULL` termination character. To avoid reading a particular attribute, pass `NULL` as the appropriate argument.

The parameters of **SDgetdimstrs** are described in Table 3AA.

TABLE 3AA **SDsetdimstrs and SDgetdimstrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetdimstrs (sfsdmstr)	dim_id	int32	integer	Dimension identifier.
	label	char *	character* (*)	Label describing the specified dimension.
	unit	char *	character* (*)	Units to be used with the specified dimension.
	format	char *	character* (*)	Format to use when displaying the scale values.

SDgetdimstrs (sfgdmstr)	dim_id	int32	integer	Dimension identifier.
	label	char *	character* (*)	Buffer for the dimension label.
	unit	char *	character* (*)	Buffer for the dimension unit.
	format	char *	character* (*)	Buffer for the dimension format.
	len	intn	integer	Maximum length of the string attributes.

3.11.4 Range Attributes

The attribute ***range*** contains user-defined maximum and minimum values in a selected range. As the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate. Also, because the maximum and minimum values are supposed to relate to the data set, it is assumed that they are of the same data type as the data.

3.11.4.1 Writing a Range Attribute: SDsetrange

SDsetrange assigns the range attribute to an SDS:

C: `status = SDsetrange(sds_id, max, min);`

FORTRAN: `status = sfsrange(sds_id, max, min)`

The parameters of **SDsetrange** are described below. (See Table 3AB on page 91.)

3.11.4.2 Reading a Range Attribute: SDgetrange

SDgetrange reads the maximum and minimum valid values of an SDS array as specified by a **SDsetrange** call or its equivalent:

C: `status = SDgetrange(sds_id, max, min);`

FORTRAN: `status = sfgrange(sds_id, max, min)`

The arguments `max` and `min` are buffers the maximum and minimum values will be read into. The arguments in the C version of **SDgetrange** are pointers rather than simple variables, whereas in the Fortran-77 version they are variables of the same data type as the data set. The parameters of **SDgetrange** are described in the following table.

TABLE 3AB **SDsetrange and SDgetrange Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetrange (sfsrange)	sds_id	int32	integer	Data set identifier.
	max	VOIDP	<valid numeric data type>	Maximum value of the range.
	min	VOIDP	<valid numeric data type>	Minimum value of the range.
SDgetrange (sfgrange)	sds_id	int32	integer	Data set identifier.
	max	VOIDP	<valid numeric data type>	Buffer for the maximum value.
	min	VOIDP	<valid numeric data type>	Buffer for the minimum value.

3.11.5 Fill Values

A ***fill value*** is the value used to fill the spaces between non-contiguous writes to SDS arrays. If a fill value is set before writing data to an SDS, the entire array is initialized to the specified fill value. By default, any location not subsequently overwritten by SDS data will contain the fill value.

A fill value must be of the same data type as the array to which it's written. To avoid conversion errors, use data-specific fill values instead of special architecture-specific values, such as infinity and ***Not-a-Number*** or ***NaN***.

Fill values can also be defined for all SDSs within a file. This is determined by setting a ***fill mode***, which can be done by calling the **SDsetfillmode** routine described below.

3.11.5.1 Writing a Fill Value Attribute: SDsetfillvalue

SDsetfillvalue assigns a new value to the fill value attribute for an SDS array:

C: `status = SDsetfillvalue(sds_id, fill_val);`

FORTRAN: `status = sfsfill(sds_id, fill_val)`

The argument `fill_val` is the new fill value. It is recommended that you set the fill value before writing data to an SDS array, as calling **SDsetfillvalue** after data is written to an SDS array only changes the fill value attribute - it does not update the existing fill values.

There are two Fortran-77 versions of this routine: **sfsfill** and **sfscfill**. The **sfsfill** routine writes numeric fill value data and **sfscfill** writes character fill value data.

The parameters of **SDsetfillvalue** are described below. (See Table 3AC on page 93.) Note that, because there are two Fortran-77 versions of **SDsetfillvalue**, there are correspondingly two entries in the "Data Type" field of the *fill_val* parameter.

3.11.5.2 Reading a Fill Value Attribute: SDgetfillvalue

SDgetfillvalue reads in the fill value of an SDS array as specified by a **SDsetfillvalue** call or its equivalent:

C: `status = SDgetfillvalue(sds_id, fill_val);`

FORTRAN: `status = sfgfill(sds_id, fill_val)`

The argument `fill_val` is the space allocated to store the fill value.

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.

The parameters of **SDgetfillvalue** are described in the following table. Note that, because there are two Fortran-77 versions of **SDgetfillvalue**, there are correspondingly two entries in the "Data Type" field of the `fill_val` parameter.

3.11.5.3 Setting the Fill Mode for all SDSs in the Specified File: SDsetfillmode

Writing fill values to an SDS can involve more I/O overhead than is necessary. This is because, whenever a fill value is set for an SDS, two write operations are generally needed - one to write the fill value and one to write the actual dataset data. It is "generally needed" because, whenever all of the data is written to the dataset in one write operation the additional write operation to add the fill values is not performed, as it isn't necessary. For datasets containing contiguous data, preventing the HDF library from performing these fill value write operations can result in a substantial performance increase.

However, it can be tedious to unset the fill value for all SDSs in a file, particularly when the file contains a large number of them. This can be done with one call to **SDsetfillmode**, which must occur before calls to **SDsetfillvalue**.

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

C:
```
status = SDsetfillmode(file_id, fill_mode);
```

FORTRAN:
```
status = sfsflmd(file_id, fill_mode)
```

The argument `file_id` is the identifier of the file the fill mode will be applied to. The `fill_mode` argument is the fill mode to be applied - it can be set to either `SD_FILL` or `SD_NOFILL`. `SD_FILL` specifies that fill values will be written to all SDSs in the specified file by default. If **SDsetfillmode** is never called before **SDsetfillvalue**, this is the default fill mode. `SD_NOFILL` specifies that, by default, fill values will not be written to all SDSs in the specified file. This can be overridden for specific SDSs by subsequent calls to **SDsetfillvalue**.

Note that, whenever a file has been newly opened, or has been closed and then re-opened, the default `SD_FILL` fill mode will be in effect until it is changed by a call to **SDsetfillmode**.

The parameters of **SDsetfillmode** are described in the table below.

TABLE 3AC **SDsetfillvalue, SDgetfillvalue and SDsetfillmode Parameter List,**

Function Call (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetfillvalue (sfsfill/ sfscfill)	sds_id	int32	integer	Data set identifier.
	fill_val	VOIDP	<valid numeric data type>	Pointer to the fill value to be set.
SDgetfillvalue (sfgfill/ sfgcfill)	sds_id	int32	integer	Data set identifier
	fill_val	VOIDP	<valid numeric data type>	Buffer for the fill value.
SDsetfillmode (sfsflmd)	file_id	int32	integer	File identifier.
	fill_mode	intn	integer	Fill mode to be set.

3.11.6 Calibration Attributes

The ***calibration*** attribute stores scale and offset values to describe calibrated data in SDS arrays. When data are calibrated using a scale and an offset, the values in an array can be represented using a smaller data type than the original. For instance, an array containing data of type `float` could be stored as an array containing data of type 8- or 16-bit integer. Both the `scale_factor` and `add_offset` attributes should be of the type `float64`.

3.11.6.1 Writing Calibrated Data: SDsetcal

SDsetcal adds the scale factor, offset, scale factor error, offset error and the data type of the uncalibrated data to the specified data set:

C:
```
status = SDsetcal(sds_id, cal, cal_error, offset, off_err,
                  num_type);
```

FORTRAN:
```
status = sfscal(sds_id, cal, cal_error, offset, off_err,
                num_type)
```

SDsetcal must be called to calibrate the data before the data is written.

SDsetcal has six arguments; `sds_id`, `cal`, `cal_error`, `offset`, `off_err` and `num_type`. The argument `cal` represents a single value that when multiplied against every value in the calibrated array reproduces the original data (assuming an `offset` of 0). The argument `offset` represents a single value that when subtracted from every value in the calibrated array reproduces the original data (assuming a `scale` of 1). The `cal` and `offset` values relate to the original data according to the following equation:

```
original_data = cal * (calibrated_data - offset)
```

In addition to `cal` and `offset`, **SDsetcal** also includes both a scale and offset error. The argument `cal_err` contains the potential error of the calibrated data due to scaling; `off_err` contains the potential error for the calibrated data due to the offset.

The parameters of **SDsetcal** are described below. (See Table 3AD on page 94.)

3.11.6.2 Reading Calibrated Data: SDgetcal

SDgetcal reads calibration attributes for an SDS array as written by a **SDsetcal** call or its equivalent:

C:
```
status = SDgetcal(sds_id, cal, cal_error, off, off_err,
                  num_type);
```

FORTRAN:
```
status = sfgcal(sds_id, cal, cal_error, off, off_err, num_type)
```

Because the HDF library does not actually apply calibration information to the data, **SDgetcal** can be called anytime before or after the data is read. If a calibration record does not exist, **SDgetcal** returns `FALSE`. **SDgetcal** takes six arguments; `sds_id`, `cal`, `cal_error`, `offset`, `off_err` and `num_type`. These parameters are described in the following table.

TABLE 3AD

SDsetcal and SDgetcal Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetcal (sfscal)	sds_id	int32	integer	Data set identifier.
	cal	float64	real*8	Calibration factor.
	cal_error	float64	real*8	Calibration error.
	off	float64	real*8	Uncalibrated offset.
	off_err	float64	real*8	Uncalibrated offset error.
	num_type	int32	integer	Data type of uncalibrated data.
SDgetcal (sfgcal)	sds_id	int32	integer	Data set identifier.
	cal	float64 *	real*8	Pointer to the calibration factor.
	cal_error	float64 *	real*8	Pointer to the calibration error.
	off	float64 *	real*8	Pointer to the uncalibrated offset.
	off_err	float64 *	real*8	Pointer to the uncalibrated offset error.
	num_type	int32 *	integer	Pointer to the data type of uncalibrated data.

EXAMPLE 21.

Calibrating Data

Suppose the values in the calibrated array `cal_val` are the following integers:

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

By applying the calibration equation `orig = cal * (cal_val - offset)` with `cal = 0.50` and `offset = -2000.0`, the calibrated array `cal_val[]` returns to its original floating-point form:

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

3.12 netCDF

HDF supports the netCDF data model and interface developed at the Unidata Program Center (UPC). Like HDF, netCDF is an interface to a library of data access programs that store and retrieve data. The file format developed at the UPC to support netCDF uses XDR (eXternal Data Representation) a non-proprietary external data representation developed by Sun Microsystems for describing and encoding data . Full documentation on netCDF and the Unidata netCDF API is available at `http://www.unidata.ucar.edu/packages/netcdf/`.

The netCDF data model is interchangeable with the SDS data model in so far as it is possible to use the netCDF calling interface to place an SDS into an HDF file and conversely the SDS interface will read from an XDR-based netCDF file. Because the netCDF API has not changed and netCDF files stored in XDR format are readable, existing netCDF programs and data are still usable, although programs will need to be relinked to the new library. However, there are important conceptual differences between the HDF and the netCDF data model that must be understood to effectively use HDF for the purpose of working with netCDF data objects and to understand enhancements to the API that will be included in the future to make the two APIs much more similar.

In the HDF model, when an n-dimensional SDS is created by **SDcreate**, data objects are also created that provide information about the individual dimensions - one for each dimension. Each SDS contains within its internal structure the array data as well as pointers to these dimensional data objects. Each dimensional data object is stored in a structure that is in the HDF file, but separate from the SDS array.

If more than one SDS have the same dimension sizes, they may share dimensions by pointing to the same dimensional data objects. This can be done in application programs by calling **SDsetdimname** to assign the same dimension name to all dimensions that are shared by several SDS objects. For example, suppose you make the following sequence of calls for every SDS in a file:

```
dim_id = SDgetdimid(sds_id, 0);
ret = SDsetdimname(dim_id, "Lat");
dim_id = SDgetdimid(sds_id, 1);
ret = SDsetdimname(dim_id, "Long");
```

This will cause every SDS to refer to the dimensional data object named "Lat" as its first dimension and to the dimensional data object named "Long" as its second dimension.

This same result is obtained differently in netCDF. Note that a netCDF "variable" is roughly the same as an HDF SDS. The netCDF API requires application programs to define all dimensions, using `ncdimdef`, before defining variables. Those defined dimensions are then used to define variables in `ncvardef`. Each dimension is defined by a name and a size. All variables using the same dimension will have the same dimension name and dimension size.

Although the HDF SDS API will read from and write to *existing* XDR-based netCDF files, HDF cannot be used to *create* XDR-based netCDF files.

There is currently no support for the mixing of HDF objects that are not SDSs and netCDF objects. For example, a raster image can exist in the same HDF file as a netCDF object, but you must use one of the HDF raster image interfaces to read the image and the HDF SD or netCDF interface to read the netCDF object. The other HDF interfaces are currently being modified to allow multifile access. Closer integration with the netCDF interface will probably be delayed until the end of that project.

3.12.1 HDF Interface vs. netCDF Interface

Existing netCDF applications can be used to read HDF files and existing HDF applications can be used to read XDR-based netCDF files. To read an HDF file using a netCDF application, the application must be recompiled using the HDF library. For example, recompiling the netCDF utility `ncdump` with HDF creates a utility that can dump scientific data sets from both HDF and XDR-based files. To read an XDR-based file using an HDF application, the application must be relinked to the HDF library.

The current version of HDF contains three APIs that support essentially the same data model:

- The multifile SD interface.
- The netCDF or NC interface.
- The single-file DFSD interface.
- The multifile GR interface.

The first three interfaces can create, read and write SDSs in HDF files. Both the SD and NC interfaces can read from and write to XDR-based netCDF files, but they cannot create them. This interoperability means that a single program may contain both SD and NC function calls and thus transparently read and write scientific data sets to HDF or XDR-based files.

The SD interface is the only HDF interface capable of accessing the XDR-based netCDF file format. The DFSD interface cannot access XDR-based files and can only access SDS arrays, dimension scales and predefined attributes. A summary of file interoperability among the three interfaces is provided in the following table.

TABLE 3AE **Summary of HDF and XDR File Compatibility for the HDF and netCDF APIs**

	Files Created by DFSD Interface	Files Created by SD Interface	Files Written by NC Interface	
	HDF	HDF	NCSA HDF Library	Unidata netCDF Library
Accessed by DFSD	Yes	Yes	Yes	No
Accessed by SD	Yes	Yes	Yes	Yes
Accessed by NC	Yes	Yes	Yes	Yes

A summary of NC function calls and their SD equivalents is presented in the following table.

TABLE 3AF

NC Interface Routine Calls and Their SD Equivalents

Purpose	Routine Name: C	Routine Name: Fortran-77	SD Equivalent	Description
Operations	nccreate	NCCRE	SDstart	Creates a file.
	ncopen	NCOPN	SDstart	Opens a file.
	ncredef	NCREDF	Not Applicable	Sets open file into define mode.
	ncendef	NCENDF	Not Applicable	Leaves define mode.
	ncclose	NCCLOS	SDend	Closes an open file.
	ncinquire	NCINQ	SDfileinfo	Inquires about an open file.
	ncsync	NCSNC	Not Applicable	Synchronizes a file to disk.
	ncabort	NCABOR	Not Applicable	Backs out of recent definitions.
	ncsetfill	NCSFIL	Not Implemented	Sets fill mode for writes.
Dimensions	ncdimdef	NCDDEF	SDsetdimname	Creates a dimension.
	ncdimid	NCDID	SDgetdimid	Returns a dimension identifier from its name.
	ncdiminq	NCDINQ	SDdiminfo	Inquires about a dimension.
	ncdimrename	NCDREN	Not Implemented	Renames a dimension.
Variables	ncvardef	NCVDEF	SDcreate	Creates a variable.
	ncvarid	NCVID	SDnametoindex and SDselect	Returns a variable identifier from its name.
	ncvarinq	NCVINQ	SDgetinfo	Returns information about a variable.
	ncvarput1	NCVPT1	Not Implemented	Writes a single data value.
	ncvarget1	NCVGT1	Not Implemented	Reads a single data value.
	ncvarput	NCVPT	SDwritedata	Writes a hyperslab of values.
	ncvarget	NCVGT/NCVGTC	SDreaddata	Reads a hyperslab of values.
	ncvarrename	NCVREN	Not Implemented	Renames a variable.
	nctypelen	NCTLEN	DFKNTsize	Returns the number of bytes for a data type.
Attributes	ncattput	NCAPT/NCAPTC	SDsetattr	Creates an attribute.
	ncattinq	NCAINQ	SDattrinfo	Returns information about an attribute.
	ncattcopy	NCACPY	Not Implemented	Copies attribute from one file to another.
	ncattget	NCAGT/NCAGTC	SDreadattr	Returns attributes values.
	ncattname	NCANAM	SDattrinfo	Returns name of attribute from its number.
	ncattrename	NCAREN	Not Implemented	Renames an attribute.
	ncattdel	NCADEL	Not Implemented	Deletes an attribute.