HDF5 Document Set

PDF and PS Versions

Release 1.2

October 1999

Hierarchical Data Format (HDF) Group
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)

A Note to the Reader: The primary HDF5 user documents are the online HTML documents distributed with the HDF5 code and binaries and found on the HDF5 website. These PDF and PostScript versions are generated from the HTML to provide the following capabilites:

- To provide a version that can be reasonably printed in a single print operation.
- To provide an easily searchable version.

In this package, you will find four PDF and PostScript documents:

- *Introduction to HDF5*
- *HDF5 Tutorial*
- *HDF5 User's Guide*
- *HDF5 Reference Manual*
- And all of the above documents concatenated into a single file

Note that these versions were created in response to user feedback; the HDF Group is eager to hear from HDF and HDF5 users so that we can better meet our users' needs. Send comments, requests, and bug reports to HDF Help at hdfhelp@ncsa.uiuc.edu.

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Last modified: 13 October 1999

Introduction to HDF5

Release 1.2
October 1999

Hierarchical Data Format (HDF) Group
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Last modified: 13 October 1999

Introduction to HDF5

Introduction to HDF5 Release 1.2

This is an introduction to the HDF5 data model and programming model. Being a *Getting Started* or *QuickStart* document, this *Introduction to HDF5* is intended to provide enough information for you to develop a basic understanding of how HDF5 works and is meant to be used. Knowledge of the current version of HDF will make it easier to follow the text, but it is not required. More complete information of the sort you will need to actually use HDF5 is available in the HDF5 documentation. Available documents include the following:

- *HDF5 User's Guide*. Where appropriate, this *Introduction* will refer to specific sections of the *User's Guide*.
- *HDF5 Reference Manual*.

Code examples are available in the source code tree when you install HDF5.

- The directories `hdf5/examples` and `hdf5/doc/html/Tutor/examples/` contain the examples used in this document.
- The directory `hdf5/test` contains the development tests used by the HDF5 developers. Since these codes are intended to fully exercise the system, they provide more diverse and sophisticated examples of what HDF5 can do.

1. What Is HDF5?

HDF5 is a completely new Hierarchical Data Format product consisting of a data format specification and a supporting library implementation. HDF5 is designed to address some of the limitations of the older HDF product and to address current and anticipated requirements of modern systems and applications. [1]

We urge you to look at HDF5, the format and the library, and give us feedback on what you like or do not like about it, and what features you would like to see added to it.

Why HDF5? The development of HDF5 is motivated by a number of limitations in the older HDF format and library. Some of these limitations are:

- A single file cannot store more than 20,000 complex objects, and a single file cannot be larger than 2 gigabytes.
- The data models are less consistent than they should be, there are more object types than necessary, and datatypes are too restricted.
- The library source is old and overly complex, does not support parallel I/O effectively, and is difficult to use in threaded applications.

HDF5 includes the following improvements.

- A new file format designed to address some of the deficiencies of HDF4.x, particularly the need to store larger files and more objects per file.
- A simpler, more comprehensive data model that includes only two basic structures: a multidimensional array of record structures, and a grouping structure.
- A simpler, better-engineered library and API, with improved support for parallel I/O, threads, and other requirements imposed by modern systems and applications.

1.1 Limitations of the Current Release

This release includes the basic functionality that was planned for the HDF5 library. However, the library does not implement all of the features detailed in the format and API specifications. Here is a listing of some of the limitations of the current release:

- Data compression is supported, though only GZIP is implemented. GZIP, or GNU Zip, is a compression function from the GNU Project.
- The library is not currently thread aware although we have planned for that possibility and intend eventually to implement it.

1.2 Changes in the Current Release

A detailed list of changes in HDF5 since the last release, HDF5 Release 1.0, can be found in the file `hdf5/RELEASE` in the source code installation. At a higher level, those changes include:

- Support for bitfield, opaque, enumeration, and variable-length datatypes
- Support for object and dataset region pointers
- Improved parallel performance and support for additional parallel platforms

- Improved and expanded documentation
- Enhancements to the `h5ls` and `h5dump` tools and a new HDF5 to HDF4 conversion tool, `h5toh4`
- Over 30 new API functions

The changes as HDF5 has evolved from the first Alpha release to the present are summarized in the file `hdf5/HISTORY` in the source code installation.

Footnote:

1. Note that HDF and HDF5 are two different products. HDF is a data format first developed in the 1980s and currently in Release 4.*x* (HDF Release 4.*x*). HDF5 is a new data format first released in *Beta* in 1998 and designed to better meet the ever-increasing demands of scientific computing and to take better advantage of the ever-increasing capabilities of computing systems. HDF5 is currently in Release 1.*x* (HDF5 Release 1.*x*).

2. HDF5 File Organization and Data Model

HDF5 files are organized in a hierarchical structure, with two primary structures: *groups* and *datasets*.

- *HDF5 group:* a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
- *HDF5 dataset:* a multidimensional array of data elements, together with supporting metadata.

Working with groups and group members is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, objects in an HDF5 file are often described by giving their full (or absolute) path names.

- / signifies the root group.
- /foo signifies a member of the root group called foo.
- /foo/zoo signifies a member of the group foo, which in turn is a member of the root group.

Any HDF5 group or dataset may have an associated *attribute list.* An HDF5 *attribute* is a user-defined HDF5 structure that provides extra information about an HDF5 object. Attributes are described in more detail below.

2.1 HDF5 Groups

An *HDF5 group* is a structure containing zero or more HDF5 objects. A group has two parts:

- A *group header*, which contains a group name and a list of group attributes.
- A group symbol table, which is a list of the HDF5 objects that belong to the group.

2.2 HDF5 Datasets

A dataset is stored in a file in two parts: a header and a data array.

The header contains information that is needed to interpret the array portion of the dataset, as well as metadata (or pointers to metadata) that describes or annotates the dataset. Header information includes the name of the object, its dimensionality, its number-type, information about how the data itself is stored on disk, and other information used by the library to speed up access to the dataset or maintain the file's integrity.

There are four essential classes of information in any header: *name*, *datatype*, *dataspace*, and *storage layout*:

Name. A dataset *name* is a sequence of alphanumeric ASCII characters.

Datatype. HDF5 allows one to define many different kinds of datatypes. There are two categories of datatypes: *atomic* datatypes and *compound* datatypes. Atomic datatypes can also be system-specific, or `NATIVE`, and all datatypes can be *named*:

- *Atomic* datatypes are those that are not decomposed at the datatype interface level, such as integers and floats.
- `NATIVE` datatypes are system-specific instances of atomic datatypes.
- Compound datatypes are made up of atomic datatypes.
- *Named* datatypes are either atomic or compound datatypes that have been specifically designated to be shared across datasets.

Atomic datatypes include integers and floating-point numbers. Each atomic type belongs to a particular class and has several properties: size, order, precision, and offset. In this introduction, we consider only a few of these properties.

Atomic classes include integer, float, date and time, string, bit field, and opaque. *(Note: Only integer, float and string classes are available in the current implementation.)*

Properties of integer types include size, order (endian-ness), and signed-ness (signed/unsigned).

Properties of float types include the size and location of the exponent and mantissa, and the location of the sign bit.

The datatypes that are supported in the current implementation are:

- Integer datatypes: 8-bit, 16-bit, 32-bit, and 64-bit integers in both little and big-endian format.
- Floating-point numbers: IEEE 32-bit and 64-bit floating-point numbers in both little and big-endian format.
- References.
- Strings.

`NATIVE` *datatypes.* Although it is possible to describe nearly any kind of atomic datatype, most applications will use predefined datatypes that are supported by their compiler. In HDF5 these are called *native* datatypes. `NATIVE` datatypes are C-like datatypes that are generally supported by the hardware of the machine on which the library was compiled. In order to be portable, applications should almost always use the `NATIVE` designation to describe data values in memory.

The `NATIVE` architecture has base names which do not follow the same rules as the others. Instead, native type names are similar to the C type names. The following figure shows several examples.

Examples of Native Datatypes and Corresponding C Types

Example	Corresponding C Type
`H5T_NATIVE_CHAR`	`signed char`
`H5T_NATIVE_UCHAR`	`unsigned char`
`H5T_NATIVE_SHORT`	`short`
`H5T_NATIVE_USHORT`	`unsigned short`
`H5T_NATIVE_INT`	`int`
`H5T_NATIVE_UINT`	`unsigned`
`H5T_NATIVE_LONG`	`long`
`H5T_NATIVE_ULONG`	`unsigned long`
`H5T_NATIVE_LLONG`	`long long`
`H5T_NATIVE_ULLONG`	`unsigned long long`
`H5T_NATIVE_FLOAT`	`float`
`H5T_NATIVE_DOUBLE`	`double`

`H5T_NATIVE_LDOUBLE`	`long double`
`H5T_NATIVE_HSIZE`	`hsize_t`
`H5T_NATIVE_HSSIZE`	`hssize_t`
`H5T_NATIVE_HERR`	`herr_t`
`H5T_NATIVE_HBOOL`	`hbool_t`

See *Datatypes* in the *HDF User's Guide* for further information.

A *compound datatype* is one in which a collection of simple datatypes are represented as a single unit, similar to a *struct* in C. The parts of a compound datatype are called *members.* The members of a compound datatype may be of any datatype, including another compound datatype. It is possible to read members from a compound type without reading the whole type.

Named datatypes. Normally each dataset has its own datatype, but sometimes we may want to share a datatype among several datasets. This can be done using a *named* datatype. A named datatype is stored in the file independently of any dataset, and referenced by all datasets that have that datatype. Named datatypes may have an associated attributes list. See *Datatypes* in the *HDF User's Guide* for further information.

Dataspace. A dataset *dataspace* describes the dimensionality of the dataset. The dimensions of a dataset can be fixed (unchanging), or they may be *unlimited*, which means that they are extendible (i.e. they can grow larger).

Properties of a dataspace consist of the *rank* (number of dimensions) of the data array, the *actual sizes of the dimensions* of the array, and the *maximum sizes of the dimensions* of the array. For a fixed-dimension dataset, the actual size is the same as the maximum size of a dimension. When a dimension is unlimited, the maximum size is set to the value `H5P_UNLIMITED`. (An example below shows how to create extendible datasets.)

A dataspace can also describe portions of a dataset, making it possible to do partial I/O operations on *selections*. *Selection* is supported by the dataspace interface (H5S). Given an n-dimensional dataset, there are currently four ways to do partial selection:

- Select a logically contiguous n-dimensional hyperslab.
- Select a non-contiguous hyperslab consisting of elements or blocks of elements (hyperslabs) that are equally spaced.
- Select a union of hyperslabs.
- Select a list of independent points.

Since I/O operations have two end-points, the raw data transfer functions require two dataspace arguments: one describes the application memory dataspace or subset thereof, and the other describes the file dataspace or subset thereof.

See *Dataspaces* in the *HDF User's Guide* for further information.

Storage layout. The HDF5 format makes it possible to store data in a variety of ways. The default storage layout format is *contiguous*, meaning that data is stored in the same linear way that it is organized in memory. Two other storage layout formats are currently defined for HDF5: *compact*, and *chunked.* In the future, other storage layouts may be added.

Compact storage is used when the amount of data is small and can be stored directly in the object header. *(Note: Compact storage is not supported in this release.)*

Chunked storage involves dividing the dataset into equal-sized "chunks" that are stored separately. Chunking has three important benefits.

- It makes it possible to achieve good performance when accessing subsets of the datasets, even when the subset to be chosen is orthogonal to the normal storage order of the dataset.
- It makes it possible to compress large datasets and still achieve good performance when accessing subsets of the dataset.
- It makes it possible efficiently to extend the dimensions of a dataset in any direction.

See *Datasets* and *Dataset Chunking Issues* in the *HDF User's Guide* for further information. We particularly encourage you to read *Dataset Chunking Issues* since the issue is complex and beyond the scope of this document.

2.3 HDF5 Attributes

Attributes are small named datasets that are attached to primary datasets, groups, or named datatypes. Attributes can be used to describe the nature and/or the intended usage of a dataset or group. An attribute has two parts: (1) a *name* and (2) a *value*. The value part contains one or more data entries of the same datatype.

The Attribute API (H5A) is used to read or write attribute information. When accessing attributes, they can be identified by name or by an *index value*. The use of an index value makes it possible to iterate through all of the attributes associated with a given object.

The HDF5 format and I/O library are designed with the assumption that attributes are small datasets. They are always stored in the object header of the object they are attached to. Because of this, large datasets should not be stored as attributes. How large is "large" is not defined by the library and is up to the user's interpretation. (Large datasets with metadata can be stored as supplemental datasets in a group with the primary dataset.)

See *Attributes* in the *HDF User's Guide* for further information.

2.4 The File as Written to Media

For those who are interested, this section takes a look at the low-level elements of the file as the file is written to disk (or other storage media) and the relation of those low-level elements to the higher level elements with which users typically are more familiar. The HDF5 API generally exposes only the high-level elements to the user; the low-level elements are often hidden. The rest of this *Introduction* does not assume an understanding of this material.

The format of an HDF5 file on disk encompasses several key ideas of the HDF4 and AIO file formats as well as addressing some shortcomings therein. The new format is more self-describing than the HDF4 format and is more uniformly applied to data objects in the file.

An HDF5 file appears to the user as a directed graph. The nodes of this graph are the higher-level HDF5 objects that are exposed by the HDF5 APIs:

- Groups
- Datasets
- Datatypes
- Dataspaces

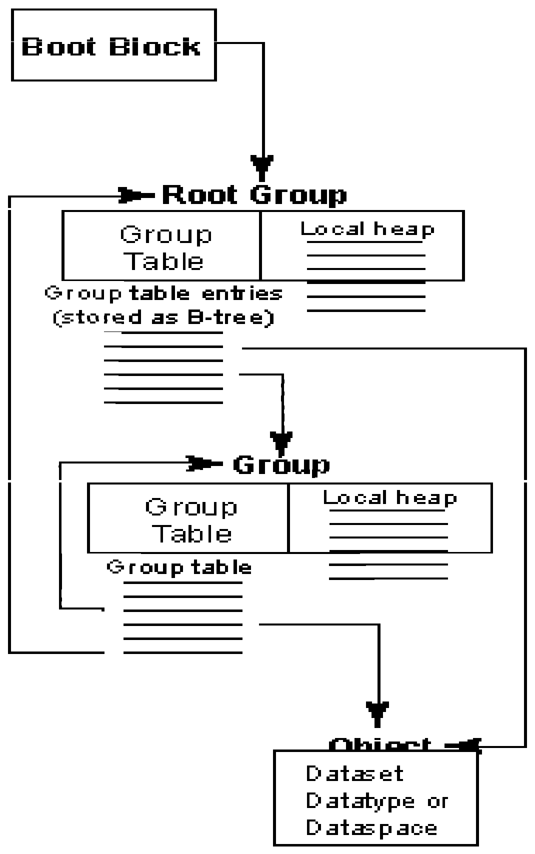


Figure 1: Relationships among the HDF5 root group, other groups, and objects

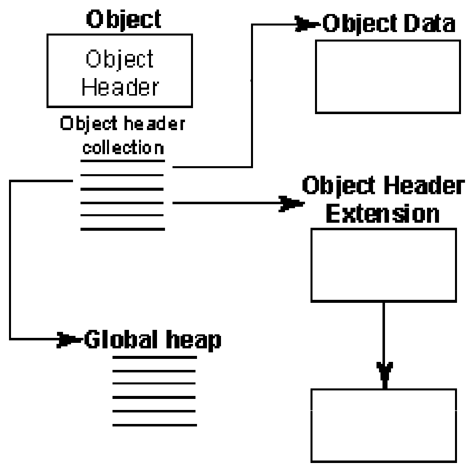


Figure 2: HDF5 objects -- datasets, datatypes, or dataspaces

At the lowest level, as information is actually written to the disk, an HDF5 file is made up of the following objects:

- A boot block
- B-tree nodes (containing either symbol nodes or raw data chunks)
- Object headers
- Collections
- Local heaps
- Free space

The HDF5 library uses these lower-level objects to represent the higher-level objects that are then presented to the user or to applications through the APIs. For instance, a group is an object header that contains a message that points to a local heap and to a B-tree which points to symbol nodes. A dataset is an object header that contains messages that describe datatype, space, layout, filters, external files, fill value, etc with the layout message pointing to either a raw data chunk or to a B-tree that points to raw data chunks.

See the *HDF5 File Format Specification* for further information.

3. The HDF5 Applications Programming Interface (API)

The current HDF5 API is implemented only in C. The API provides routines for creating HDF5 files, creating and writing groups, datasets, and their attributes to HDF5 files, and reading groups, datasets and their attributes from HDF5 files.

Naming conventions

All C routines in the HDF 5 library begin with a prefix of the form **H5***, where ***** is a single letter indicating the object on which the operation is to be performed:

- **H5F**: **F**ile-level access routines.
  Example: `H5Fopen`, which opens an HDF5 file.
- **H5G**: **G**roup functions, for creating and operating on groups of objects.
  Example: `H5Gset,` which sets the working group to the specified group.
- **H5T:** Data**T**ype functions, for creating and operating on simple and compound datatypes to be used as the elements in data arrays.
  Example: `H5Tcopy,` which creates a copy of an existing datatype.
- **H5S:** Data**S**pace functions, which create and manipulate the dataspace in which the elements of a data array are stored.
  Example: `H5Screate_simple`, which creates simple dataspaces.
- **H5D: D**ataset functions, which manipulate the data within datasets and determine how the data is to be stored in the file.
  Example: `H5Dread`, which reads all or part of a dataset into a buffer in memory.
- **H5P**: **P**roperty list functions, for manipulating object creation and access properties.
  Example: `H5Pset_chunk`, which sets the number of dimensions and the size of a chunk.
- **H5A**: **A**ttribute access and manipulating routines.
  Example: `H5Aget_name`, which retrieves name of an attribute.
- **H5Z**: **C**ompression registration routine.
  Example: `H5Zregister`, which registers new compression and uncompression functions for use with the HDF5 library.
- **H5E**: **E**rror handling routines.
  Example: `H5Eprint`, which prints the current error stack.
- **H5R**: **R**eference routines.
  Example: `H5Rcreate`, which creates a reference.
- **H5I**: **I**dentifier routine.
  Example: `H5Iget_type`, which retrieves the type of an object.

3.1 Include Files

There are a number definitions and declarations that should be included with any HDF5 program. These definitions and declarations are contained in several *include* files. The main include file is `hdf5.h`. This file includes all of the other files that your program is likely to need. *Be sure to include* `hdf5.h` *in any program that uses the HDF5 library.*

3.2 Programming Models

In this section we describe how to program some basic operations on files, including how to

- Create a file.
- Create and initialize a dataset.
- Discard objects when they are no longer needed.
- Write a dataset to a new file.
- Obtain information about a dataset.
- Read a portion of a dataset.
- Create and write compound datatypes.
- Create and write extendible datasets.
- Create and populate groups.
- Work with attributes.

How to create an HDF5 file

This programming model shows how to create a file and also how to close the file.

1. Create the file.
2. Close the file.

The following code fragment implements the specified model. If there is a possibility that the file already exists, the user must add the flag `H5ACC_TRUNC` to the access mode to overwrite the previous file's information.

```
hid_t       file;                          /* identifier */
/*
* Create a new file using H5ACC_TRUNC access,
* default file creation properties, and default file
* access properties.
* Then close the file.
*/
file = H5Fcreate(FILE, H5ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
status = H5Fclose(file);
```

How to create and initialize the essential components of a dataset for writing to a file

Recall that datatypes and dimensionality (dataspace) are independent objects, which are created separately from any dataset that they might be attached to. Because of this the creation of a dataset requires, at a minimum, separate definitions of datatype, dimensionality, and dataset. Hence, to create a dataset the following steps need to be taken:

1. Create and initialize a dataspace for the dataset to be written.
2. Define the datatype for the dataset to be written.
3. Create and initialize the dataset itself.

The following code illustrates the creation of these three components of a dataset object.

```
hid_t    dataset, datatype, dataspace;   /* declare identifiers */

/*
 * Create dataspace: Describe the size of the array and
 * create the data space for fixed size dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
dataspace = H5Screate_simple(RANK, dimsf, NULL);
/*
 * Define datatype for the data in the file.
 * We will store little endian integer numbers.
 */
datatype = H5Tcopy(H5T_NATIVE_INT);
status = H5Tset_order(datatype, H5T_ORDER_LE);
/*
 * Create a new dataset within the file using defined
 * dataspace and datatype and default dataset creation
 * properties.
 * NOTE: H5T_NATIVE_INT can be used as datatype if conversion
 * to little endian is not needed.
 */
dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace, H5P_DEFAULT);
```

How to discard objects when they are no longer needed

The datatype, dataspace and dataset objects should be released once they are no longer needed by a program. Since each is an independent object, the must be released (or *closed*) separately. The following lines of code close the datatype, dataspace, and datasets that were created in the preceding section.

```
H5Tclose(datatype);

H5Dclose(dataset);

H5Sclose(dataspace);
```

How to write a dataset to a new file

Having defined the datatype, dataset, and dataspace parameters, you write out the data with a call to `H5Dwrite.`

```
/*
* Write the data to the dataset using default transfer
* properties.
*/
status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
         H5P_DEFAULT, data);
```

The third and fourth parameters of `H5Dwrite` in the example describe the dataspaces in memory and in the file, respectively. They are set to the value `H5S_ALL` to indicate that an entire dataset is to be written. In a later section we look at how we would access a portion of a dataset.

Example 1 contains a program that creates a file and a dataset, and writes the dataset to the file.

Reading is analogous to writing. If, in the previous example, we wish to read an entire dataset, we would use the same basic calls with the same parameters. Of course, the routine `H5Dread` would replace `H5Dwrite.`

Getting information about a dataset

Although reading is analogous to writing, it is often necessary to query a file to obtain information about a dataset. For instance, we often need to know about the datatype associated with a dataset, as well dataspace information (e.g. rank and dimensions). There are several "get" routines for obtaining this information. The following code segment illustrates how we would get this kind of information:

```
/*
* Get datatype and dataspace identifiers and then query
* dataset class, order, size, rank and dimensions.
*/

datatype  = H5Dget_type(dataset);     /* datatype identifier */
class     = H5Tget_class(datatype);
if (class == H5T_INTEGER) printf("Data set has INTEGER type \n");
order     = H5Tget_order(datatype);
if (order == H5T_ORDER_LE) printf("Little endian order \n");

size  = H5Tget_size(datatype);
printf(" Data size is %d \n", size);

dataspace = H5Dget_space(dataset);    /* dataspace identifier */
rank      = H5Sget_simple_extent_ndims(dataspace);
status_n  = H5Sget_simple_extent_dims(dataspace, dims_out);
printf("rank %d, dimensions %d x %d \n", rank, dims_out[0], dims_out[1]);
```

Reading and writing a portion of a dataset

In the previous discussion, we describe how to access an entire dataset with one write (or read) operation. HDF5 also supports access to portions (or selections) of a dataset in one read/write operation. Currently selections are limited to hyperslabs, their unions, and the lists of independent points. Both types of selection will be discussed in the following sections. Several sample cases of selection reading/writing are shown on the figure on the following page.

In example (a) a single hyperslab is read from the midst of a two-dimensional array in a file and stored in the corner of a smaller two-dimensional array in memory. In (b) a regular series of blocks is read from a two-dimensional array in the file and stored as a contiguous sequence of values at a certain offset in a one-dimensional array in memory. In (c) a sequence of points with no regular pattern is read from a two-dimensional array in a file and stored as a sequence of points with no regular pattern in a three-dimensional array in memory. In (d) a union of hyperslabs in the file dataspace is read and the data is stored in another union of hyperslabs in the memory dataspace.

As these examples illustrate, whenever we perform partial read/write operations on the data, the following information must be provided: file dataspace, file dataspace selection, memory dataspace and memory dataspace selection. After the required information is specified, actual read/write operation on the portion of data is done in a single call to the HDF5 read/write functions H5Dread(write).

Selecting hyperslabs

Hyperslabs are portions of datasets. A hyperslab selection can be a logically contiguous collection of points in a dataspace, or it can be regular pattern of points or blocks in a dataspace. The following picture illustrates a selection of regularly spaced 3x2 blocks in an 8x12 dataspace.

Hyperslab selection

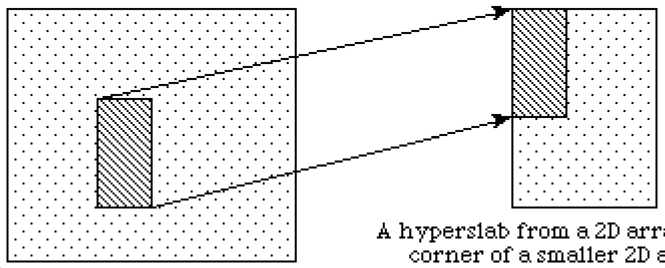
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X

Four parameters are required to describe a completely general hyperslab. Each parameter is an array whose rank is the same as that of the dataspace:

- `start`: a starting location for the hyperslab. In the example `start` is (0,1).
- `stride`: the number of elements to separate each element or block to be selected. In the example `stride` is (4,3). If the stride parameter is set to NULL, the stride size defaults to 1 in each dimension.

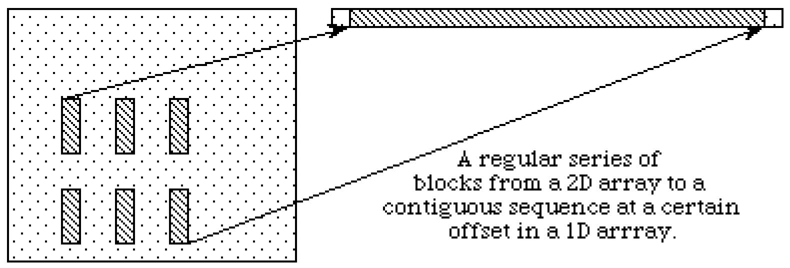
Mappings between File Dataspaces and Selections and Memory Dataspaces and Selections

File dataspace and selection **Memory dataspace and selection**



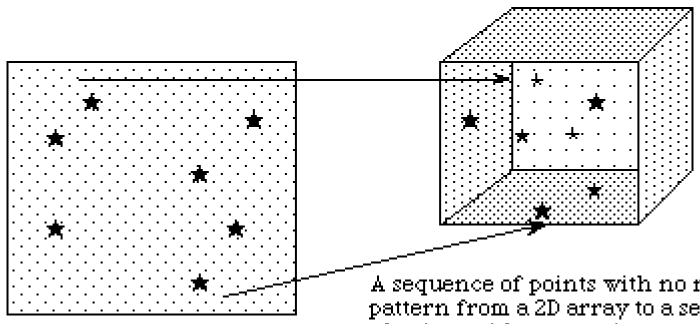
A hyperslab from a 2D array to the corner of a smaller 2D array.

a



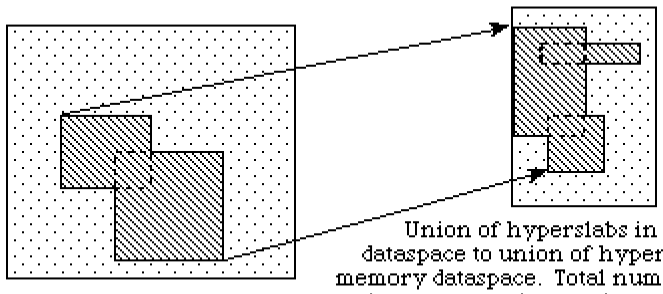
A regular series of blocks from a 2D array to a contiguous sequence at a certain offset in a 1D arrray.

b



A sequence of points with no regular pattern from a 2D array to a sequence of points with no regular pattern in a 3D array.

c



Union of hyperslabs in file dataspace to union of hyperslabs in memory dataspace. Total number of data elements must be equal; number and shape of hyperslabs can differ.

d

Key: Dataspace ░ Selection ▨ or ★ (single point)

- `count`: the number of elements or blocks to select along each dimension. In the example, `count` is (2,4).

- `block`: the size of the block selected from the dataspace. In the example, `block` is (3,2). If the block parameter is set to NULL, the block size defaults to a single element in each dimension, as if the block array was set to all 1s.

In what order is data copied? When actual I/O is performed data values are copied by default from one dataspace to another in so-called row-major, or C order. That is, it is assumed that the first dimension varies slowest, the second next slowest, and so forth.

Example without strides or blocks. Suppose we want to read a 3x4 hyperslab from a dataset in a file beginning at the element `<1,2>` in the dataset. In order to do this, we must create a dataspace that describes the overall rank and dimensions of the dataset in the file, as well as the position and size of the hyperslab that we are extracting from that dataset. The following code illustrates the selection of the hyperslab in the file dataspace.

```
/*
* Define file dataspace.
*/
dataspace = H5Dget_space(dataset);    /* dataspace identifier */
rank      = H5Sget_simple_extent_ndims(dataspace);
status_n  = H5Sget_simple_extent_dims(dataspace, dims_out, NULL);

/*
* Define hyperslab in the dataset.
*/
offset[0] = 1;
offset[1] = 2;
count[0]  = 3;
count[1]  = 4;
status = H5Sselect_hyperslab(dataspace, H5S_SELECT_SET, offset, NULL,
         count, NULL);
```

This describes the dataspace from which we wish to read. We need to define the dataspace in memory analogously. Suppose, for instance, that we have in memory a 3 dimensional 7x7x3 array into which we wish to read the 3x4 hyperslab described above beginning at the element `<3,0,0>`. Since the in-memory dataspace has three dimensions, we have to describe the hyperslab as an array with three dimensions, with the last dimension being 1: `<3,4,1>`.

Notice that we must describe two things: the dimensions of the in-memory array, and the size and position of the hyperslab that we wish to read in. The following code illustrates how this would be done.

```
/*
* Define memory dataspace.
*/
dimsm[0] = 7;
dimsm[1] = 7;
dimsm[2] = 3;
memspace = H5Screate_simple(RANK_OUT,dimsm,NULL);

/*
* Define memory hyperslab.
*/
offset_out[0] = 3;
offset_out[1] = 0;
offset_out[2] = 0;
count_out[0]  = 3;
count_out[1]  = 4;
count_out[2]  = 1;
status = H5Sselect_hyperslab(memspace, H5S_SELECT_SET, offset_out, NULL,
         count_out, NULL);
```

```
/*
```

Example 2 contains a complete program that performs these operations.

Example with strides and blocks. Consider the 8x12 dataspace described above, in which we selected eight 3x2 blocks. Suppose we wish to fill these eight blocks.

Hyperslab selection

	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X
	X	X		X	X		X	X		X	X

This hyperslab has the following parameters: `start=(0,1), stride=(4,3), count=(2,4), block=(3,2).`

Suppose that the source dataspace in memory is this 50-element one dimensional array called `vector`:

A 50-element one dimensional array

-1	1	2	3	4	5	6	7	...	47	48	-1

The following code will write 48 elements from `vector` to our file dataset, starting with the second element in `vector`.

```
/* Select hyperslab for the dataset in the file, using 3x2 blocks, (4,3) stride
 * (2,4) count starting at the position (0,1).
 */
start[0]  = 0; start[1]  = 1;
stride[0] = 4; stride[1] = 3;
count[0]  = 2; count[1]  = 4;
block[0]  = 3; block[1]  = 2;
ret = H5Sselect_hyperslab(fid, H5S_SELECT_SET, start, stride, count, block);

/*
 * Create dataspace for the first dataset.
 */
mid1 = H5Screate_simple(MSPACE1_RANK, dim1, NULL);

/*
 * Select hyperslab.
 * We will use 48 elements of the vector buffer starting at the second element.
 * Selected elements are 1 2 3 . . . 48
 */
start[0]  = 1;
stride[0] = 1;
count[0]  = 48;
```

```
block[0]  = 1;
ret = H5Sselect_hyperslab(mid1, H5S_SELECT_SET, start, stride, count, block);

/*
 * Write selection from the vector buffer to the dataset in the file.
 *
ret = H5Dwrite(dataset, H5T_NATIVE_INT, midd1, fid, H5P_DEFAULT, vector)
```

After these operations, the file dataspace will have the following values.

Hyperslab selection with assigned values

	1	2		3	4		5	6		7	8
	9	10		11	12		13	14		15	16
	17	18		19	20		21	22		23	24
	25	26		27	28		29	30		31	32
	33	34		35	36		37	38		39	40
	41	42		43	44		45	46		47	48

Notice that the values are inserted in the file dataset in row-major order.

Example 3 includes this code and other example code illustrating the use of hyperslab selection.

Selecting a list of independent points

A hyperslab specifies a regular pattern of elements in a dataset. It is also possible to specify a list of independent elements to read or write using the function `H5Sselect_elements`. Suppose, for example, that we wish to write the values 53, 59, 61, 67 to the following elements of the 8x12 array used in the previous example: (0,0), (3,3), (3,5), and (5,6). The following code selects the points and writes them to the dataset:

```
#define FSPACE_RANK      2    /* Dataset rank as it is stored in the file */
#define NPOINTS          4    /* Number of points that will be selected
                                 and overwritten */
#define MSPACE2_RANK     1    /* Rank of the second dataset in memory */
#define MSPACE2_DIM      4    /* Dataset size in memory */


hsize_t dim2[] = {MSPACE2_DIM};       /* Dimension size of the second
                                         dataset (in memory) */
int     values[] = {53, 59, 61, 67};  /* New values to be written */
hssize_t coord[NPOINTS][FSPACE_RANK]; /* Array to store selected points
                                         from the file dataspace */

/*
 * Create dataspace for the second dataset.
 */
mid2 = H5Screate_simple(MSPACE2_RANK, dim2, NULL);
```

```
/*
 * Select sequence of NPOINTS points in the file dataspace.
 */
coord[0][0] = 0; coord[0][1] = 0;
coord[1][0] = 3; coord[1][1] = 3;
coord[2][0] = 3; coord[2][1] = 5;
coord[3][0] = 5; coord[3][1] = 6;

ret = H5Sselect_elements(fid, H5S_SELECT_SET, NPOINTS,
                         (const hssize_t **)coord);

/*
 * Write new selection of points to the dataset.
 */
ret = H5Dwrite(dataset, H5T_NATIVE_INT, mid2, fid, H5P_DEFAULT, values);
```

After these operations, the file dataspace will have the following values:

Hyperslab selection with an overlay of independent points

53	1	2		3	4		5	6		7	8
	9	10		11	12		13	14		15	16
	17	18		19	20		21	22		23	24
			59		**61**						
	25	26		27	28		29	30		31	32
	33	34		35	36	**67**	37	38		39	40
	41	42		43	44		45	46		47	48

Example 3 contains a complete program that performs these subsetting operations.

Selecting a union of hyperslabs

The HDF5 Library allows the user to select a union of hyperslabs and write or read the selection into another selection. The shapes of the two selections may differ, but the number of elements must be equal.

Suppose that we want to read two overlapping hyperslabs from the dataset written in the previous example into a union of hyperslabs in the memory dataset. This exercise is illustrated in the two figures immediately below. Note that the memory dataset has a different shape from the previously written dataset. Similarly, the selection in the memory dataset could have a different shape than the selected union of hyperslabs in the original file; for simplicity, we will preserve the selection's shape in this example.

Selection of a union of hyperslabs in a file dataset

53	1	2		3	4		5	6		7	8
	9	10		11	12		13	14		15	16
	17	18		19	20		21	22		23	24
			59		**61**						
	25	26		27	28		29	30		31	32
	33	34		35	36	**67**	37	38		39	40
	41	42		43	44		45	46		47	48

Selection of a union of hyperslabs in a memory dataset
Blank cells in this figure actually contain values written when the dataset was initialized.

10		11	12					
18		19	20		21	22		
	59		**61**					
		27	28		29	30		
		35	36	**67**	37	38		
		43	44		45	46		

(Note: The above tables highlight hyperslab selections with green, blue, and yellow shading. This shading may not appear properly in black-and-white printed copies.)

The following lines of code show the required steps.

First obtain the dataspace identifier for the dataset in the file.

```
    /*
     * Get dataspace of the open dataset.
     */
    fid = H5Dget_space(dataset);
```

Then select the hyperslab with the size 3x4 and the left upper corner at the position (1,2):

```
    /*
     * Select first hyperslab for the dataset in the file. The following
     * elements are selected:
     *                     10  0 11 12
     *                     18  0 19 20
     *                      0 59  0 61
     *
     */
    start[0] = 1; start[1] = 2;
    block[0] = 1; block[1] = 1;
    stride[0] = 1; stride[1] = 1;
    count[0]  = 3; count[1]  = 4;
    ret = H5Sselect_hyperslab(fid, H5S_SELECT_SET, start, stride, count, block);
```

Now select the second hyperslab with the size 6x5 at the position (2,4), and create the union with the first hyperslab.

```
    /*
     * Add second selected hyperslab to the selection.
     * The following elements are selected:
     *                    19 20  0 21 22
     *                     0 61  0  0  0
     *                    27 28  0 29 30
     *                    35 36 67 37 38
     *                    43 44  0 45 46
     *                     0  0  0  0  0
     * Note that two hyperslabs overlap. Common elements are:
     *                                                19 20
     *                                                 0 61
     */
    start[0] = 2; start[1] = 4;
    block[0] = 1; block[1] = 1;
    stride[0] = 1; stride[1] = 1;
    count[0]  = 6; count[1]  = 5;
    ret = H5Sselect_hyperslab(fid, H5S_SELECT_OR, start, stride, count, block);
```

Note that when we add the selected hyperslab to the union, the second argument to the `H5Sselect_hyperslab` function has to be `H5S_SELECT_OR` instead of `H5S_SELECT_SET`. Using `H5S_SELECT_SET` would reset the selection to the second hyperslab.

Now define the memory dataspace and select the union of the hyperslabs in the memory dataset.

```
/*
 * Create memory dataspace.
 */
mid = H5Screate_simple(MSPACE_RANK, mdim, NULL);

/*
 * Select two hyperslabs in memory. Hyperslabs has the same
 * size and shape as the selected hyperslabs for the file dataspace.
 */
start[0] = 0; start[1] = 0;
block[0] = 1; block[1] = 1;
stride[0] = 1; stride[1] = 1;
count[0]  = 3; count[1]  = 4;
ret = H5Sselect_hyperslab(mid, H5S_SELECT_SET, start, stride, count, block);
start[0] = 1; start[1] = 2;
block[0] = 1; block[1] = 1;
stride[0] = 1; stride[1] = 1;
count[0]  = 6; count[1]  = 5;
ret = H5Sselect_hyperslab(mid, H5S_SELECT_OR, start, stride, count, block);
```

Finally we can read the selected data from the file dataspace to the selection in memory with one call to the `H5Dread` function.

```
ret = H5Dread(dataset, H5T_NATIVE_INT, mid, fid, H5P_DEFAULT, matrix_out);
```

Example 3 includes this code along with the previous selection example.

Creating compound datatypes

Properties of compound datatypes. A compound datatype is similar to a struct in C or a common block in Fortran. It is a collection of one or more atomic types or small arrays of such types. To create and use of a compound datatype you need to refer to various *properties* of the data compound datatype:

- It is of class *compound*.
- It has a fixed total *size*, in bytes.
- It consists of zero or more *members* (defined in any order) with unique names and which occupy non-overlapping regions within the datum.
- Each member has its own *datatype*.
- Each member is referenced by an *index number* between zero and N-1, where N is the number of members in the compound datatype.
- Each member has a *name* which is unique among its siblings in a compound datatype.
- Each member has a fixed *byte offset*, which is the first byte (smallest byte address) of that member in a compound datatype.
- Each member can be a small array of up to four dimensions.

Properties of members of a compound datatype are defined when the member is added to the compound type and cannot be subsequently modified.

Defining compound datatypes. Compound datatypes must be built out of other datatypes. First, one creates an empty compound datatype and specifies its total size. Then members are added to the compound datatype in any order.

Member names. Each member must have a descriptive name, which is the key used to uniquely identify the member within the compound datatype. A member name in an HDF5 datatype does not necessarily have to be the same as the name of the corresponding member in the C struct in memory, although this is often the case. Nor does one need to define all members of the C struct in the HDF5 compound datatype (or vice versa).

Offsets. Usually a C struct will be defined to hold a data point in memory, and the offsets of the members in memory will be the offsets of the struct members from the beginning of an instance of the struct. The library defines the macro to compute the offset of a member within a struct:

```
  HOFFSET(s,m)
```

This macro computes the offset of member *m* within a struct variable *s*.

Here is an example in which a compound datatype is created to describe complex numbers whose type is defined by the `complex_t` struct.

```
typedef struct {
   double re;   /*real part */
   double im;   /*imaginary part */
} complex_t;

complex_t tmp;  /*used only to compute offsets */
hid_t complex_id = H5Tcreate (H5T_COMPOUND, sizeof tmp);
H5Tinsert (complex_id, "real", HOFFSET(tmp,re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (complex_id, "imaginary", HOFFSET(tmp,im),
           H5T_NATIVE_DOUBLE);
```

Example 4 shows how to create a compound datatype, write an array that has the compound datatype to the file, and read back subsets of the members.

Creating and writing extendible and chunked datasets

An *extendible* dataset is one whose dimensions can grow. In HDF5, it is possible to define a dataset to have certain initial dimensions, then later to increase the size of any of the initial dimensions.

For example, you can create and store the following 3x3 HDF5 dataset:

```
1 1 1
1 1 1
1 1 1
```

then later to extend this into a 10x3 dataset by adding 7 rows, such as this:

```
1 1 1
1 1 1
1 1 1
2 2 2
2 2 2
2 2 2
2 2 2
2 2 2
2 2 2
2 2 2
```

then further extend it to a 10x5 dataset by adding two columns, such as this:

```
1 1 1 3 3
1 1 1 3 3
1 1 1 3 3
2 2 2 3 3
2 2 2 3 3
2 2 2 3 3
2 2 2 3 3
2 2 2 3 3
2 2 2 3 3
2 2 2 3 3
```

HDF 5 requires you to use *chunking* in order to define extendible datasets. Chunking makes it possible to extend datasets efficiently, without having to reorganize storage excessively.

The following operations are required in order to write an extendible dataset:

1. Declare the dataspace of the dataset to have *unlimited dimensions* for all dimensions that might eventually be extended.

2. Set dataset creation properties to enable chunking and create a dataset.

3. Extend the size of the dataset.

For example, suppose we wish to create a dataset similar to the one shown above. We want to start with a 3x3 dataset, then later extend it in both directions.

Declaring unlimited dimensions. We could declare the dataspace to have unlimited dimensions with the following code, which uses the predefined constant `H5S_UNLIMITED` to specify unlimited dimensions.

```
hsize_t dims[2] = { 3, 3}; /* dataset dimensions
at the creation time */
hsize_t maxdims[2] = {H5S_UNLIMITED, H5S_UNLIMITED};
/*
 * Create the data space with unlimited dimensions.
 */
dataspace = H5Screate_simple(RANK, dims, maxdims);
```

Enabling chunking. We can then set the dataset storage layout properties to enable chunking. We do this using the routine `H5Pset_chunk`:

```
hid_t cparms;
hsize_t chunk_dims[2] ={2, 5};
/*
 * Modify dataset creation properties to enable chunking.
 */
cparms = H5Pcreate (H5P_DATASET_CREATE);
status = H5Pset_chunk( cparms, RANK, chunk_dims);
```

Then create a dataset.

```
/*
 * Create a new dataset within the file using cparms
 * creation properties.
 */
dataset = H5Dcreate(file, DATASETNAME, H5T_NATIVE_INT, dataspace,
                 cparms);
```

Extending dataset size. Finally, when we want to extend the size of the dataset, we invoke `H5Dextend` to extend the size of the dataset. In the following example, we extend the dataset along the first dimension, by seven rows, so that the new dimensions are `<10,3>`:

```
/*
 * Extend the dataset. Dataset becomes 10 x 3.
 */
dims[0] = dims[0] + 7;
size[0] = dims[0];
size[1] = dims[1];
status = H5Dextend (dataset, size);
```

Example 5 shows how to create a 3x3 extendible dataset, write the dataset, extend the dataset to 10x3, write the dataset again, extend it again to 10x5, write the dataset again. Example 6 shows how to read the data written by Example 5.

Working with groups in files

Groups provide a mechanism for organizing meaningful and extendible sets of datasets within an HDF5 file. The H5G API contains routines for working with groups.

Creating a group. To create a group, use `H5Gcreate`. For example, the following code creates a group called `Data` in the root group.

```
/*
 *  Create a group in the file.
 */
grp = H5Gcreate(file, "/Data", 0);
```

A group may be created in another group by providing the absolute name of the group to the `H5Gcreate` function or by specifying its location. For example, to create the group `Data_new` in the `Data` group, one can use the following sequence of calls:

```
/*
 * Create group "Data_new" in the group "Data" by specifying
 * absolute name of the group.
 */
grp_new = H5Gcreate(file, "/Data/Data_new", 0);
```

or

```
/*
 * Create group "Data_new" in the "Data" group.
 */
grp_new = H5Gcreate(grp, "Data_new", 0);
```

Note that the group identifier `grp` is used as the first parameter in the `H5Gcreate` function when the relative name is provided.

The third parameter in `H5Gcreate` optionally specifies how much file space to reserve to store the names that will appear in this group. If a non-positive value is supplied, then a default size is chosen.

`H5Gclose` closes the group and releases the group identifier.

Creating a dataset in a particular group. As with groups, a dataset can be created in a particular group by specifying its absolute name as illustrated in the following example:

```
/*
 * Create the dataset "Compressed_Data" in the group using the
 * absolute name. The dataset creation property list is modified
 * to use GZIP compression with the compression effort set to 6.
 * Note that compression can be used only when the dataset is
 * chunked.
 */
dims[0] = 1000;
dims[1] = 20;
cdims[0] = 20;
cdims[1] = 20;
dataspace = H5Screate_simple(RANK, dims, NULL);
plist     = H5Pcreate(H5P_DATASET_CREATE);
            H5Pset_chunk(plist, 2, cdims);
            H5Pset_deflate( plist, 6);
```

```
dataset = H5Dcreate(file, "/Data/Compressed_Data", H5T_NATIVE_INT,
                    dataspace, plist);
```

A relative dataset name may also be used when a dataset is created. First obtain the identifier of the group in which the dataset is to be created. Then create the dataset with `H5Dcreate` as illustrated in the following example:

```
/*
 * Open the group.
 */
grp = H5Gopen(file, "Data");

/*
 * Create the dataset "Compressed_Data" in the "Data" group
 * by providing a group identifier and a relative dataset
 * name as parameters to the H5Dcreate function.
 */
dataset = H5Dcreate(grp, "Compressed_Data", H5T_NATIVE_INT,
                    dataspace, plist);
```

Accessing an object in a group. Any object in a group can be accessed by its absolute or relative name. The following lines of code show how to use the absolute name to access the dataset `Compressed_Data` in the group `Data` created in the examples above:

```
/*
 * Open the dataset "Compressed_Data" in the "Data" group.
 */
dataset = H5Dopen(file, "/Data/Compressed_Data");
```

The same dataset can be accessed in another manner. First access the group to which the dataset belongs, then open the dataset.

```
/*
 * Open the group "data" in the file.
 */
grp  = H5Gopen(file, "Data");

/*
 * Access the "Compressed_Data" dataset in the group.
 */
dataset = H5Dopen(grp, "Compressed_Data");
```

Example 7 shows how to create a group in a file and a dataset in a group. It uses the iterator function `H5Giterate` to find the names of the objects in the root group, and `H5Glink` and `H5Gunlink` to create a new group name and delete the original name.

Working with attributes

Think of an attribute as a small datasets that is attached to a normal dataset or group. The H5A API contains routines for working with attributes. Since attributes share many of the characteristics of datasets, the programming model for working with attributes is analogous in many ways to the model for working with datasets. The primary differences are that an attribute must be attached to a dataset or a group, and subsetting operations cannot be performed on attributes.

To create an attribute belonging to a particular dataset or group**,** first create a dataspace for the attribute with the call to `H5Screate`, then create the attribute using `H5Acreate`. For example, the following code creates an attribute called `Integer_attribute` that is a member of a dataset whose identifier is `dataset`. The attribute identifier is `attr2`. `H5Awrite` then sets the value of the attribute of that of the integer variable `point`. `H5Aclose` then releases the attribute identifier.

```
int point = 1;                         /* Value of the scalar attribute */

/*
 * Create scalar attribute.
 */
aid2  = H5Screate(H5S_SCALAR);
attr2 = H5Acreate(dataset, "Integer attribute", H5T_NATIVE_INT, aid2,
                  H5P_DEFAULT);

/*
 * Write scalar attribute.
 */
ret = H5Awrite(attr2, H5T_NATIVE_INT, &point);

/*
 * Close attribute dataspace.
 */
ret = H5Sclose(aid2);

/*
 * Close attribute.
 */
ret = H5Aclose(attr2);
```

To read a scalar attribute whose name and datatype are known, first open the attribute using `H5Aopen_name`, then use H5Aread to get its value. For example the following reads a scalar attribute called `Integer_attribute` whose datatype is a native integer, and whose parent dataset has the identifier `dataset`.

```
/*
 * Attach to the scalar attribute using attribute name, then read and
 * display its value.
 */
attr = H5Aopen_name(dataset,"Integer attribute");
ret  = H5Aread(attr, H5T_NATIVE_INT, &point_out);
printf("The value of the attribute \"Integer attribute\" is %d \n", point_out);
ret =  H5Aclose(attr);
```

Reading an attribute whose characteristics are not known. It may be necessary to query a file to obtain information about an attribute, namely its name, datatype, rank and dimensions. The following code opens an attribute by its index value using `H5Aopen_index`, then reads in information about its datatype.

```
/*
 * Attach to the string attribute using its index, then read and display the value.
 */
```

```
attr =  H5Aopen_idx(dataset, 2);
atype = H5Tcopy(H5T_C_S1);
        H5Tset_size(atype, 4);
ret   = H5Aread(attr, atype, string_out);
printf("The value of the attribute with the index 2 is %s \n", string_out);
```

In practice, if the characteristics of attributes are not known, the code involved in accessing and processing the attribute can be quite complex. For this reason, HDF5 includes a function called `H5Aiterate`, which applies a user-supplied function to each of a set of attributes. The user-supplied function can contain the code that interprets, accesses and processes each attribute.

Example 8 illustrates the use of the `H5Aiterate` function, as well as the other attribute examples described above.

Working with references to objects

In HDF5, objects (i.e. groups, datasets, and named datatypes) are usually accessed by name. This access method was discussed in previous sections. There is another way to access stored objects -- by reference.

An object reference is based on the relative file address of the object header in the file and is constant for the life of the object. Once a reference to an object is created and stored in a dataset in the file, it can be used to dereference the object it points to. References are handy for creating a file index or for grouping related objects by storing references to them in one dataset.

Creating and Storing References to Objects

The following steps are involved in creating and storing file references to objects:

1. Create the objects or open them if they already exist in the file.
2. Create a dataset to store the objects' references.
3. Create and store references to the objects in a buffer.
4. Write a buffer with the references to the dataset.

Programming Example

Description: The example below [also Example 9] creates a group and two datasets and a named datatype in the group. References to these four objects are stored in the dataset in the root group.

```
#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void) {
    hid_t                 fid1;             /* HDF5 File IDs           */
    hid_t                 dataset; /* Dataset ID                       */
    hid_t                 group;      /* Group ID             */
    hid_t                 sid1;       /* Dataspace ID                          */
    hid_t                 tid1;       /* Datatype ID                   */
    hsize_t               dims1[] = {SPACE1_DIM1};
    hobj_ref_t      *wbuf;      /* buffer to write to disk */
    int       *tu32;      /* Temporary pointer to int data */
    int        i;          /* counting variables */
    const char *write_comment="Foo!"; /* Comments for group */
    herr_t                ret;              /* Generic return value            */

/* Compound datatype */
typedef struct s1_t {
    unsigned int a;
    unsigned int b;
    float c;
} s1_t;

    /* Allocate write buffers */
    wbuf=(hobj_ref_t *)malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32=malloc(sizeof(int)*SPACE1_DIM1);

    /* Create file */
    fid1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a group */
    group=H5Gcreate(fid1,"Group1",-1);

    /* Set group’s comment */
    ret=H5Gset_comment(group,".",write_comment);

    /* Create a dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset1",H5T_STD_U32LE,sid1,H5P_DEFAULT);

    for(i=0; i < SPACE1_DIM1; i++)
        tu32[i] = i*3;

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create another dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset2",H5T_NATIVE_UCHAR,sid1,H5P_DEFAULT);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create a datatype to refer to */
    tid1 = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));

    /* Insert fields */
    ret=H5Tinsert (tid1, "a", HOFFSET(s1_t,a), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "b", HOFFSET(s1_t,b), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "c", HOFFSET(s1_t,c), H5T_NATIVE_FLOAT);

    /* Save datatype for later */
    ret=H5Tcommit (group, "Datatype1", tid1);

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close group */
    ret = H5Gclose(group);

    /* Create a dataset to store references */
    dataset=H5Dcreate(fid1,"Dataset3",H5T_STD_REF_OBJ,sid1,H5P_DEFAULT);

    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[0],fid1,"/Group1/Dataset1",H5R_OBJECT,-1);
    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[1],fid1,"/Group1/Dataset2",H5R_OBJECT,-1);

    /* Create reference to group */
    ret = H5Rcreate(&wbuf[2],fid1,"/Group1",H5R_OBJECT,-1);

    /* Create reference to named datatype */
    ret = H5Rcreate(&wbuf[3],fid1,"/Group1/Datatype1",H5R_OBJECT,-1);

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close disk dataspace */
    ret = H5Sclose(sid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Close file */
    ret = H5Fclose(fid1);
    free(wbuf);
    free(tu32);
    return 0;
}
```

Remarks:

- The following code,

  ```
  dataset = H5Dcreate ( fid1,"Dataset3",H5T_STD_REF_OBJ,sid1,H5P_DEFAULT );
  ```

  creates a dataset to store references. Notice that the `H5T_SDT_REF_OBJ` datatype is used to specify that references to objects will be stored. The datatype `H5T_STD_REF_DSETREG` is used to store the dataset region references and is be discussed later.

- The next few calls to the `H5Rcreate` function create references to the objects and store them in the buffer *wbuf*. The signature of the `H5Rcreate` function is:

  ```
  herr_t H5Rcreate ( void* buf, hid_t loc_id, const char *name,
                     H5R_type_t ref_type, hid_t space_id )
  ```

  - The first argument specifies the buffer to store the reference.
  - The second and third arguments specify the name of the referenced object. In the example, the file identifier *fid1* and absolute name of the dataset `/Group1/Dataset1` identify the dataset. One could also use the group identifier of group `Group1` and the relative name of the dataset `Dataset1` to create the same reference.
  - The fourth argument specifies the type of the reference. The example uses references to the objects (`H5R_OBJECT`). Another type of reference, reference to the dataset region (`H5R_DATASET_REGION`), is discussed later.
  - The fifth argument specifies the space identifier. When references to the objects are created, it should be set to `-1`.

- The `H5Dwrite` function writes a dataset with the references to the file. Notice that the `H5T_SDT_REF_OBJ` datatype is used to describe the dataset’s memory datatype.

File Contents: The contents of the `trefer1.h5` file created by this example are as follows:

```
HDF5 "trefer1.h5" {
GROUP "/" {
   DATASET "Dataset3" {
      DATATYPE { H5T_REFERENCE }
      DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
      DATA {
         DATASET 0:1696, DATASET 0:2152, GROUP 0:1320, DATATYPE 0:2268
      }
   }
   GROUP "Group1" {
      DATASET "Dataset1" {
         DATATYPE { H5T_STD_U32LE }
         DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
         DATA {
            0, 3, 6, 9
         }
      }
      DATASET "Dataset2" {
         DATATYPE { H5T_STD_U8LE }
         DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
         DATA {
            0, 0, 0, 0
         }
      }
      DATATYPE "Datatype1" {
         H5T_STD_I32BE "a";
         H5T_STD_I32BE "b";
         H5T_IEEE_F32BE "c";
      }
   }
}
}
```

Notice how the data in dataset `Dataset3` is described. The two numbers with the colon in between represent a unique identifier of the object. These numbers are constant for the life of the object.

Reading References and Accessing Objects Using References

The following steps are involved:

1. Open the dataset with the references and read them. The `H5T_STD_REF_OBJ` datatype must be used to describe the memory datatype.

2. Use the read reference to obtain the identifier of the object the reference points to.

3. Open the dereferenced object and perform the desired operations.

4. Close all objects when the task is complete.

Programming Example

Description: The following example [also Example 10] below opens and reads dataset `Dataset3` from the file created previously. Then the program dereferences the references to dataset `Dataset1`, the group and the named datatype, and opens those objects. The program reads and displays the dataset's data, the group's comment, and the number of members of the compound datatype.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

int
main(void)
{
    hid_t                 fid1;             /* HDF5 File IDs           */
    hid_t                 dataset, /* Dataset ID                       */
                dset2;      /* Dereferenced dataset ID */
    hid_t                 group;      /* Group ID            */
    hid_t                 sid1;       /* Dataspace ID                       */
    hid_t                 tid1;       /* Datatype ID                   */
    hobj_ref_t      *rbuf;      /* buffer to read from disk */
    int                *tu32;      /* temp. buffer read from disk */
    int        i;          /* counting variables */
    char read_comment[10];
    herr_t                ret;             /* Generic return value            */

    /* Allocate read buffers */
    rbuf = malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32 = malloc(sizeof(int)*SPACE1_DIM1);

    /* Open the file */
    fid1 = H5Fopen(FILE1, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dataset=H5Dopen(fid1,"/Dataset3");

    /* Read selection from disk */
    ret=H5Dread(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);
    /* Open dataset object */
    dset2 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);

    ret=H5Sget_simple_extent_npoints(sid1);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);
    printf("Dataset data : \n");
     for (i=0; i < SPACE1_DIM1 ; i++) printf (" %d ", tu32[i]);
    printf("\n");
    printf("\n");

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Open group object */
    group = H5Rdereference(dataset,H5R_OBJECT,&rbuf[2]);

    /* Get group’s comment */
    ret=H5Gget_comment(group,".",10,read_comment);
    printf("Group comment is %s \n", read_comment);
    printf(" \n");
    /* Close group */
    ret = H5Gclose(group);

    /* Open datatype object */
    tid1 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[3]);

    /* Verify correct datatype */
    {
        H5T_class_t tclass;

        tclass= H5Tget_class(tid1);
        if ((tclass == H5T_COMPOUND))
           printf ("Number of compound datatype members is %d \n", H5Tget_nmembers(tid1));
    printf(" \n");
    }

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Close file */
    ret = H5Fclose(fid1);

    /* Free memory buffers */
    free(rbuf);
    free(tu32);
    return 0;
}
```

The output of this program is as follows:

```
Dataset data :
 0  3  6  9

Group comment is Foo!

Number of compound datatype members is 3
```

Remarks:

- The `H5Dread` function was used to read dataset `Dataset3` containing the references to the objects. The `H5T_STD_REF_OBJ` memory datatype was used to read references to memory.
- `H5Rdereference` obtains the object's identifier. The signature of this function is:

  ```
  hid_t H5Rdereference (hid_t datatset, H5R_type_t ref_type, void *ref)
  ```

  - The first argument is an identifier of the dataset with the references.
  - The second argument specifies the reference type. `H5R_OBJECT` was used to specify a reference to an object. Another type, used to specifiy a reference to a dataset region and discussed later, is `H5R_DATASET_REGION`.
  - The third argument is a buffer to store the reference to be read.
  - The function returns an identifier of the object the reference points to. In this simplified situation, the type that was stored in the dataset is known. When the type of the object is unknown, `H5Rget_object_type` should be used to identify the type of object the reference points to.

Working with references to dataset regions

A dataset region reference points to the dataset selection by storing the relative file address of the dataset header and the global heap offset of the referenced selection. The selection referenced is located by retrieving the coordinates of the areas in the selection from the global heap. This internal mechanism of storing and retrieving dataset selections is transparent to the user. A reference to the dataset selection (region) is constant for the life of the dataset.

Creating and Storing References to Dataset Regions

The following steps are involved in creating and storing references to the dataset regions:

1. Create a dataset to store the dataset regions (selections).
2. Create selections in the dataset(s). Dataset(s) should already exist in the file.
3. Create references to the selections and store them in a buffer.
4. Write references to the dataset regions in the file.
5. Close all objects.

Programming Example

Description: The example below [also Example 11] creates a dataset in the file. Then it creates a dataset to store references to the dataset regions (selections). The first selection is a 6 x 6 hyperslab. The second selection is a point selection in the same dataset. References to both selections are created and stored in the buffer, and then written to the dataset in the file.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK     1
#define SPACE1_DIM1     4

/* Dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

/* Element selection information */
#define POINT1_NPOINTS 10

int
main(void)
{
    hid_t                fid1;            /* HDF5 File IDs          */
    hid_t                dset1,   /* Dataset ID                     */
                dset2;      /* Dereferenced dataset ID */
    hid_t                sid1,       /* Dataspace ID        #1               */
                sid2;       /* Dataspace ID  #2                */
    hsize_t              dims1[] = {SPACE1_DIM1},
                 dims2[] = {SPACE2_DIM1, SPACE2_DIM2};
    hssize_t      start[SPACE2_RANK];     /* Starting location of hyperslab */
    hsize_t              stride[SPACE2_RANK];    /* Stride of hyperslab */
    hsize_t              count[SPACE2_RANK];     /* Element count of hyperslab */
    hsize_t                block[SPACE2_RANK];     /* Block size of hyperslab */
    hssize_t      coord1[POINT1_NPOINTS][SPACE2_RANK];
                                    /* Coordinates for point selection */
    hdset_reg_ref_t      *wbuf;      /* buffer to write to disk */
    int     *dwbuf;      /* Buffer for writing numeric data to disk */
    int        i;          /* counting variables */
    herr_t                 ret;             /* Generic return value             */


    /* Allocate write & read buffers */
    wbuf=calloc(sizeof(hdset_reg_ref_t), SPACE1_DIM1);
    dwbuf=malloc(sizeof(int)*SPACE2_DIM1*SPACE2_DIM2);

    /* Create file */
    fid1 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid2 = H5Screate_simple(SPACE2_RANK, dims2, NULL);

    /* Create a dataset */
    dset2=H5Dcreate(fid1,"Dataset2",H5T_STD_U8LE,sid2,H5P_DEFAULT);

    for(i=0; i < SPACE2_DIM1*SPACE2_DIM2; i++)
        dwbuf[i]=i*3;

    /* Write selection to disk */
    ret=H5Dwrite(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,dwbuf);

    /* Close Dataset */
    ret = H5Dclose(dset2);

    /* Create dataspace for the reference dataset */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a dataset */
    dset1=H5Dcreate(fid1,"Dataset1",H5T_STD_REF_DSETREG,sid1,H5P_DEFAULT);

    /* Create references */

    /* Select 6x6 hyperslab for first reference */
    start[0]=2; start[1]=2;
    stride[0]=1; stride[1]=1;
    count[0]=6; count[1]=6;
    block[0]=1; block[1]=1;
    ret = H5Sselect_hyperslab(sid2,H5S_SELECT_SET,start,stride,count,block);

    /* Store first dataset region */
    ret = H5Rcreate(&wbuf[0],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Select sequence of ten points for second reference */
    coord1[0][0]=6; coord1[0][1]=9;
    coord1[1][0]=2; coord1[1][1]=2;
    coord1[2][0]=8; coord1[2][1]=4;
    coord1[3][0]=1; coord1[3][1]=6;
    coord1[4][0]=2; coord1[4][1]=8;
    coord1[5][0]=3; coord1[5][1]=2;
    coord1[6][0]=0; coord1[6][1]=4;
    coord1[7][0]=9; coord1[7][1]=0;
    coord1[8][0]=7; coord1[8][1]=1;
    coord1[9][0]=3; coord1[9][1]=3;
    ret = H5Sselect_elements(sid2,H5S_SELECT_SET,POINT1_NPOINTS,(const hssize_t
**)coord1);
```

```
    /* Store second dataset region */
    ret = H5Rcreate(&wbuf[1],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Write selection to disk */
    ret=H5Dwrite(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close all objects */
    ret = H5Sclose(sid1);
    ret = H5Dclose(dset1);
    ret = H5Sclose(sid2);

    /* Close file */
    ret = H5Fclose(fid1);

    free(wbuf);
    free(dwbuf);
    return 0;
}
```

Remarks:

- The code,

  ```
  dset1=H5Dcreate(fid1,"Dataset1",H5T_STD_REF_DSETREG,sid1,H5P_DEFAULT);
  ```

  creates a dataset to store references to the dataset(s) regions (selections). Notice that the `H5T_STD_REF_DSETREG` datatype is used.

- This program uses hyperslab and point selections. The dataspace handle *sid2* is used for the calls to `H5Sselect_hyperslab` and `H5Sselect_elements`. The handle was created when dataset **`Dataset2`** was created and it describes the dataset's dataspace. It was not closed when the dataset was closed to decrease the number of function calls used in the example. In a real application program, one should open the dataset and determine its dataspace using the `H5Dget_space` function.

- `H5Rcreate` is used to create a dataset region reference and store it in a buffer. The signature of the function is:

  ```
  herr_t H5Rcreate(void *buf, hid_t loc_id, const char *name,
                   H5R_type_t ref_type, hid_t space_id)
  ```

  - The first argument specifies the buffer to store the reference.

  - The second and third arguments specify the name of the referenced dataset. In the example, the file identifier *fid1* and the absolute name of the dataset **`/Dataset2`** were used to identify the dataset. The reference to the region of this dataset is stored in the buffer *buf*.

  - The fourth argument specifies the type of the reference. Since the example creates references to the dataset regions, the `H5R_DATASET_REGION` datatype is used.

  - The fifth argument is a dataspace identifier of the referenced dataset.

File Contents: The contents of the file `trefer2.h5` created by this program are as follows:

```
HDF5 "trefer2.h5" {
GROUP "/" {
   DATASET "Dataset1" {
      DATATYPE { H5T_REFERENCE }
      DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
      DATA {
         DATASET 0:744 {(2,2)-(7,7)}, DATASET 0:744 {(6,9), (2,2), (8,4), (1,6),
          (2,8), (3,2), (0,4), (9,0), (7,1), (3,3)}, NULL, NULL
      }
   }
   DATASET "Dataset2" {
      DATATYPE { H5T_STD_U8LE }
      DATASPACE { SIMPLE ( 10, 10 ) / ( 10, 10 ) }
      DATA {
         0, 3, 6, 9, 12, 15, 18, 21, 24, 27,
         30, 33, 36, 39, 42, 45, 48, 51, 54, 57,
         60, 63, 66, 69, 72, 75, 78, 81, 84, 87,
         90, 93, 96, 99, 102, 105, 108, 111, 114, 117,
         120, 123, 126, 129, 132, 135, 138, 141, 144, 147,
         150, 153, 156, 159, 162, 165, 168, 171, 174, 177,
         180, 183, 186, 189, 192, 195, 198, 201, 204, 207,
         210, 213, 216, 219, 222, 225, 228, 231, 234, 237,
         240, 243, 246, 249, 252, 255, 255, 255, 255, 255,
         255, 255, 255, 255, 255, 255, 255, 255, 255, 255
      }
   }
}
}
```

Notice how raw data of the dataset with the dataset regions is displayed. Each element of the raw data consists of a reference to the dataset (`DATASET number1:number2`) and its selected region. If the selection is a hyperslab, the corner coordinates of the hyperslab are displayed. For the point selection, the coordinates of each point are displayed. Since only two selections were stored, the third and fourth elements of the dataset `Dataset1` are set to `NULL`. This was done by the buffer inizialization in the program.

Reading references to dataset regions

The following steps are involved in reading references to dataset regions and referenced dataset regions (selections).

1. Open and read the dataset containing references to the dataset regions. The datatype `H5T_STD_REF_DSETREG` must be used during read operation.

2. Use `H5Rdereference` to obtain the dataset identifier from the read dataset region reference.

   or

   Use `H5Rget_region` to obtain the dataspace identifier for the dataset containing the selection from the read dataset region reference.

3. With the dataspace identifier, the H5S interface functions, `H5Sget_select_*`, can be used to obtain information about the selection.

4. Close all objects when they are no longer needed.

Programming Example

Description: The following example [also Example 12] reads a dataset containing dataset region references. It reads data from the dereferenced dataset and displays the number of elements and raw data. Then it reads two selections: a hyperslab selection and a point selection. The program queries a number of points in the hyperslab and the coordinates and displays them. Then it queries a number of selected points and their coordinates and displays the information.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define NPOINTS 10

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void)
{
    hid_t                 fid1;            /* HDF5 File IDs           */
    hid_t                 dset1,   /* Dataset ID                      */
            dset2;      /* Dereferenced dataset ID */
    hid_t                 sid1,      /* Dataspace ID        #1                */
            sid2;       /* Dataspace ID  #2             */
    hsize_t *   coords;             /* Coordinate buffer */
    hsize_t               low[SPACE2_RANK];   /* Selection bounds */
    hsize_t               high[SPACE2_RANK];     /* Selection bounds */
    hdset_reg_ref_t      *rbuf;      /* buffer to to read disk */
    int    *drbuf;      /* Buffer for reading numeric data from disk */
    int        i, j;          /* counting variables */
    herr_t                ret;             /* Generic return value            */

    /* Output message about test being performed */

    /* Allocate write & read buffers */
    rbuf=malloc(sizeof(hdset_reg_ref_t)*SPACE1_DIM1);
    drbuf=calloc(sizeof(int),SPACE2_DIM1*SPACE2_DIM2);

    /* Open the file */
    fid1 = H5Fopen(FILE2, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dset1=H5Dopen(fid1,"/Dataset1");

    /* Read selection from disk */
    ret=H5Dread(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);

    /* Try to open objects */
    dset2 = H5Rdereference(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);
    ret=H5Sget_simple_extent_npoints(sid1);
    printf(" Number of elements in the dataset is : %d\n",ret);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,drbuf);

    for(i=0; i < SPACE2_DIM1; i++) {
        for (j=0; j < SPACE2_DIM2; j++) printf (" %d ", drbuf[i*SPACE2_DIM2+j]);
        printf("\n"); }

    /* Get the hyperslab selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Verify correct hyperslab selected */
    ret = H5Sget_select_npoints(sid2);
    printf(" Number of elements in the hyperslab is : %d \n", ret);
    ret = H5Sget_select_hyper_nblocks(sid2);
    coords=malloc(ret*SPACE2_RANK*sizeof(hsize_t)*2); /* allocate space for the hyperslab
blocks */
    ret = H5Sget_select_hyper_blocklist(sid2,0,ret,coords);
    printf(" Hyperslab coordinates are : \n");
    printf (" ( %lu , %lu ) ( %lu , %lu ) \n", \
(unsigned long)coords[0],(unsigned long)coords[1],(unsigned long)coords[2],(unsigned
long)coords[3]);
    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Get the element selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[1]);

    /* Verify correct elements selected */
    ret = H5Sget_select_elem_npoints(sid2);
    printf(" Number of selected elements is : %d\n", ret);

    /* Allocate space for the element points */
    coords= malloc(ret*SPACE2_RANK*sizeof(hsize_t));
    ret = H5Sget_select_elem_pointlist(sid2,0,ret,coords);
    printf(" Coordinates of selected elements are : \n");
    for (i=0; i < 2*NPOINTS; i=i+2)
         printf(" ( %lu , %lu ) \n", (unsigned long)coords[i],(unsigned long)coords[i+1]);

    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Close first space */
    ret = H5Sclose(sid1);

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Close Dataset */
    ret = H5Dclose(dset1);

    /* Close file */
    ret = H5Fclose(fid1);
    /* Free memory buffers */
    free(rbuf);
    free(drbuf);
    return 0;
}
```

The output of this program is :

```
Number of elements in the dataset is : 100
0  3  6  9  12  15  18  21  24  27
30  33  36  39  42  45  48  51  54  57
60  63  66  69  72  75  78  81  84  87
90  93  96  99  102  105  108  111  114  117
120  123  126  129  132  135  138  141  144  147
150  153  156  159  162  165  168  171  174  177
180  183  186  189  192  195  198  201  204  207
210  213  216  219  222  225  228  231  234  237
240  243  246  249  252  255  255  255  255  255
255  255  255  255  255  255  255  255  255  255
Number of elements in the hyperslab is : 36
Hyperslab coordinates are :
( 2 , 2 ) ( 7 , 7 )
Number of selected elements is : 10
Coordinates of selected elements are :
( 6 , 9 )
( 2 , 2 )
( 8 , 4 )
( 1 , 6 )
( 2 , 8 )
( 3 , 2 )
( 0 , 4 )
( 9 , 0 )
( 7 , 1 )
( 3 , 3 )
```

Remarks:

- The dataset with the region references was read by `H5Dread` with the `H5T_STD_REF_DSETREG` datatype specified.
- The read reference can be used to obtain the dataset identifier with the following call:

  ```
  dset2 = H5Rdereference (dset1,H5R_DATASET_REGION,&rbuf[0]);
  ```

  or to obtain spacial information (dataspace and selection) with the call to `H5Rget_region`:

  ```
  sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[0]);
  ```

  The reference to the dataset region has information for both the dataset itself and its selection. In both functions:

  - The first parameter is an identifier of the dataset with the region references.
  - The second parameter specifies the type of reference stored. In this example, a reference to the dataset region is stored.
  - The third parameter is a buffer containing the reference of the specified type.
- This example introduces several `H5Sget_select*` functions used to obtain information about selections:

`H5Sget_select_npoints:` returns the number of elements in the hyperslab
`H5Sget_select_hyper_nblocks:` returns the number of blocks in the hyperslab
`H5Sget_select_blocklist:` returns the "lower left" and "upper right" coordinates of the blocks in the hyperslab selection
`H5Sget_select_bounds:` returns the coordinates of the "minimal" block containing a hyperslab selection
`H5Sget_select_elem_npoints:` returns the number of points in the element selection
`H5Sget_select_elem_points:` returns the coordinates of the element selection

4. Example Codes

Example 1: How to create a homogeneous multi-dimensional dataset and write it to a file.

This example creates a 2-dimensional HDF 5 dataset of little endian 32-bit integers.

```
/*
 *  This example writes data to the HDF5 file.
 *  Data conversion is performed during write operation.
 */

#include

#define FILE        "SDS.h5"
#define DATASETNAME "IntArray"
#define NX     5                      /* dataset dimensions */
#define NY     6
#define RANK   2

int
main (void)
{
    hid_t       file, dataset;         /* file and dataset handles */
    hid_t       datatype, dataspace;   /* handles */
    hsize_t     dimsf[2];              /* dataset dimensions */
    herr_t      status;
    int         data[NX][NY];          /* data to write */
    int         i, j;

    /*
     * Data  and output buffer initialization.
     */
    for (j = 0; j < NX; j++) {
      for (i = 0; i < NY; i++)
          data[j][i] = i + j;
    }
    /*
     * 0 1 2 3 4 5
     * 1 2 3 4 5 6
     * 2 3 4 5 6 7
     * 3 4 5 6 7 8
     * 4 5 6 7 8 9
     */

    /*
     * Create a new file using H5F_ACC_TRUNC access,
     * default file creation properties, and default file
     * access properties.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Describe the size of the array and create the data space for fixed
     * size dataset.
     */
    dimsf[0] = NX;
    dimsf[1] = NY;
    dataspace = H5Screate_simple(RANK, dimsf, NULL);

    /*
     * Define datatype for the data in the file.
     * We will store little endian INT numbers.
     */
    datatype = H5Tcopy(H5T_NATIVE_INT);
    status = H5Tset_order(datatype, H5T_ORDER_LE);

    /*
     * Create a new dataset within the file using defined dataspace and
     * datatype and default dataset creation properties.
     */
    dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace,
                        H5P_DEFAULT);

    /*
     * Write the data to the dataset using default transfer properties.
     */
    status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                      H5P_DEFAULT, data);

    /*
     * Close/release resources.
     */
    H5Sclose(dataspace);
    H5Tclose(datatype);
    H5Dclose(dataset);
    H5Fclose(file);

    return 0;
}
```

Example 2. How to read a hyperslab from file into memory.

This example reads a hyperslab from a 2-d HDF5 dataset into a 3-d dataset in memory.

```
/*
 *   This example reads hyperslab from the SDS.h5 file
 *   created by h5_write.c program into two-dimensional
 *   plane of the three-dimensional array.
 *   Information about dataset in the SDS.h5 file is obtained.
 */

#include "hdf5.h"

#define FILE        "SDS.h5"
#define DATASETNAME "IntArray"
#define NX_SUB  3           /* hyperslab dimensions */
#define NY_SUB  4
#define NX 7           /* output buffer dimensions */
#define NY 7
#define NZ  3
#define RANK         2
#define RANK_OUT     3

int
main (void)
{
    hid_t       file, dataset;         /* handles */
    hid_t       datatype, dataspace;
    hid_t       memspace;
    H5T_class_t class;                 /* datatype class */
    H5T_order_t order;                 /* data order */
    size_t      size;                  /*
                                   * size of the data element
                                   * stored in file
                                   */
    hsize_t     dimsm[3];              /* memory space dimensions */
    hsize_t     dims_out[2];           /* dataset dimensions */
    herr_t      status;

    int         data_out[NX][NY][NZ ]; /* output buffer */

    hsize_t      count[2];              /* size of the hyperslab in the file */
    hssize_t     offset[2];             /* hyperslab offset in the file */
    hsize_t      count_out[3];          /* size of the hyperslab in memory */
    hssize_t     offset_out[3];         /* hyperslab offset in memory */
    int          i, j, k, status_n, rank;

    for (j = 0; j < NX; j++) {
       for (i = 0; i < NY; i++) {
           for (k = 0; k < NX; j++) {
       for (i = 0; i < NY; i++) printf("%d ", data_out[j][i][0]);
       printf("\n");
    }
    /*
     * 0 0 0 0 0 0 0
     * 0 0 0 0 0 0 0
     * 0 0 0 0 0 0 0
     * 3 4 5 6 0 0 0
     * 4 5 6 7 0 0 0
     * 5 6 7 8 0 0 0
     * 0 0 0 0 0 0 0
     */

    /*
     * Close/release resources.
     */
    H5Tclose(datatype);
    H5Dclose(dataset);
    H5Sclose(dataspace);
    H5Sclose(memspace);
    H5Fclose(file);

    return 0;
}
```

Example 3. Writing selected data from memory to a file.

This example shows how to use the selection capabilities of HDF5 to write selected data to a file. It includes the examples discussed in the text.

```
/*
 *  This program shows how the H5Sselect_hyperslab and H5Sselect_elements
 *  functions are used to write selected data from memory to the file.
 *  Program takes 48 elements from the linear buffer and writes them into
 *  the matrix using 3x2 blocks, (4,3) stride and (2,4) count.
 *  Then four elements  of the matrix are overwritten with the new values and
 *  file is closed. Program reopens the file and reads and displays the result.
 */

#include

#define FILE "Select.h5"

#define MSPACE1_RANK     1          /* Rank of the first dataset in memory */
#define MSPACE1_DIM      50         /* Dataset size in memory */

#define MSPACE2_RANK     1          /* Rank of the second dataset in memory */
#define MSPACE2_DIM      4          /* Dataset size in memory */

#define FSPACE_RANK      2          /* Dataset rank as it is stored in the file */
#define FSPACE_DIM1      8          /* Dimension sizes of the dataset as it is
                                       stored in the file */
#define FSPACE_DIM2      12

                                    /* We will read dataset back from the file
                                       to the dataset in memory with these
                                       dataspace parameters. */
#define MSPACE_RANK      2
#define MSPACE_DIM1      8
#define MSPACE_DIM2      12

#define NPOINTS          4          /* Number of points that will be selected
                                       and overwritten */
int main (void)
{

   hid_t   file, dataset;           /* File and dataset identifiers */
   hid_t   mid1, mid2, fid;         /* Dataspace identifiers */
   hsize_t dim1[] = {MSPACE1_DIM};  /* Dimension size of the first dataset
                                       (in memory) */
   hsize_t dim2[] = {MSPACE2_DIM};  /* Dimension size of the second dataset
                                       (in memory */
   hsize_t fdim[] = {FSPACE_DIM1, FSPACE_DIM2};
                                    /* Dimension sizes of the dataset (on disk) */

   hssize_t start[2]; /* Start of hyperslab */
   hsize_t stride[2]; /* Stride of hyperslab */
   hsize_t count[2];  /* Block count */
   hsize_t block[2];  /* Block sizes */

   hssize_t coord[NPOINTS][FSPACE_RANK]; /* Array to store selected points
                                            from the file dataspace */
   herr_t  ret;
   uint    i,j;
   int     matrix[MSPACE_DIM1][MSPACE_DIM2];
   int     vector[MSPACE1_DIM];
   int     values[] = {53, 59, 61, 67};  /* New values to be written */

   /*
    * Buffers' initialization.
    */
   vector[0] = vector[MSPACE1_DIM - 1] = -1;
   for (i = 1; i < MSPACE_DIM1; i++) {
       for (j = 0; j < MSPACE_DIM2; j++)
       matrix[i][j] = 0;
    }

    /*
     * Create a file.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create dataspace for the dataset in the file.
     */
    fid = H5Screate_simple(FSPACE_RANK, fdim, NULL);

    /*
     * Create dataset and write it into the file.
     */
    dataset = H5Dcreate(file, "Matrix in file", H5T_NATIVE_INT, fid, H5P_DEFAULT);
    ret = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, matrix);

    /*
     * Select hyperslab for the dataset in the file, using 3x2 blocks,
     * (4,3) stride and (2,4) count starting at the position (0,1).
     */
    start[0]  = 0; start[1]  = 1;
    stride[0] = 4; stride[1] = 3;
    count[0]  = 2; count[1]  = 4;
    block[0]  = 3; block[1]  = 2;
    ret = H5Sselect_hyperslab(fid, H5S_SELECT_SET, start, stride, count, block);

    /*
     * Create dataspace for the first dataset.
     */
    mid1 = H5Screate_simple(MSPACE1_RANK, dim1, NULL);

    /*
     * Select hyperslab.
     * We will use 48 elements of the vector buffer starting at the second element.
     * Selected elements are 1 2 3 . . . 48
     */
    start[0]  = 1;
    stride[0] = 1;
    count[0]  = 48;
    block[0]  = 1;
    ret = H5Sselect_hyperslab(mid1, H5S_SELECT_SET, start, stride, count, block);

    /*
     * Write selection from the vector buffer to the dataset in the file.
     *
     * File dataset should look like this:
     *                   0  1  2  0  3  4  0  5  6  0  7  8
     *                   0  9 10  0 11 12  0 13 14  0 15 16
     *                   0 17 18  0 19 20  0 21 22  0 23 24
     *                   0  0  0  0  0  0  0  0  0  0  0  0
     *                   0 25 26  0 27 28  0 29 30  0 31 32
     *                    0 33 34  0 35 36  0 37 38  0 39 40
     *                    0 41 42  0 43 44  0 45 46  0 47 48
     *                    0  0  0  0  0  0  0  0  0  0  0  0
     */
     ret = H5Dwrite(dataset, H5T_NATIVE_INT, mid1, fid, H5P_DEFAULT, vector);

    /*
     * Reset the selection for the file dataspace fid.
     */
    ret = H5Sselect_none(fid);

    /*
     * Create dataspace for the second dataset.
     */
    mid2 = H5Screate_simple(MSPACE2_RANK, dim2, NULL);

    /*
     * Select sequence of NPOINTS points in the file dataspace.
     */
    coord[0][0] = 0; coord[0][1] = 0;
    coord[1][0] = 3; coord[1][1] = 3;
    coord[2][0] = 3; coord[2][1] = 5;
    coord[3][0] = 5; coord[3][1] = 6;

    ret = H5Sselect_elements(fid, H5S_SELECT_SET, NPOINTS,
                             (const hssize_t **)coord);

    /*
     * Write new selection of points to the dataset.
     */
    ret = H5Dwrite(dataset, H5T_NATIVE_INT, mid2, fid, H5P_DEFAULT, values);

    /*
     * File dataset should look like this:
     *                   53  1  2  0  3  4  0  5  6  0  7  8
     *                    0  9 10  0 11 12  0 13 14  0 15 16
     *                    0 17 18  0 19 20  0 21 22  0 23 24
     *                    0  0  0 59  0 61  0  0  0  0  0  0
     *                    0 25 26  0 27 28  0 29 30  0 31 32
     *                    0 33 34  0 35 36 67 37 38  0 39 40
     *                    0 41 42  0 43 44  0 45 46  0 47 48
     *                    0  0  0  0  0  0  0  0  0  0  0  0
     *
     */

    /*
     * Close memory file and memory dataspaces.
     */
    ret = H5Sclose(mid1);
    ret = H5Sclose(mid2);
    ret = H5Sclose(fid);

    /*
     * Close dataset.
     */
    ret = H5Dclose(dataset);

    /*
     * Close the file.
     */
    ret = H5Fclose(file);

    /*
```

```
     * Open the file.
     */
    file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);

    /*
     * Open the dataset.
     */
    dataset = dataset = H5Dopen(file,"Matrix in file");

    /*
     * Read data back to the buffer matrix.
     */
    ret = H5Dread(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                  H5P_DEFAULT, matrix);

    /*
     * Display the result.
     */
    for (i=0; i < MSPACE_DIM1; i++) {
        for(j=0; j < MSPACE_DIM2; j++) printf("%3d  ", matrix[i][j]);
        printf("\n");
    }

    return 0;
}
```

Example 4. Working with compound datatypes.

This example shows how to create a compound datatype, write an array which has the compound datatype to the file, and read back subsets of fields.

```
/*
 * This example shows how to create a compound datatype,
 * write an array which has the compound datatype to the file,
 * and read back fields’ subsets.
 */

#include "hdf5.h"

#define FILE          "SDScompound.h5"
#define DATASETNAME   "ArrayOfStructures"
#define LENGTH        10
#define RANK          1

int
main(void)
{

    /* First structure  and dataset*/
    typedef struct s1_t {
       int    a;
       float  b;
       double c;
    } s1_t;
    s1_t       s1[LENGTH];
    hid_t      s1_tid;     /* File datatype identifier */

    /* Second structure (subset of s1_t)  and dataset*/
    typedef struct s2_t {
       double c;
       int    a;
    } s2_t;
    s2_t       s2[LENGTH];
    hid_t      s2_tid;    /* Memory datatype handle */

    /* Third "structure" ( will be used to read float field of s1) */
    hid_t      s3_tid;   /* Memory datatype handle */
    float      s3[LENGTH];

    int        i;
    hid_t      file, dataset, space; /* Handles */
    herr_t     status;
    hsize_t    dim[] = {LENGTH};   /* Dataspace dimensions */


    /*
     * Initialize the data
     */
    for (i = 0; i< LENGTH; i++) {
        s1[i].a = i;
        s1[i].b = i*i;
        s1[i].c = 1./(i+1);
    }

    /*
     * Create the data space.
     */
    space = H5Screate_simple(RANK, dim, NULL);

    /*
     * Create the file.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create the memory datatype.
     */
    s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
    H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
    H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
    H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_FLOAT);

    /*
     * Create the dataset.
     */
    dataset = H5Dcreate(file, DATASETNAME, s1_tid, space, H5P_DEFAULT);

    /*
     * Wtite data to the dataset;
     */
    status = H5Dwrite(dataset, s1_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s1);

    /*
     * Release resources
     */
    H5Tclose(s1_tid);
    H5Sclose(space);
    H5Dclose(dataset);
    H5Fclose(file);

    /*
     * Open the file and the dataset.
     */
    file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);

    dataset = H5Dopen(file, DATASETNAME);

    /*
     * Create a datatype for s2
     */
    s2_tid = H5Tcreate(H5T_COMPOUND, sizeof(s2_t));

    H5Tinsert(s2_tid, "c_name", HOFFSET(s2_t, c), H5T_NATIVE_DOUBLE);
    H5Tinsert(s2_tid, "a_name", HOFFSET(s2_t, a), H5T_NATIVE_INT);

    /*
     * Read two fields c and a from s1 dataset. Fields in the file
     * are found by their names "c_name" and "a_name".
     */
    status = H5Dread(dataset, s2_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s2);

    /*
     * Display the fields
     */
    printf("\n");
    printf("Field c : \n");
    for( i = 0; i < LENGTH; i++) printf("%.4f ", s2[i].c);
    printf("\n");
    printf("\n");
    printf("Field a : \n");
    for( i = 0; i < LENGTH; i++) printf("%d ", s2[i].a);
    printf("\n");

    /*
     * Create a datatype for s3.
     */
    s3_tid = H5Tcreate(H5T_COMPOUND, sizeof(float));

    status = H5Tinsert(s3_tid, "b_name", 0, H5T_NATIVE_FLOAT);

    /*
     * Read field b from s1 dataset. Field in the file is found by its name.
     */
    status = H5Dread(dataset, s3_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s3);

    /*
     * Display the field
     */
    printf("\n");
    printf("Field b : \n");
    for( i = 0; i < LENGTH; i++) printf("%.4f ", s3[i]);
    printf("\n");

    /*
     * Release resources
     */
    H5Tclose(s2_tid);
    H5Tclose(s3_tid);
    H5Dclose(dataset);
    H5Fclose(file);

    return 0;
}
```

Example 5. Creating and writing an extendible dataset.

This example shows how to create a 3x3 extendible dataset, to extend the dataset to 10x3, then to extend it again to 10x5.

```
/*
 *   This example shows how to work with extendible dataset.
 *   In the current version of the library dataset MUST be
 *   chunked.
 *
 */

#include "hdf5.h"

#define FILE        "SDSextendible.h5"
#define DATASETNAME "ExtendibleArray"
#define RANK         2
#define NX     10
#define NY     5

int
main (void)
{
    hid_t       file;                          /* handles */
    hid_t       dataspace, dataset;
    hid_t       filespace;
    hid_t       cparms;
    hsize_t      dims[2]  = { 3, 3};            /*
                                             * dataset dimensions
                                             * at the creation time
                                             */
    hsize_t      dims1[2] = { 3, 3};            /* data1 dimensions */
    hsize_t      dims2[2] = { 7, 1};            /* data2 dimensions */
    hsize_t      dims3[2] = { 2, 2};            /* data3 dimensions */

    hsize_t      maxdims[2] = {H5S_UNLIMITED, H5S_UNLIMITED};
    hsize_t      chunk_dims[2] ={2, 5};
    hsize_t      size[2];
    hssize_t     offset[2];

    herr_t      status;

    int         data1[3][3] = { {1, 1, 1},       /* data to write */
                          {1, 1, 1},
                          {1, 1, 1} };

    int         data2[7]    = { 2, 2, 2, 2, 2, 2, 2};

    int         data3[2][2] = { {3, 3},
                          {3, 3} };

    /*
     * Create the data space with unlimited dimensions.
     */
    dataspace = H5Screate_simple(RANK, dims, maxdims);

    /*
     * Create a new file. If file exists its contents will be overwritten.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    /*
     * Modify dataset creation properties, i.e. enable chunking.
     */
    cparms = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_chunk( cparms, RANK, chunk_dims);

    /*
     * Create a new dataset within the file using cparms
     * creation properties.
     */
    dataset = H5Dcreate(file, DATASETNAME, H5T_NATIVE_INT, dataspace,
                 cparms);

    /*
     * Extend the dataset. This call assures that dataset is at least 3 x 3.
     */
    size[0]   = 3;
    size[1]   = 3;
    status = H5Dextend (dataset, size);

    /*
     * Select a hyperslab.
     */
    filespace = H5Dget_space (dataset);
    offset[0] = 0;
    offset[1] = 0;
    status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
                        dims1, NULL);

    /*
     * Write the data to the hyperslab.
     */
    status = H5Dwrite(dataset, H5T_NATIVE_INT, dataspace, filespace,
               H5P_DEFAULT, data1);

    /*
     * Extend the dataset. Dataset becomes 10 x 3.
     */
    dims[0]   = dims1[0] + dims2[0];
    size[0]   = dims[0];
    size[1]   = dims[1];
    status = H5Dextend (dataset, size);

    /*
     * Select a hyperslab.
     */
    filespace = H5Dget_space (dataset);
    offset[0] = 3;
    offset[1] = 0;
    status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
                        dims2, NULL);

    /*
     * Define memory space
     */
    dataspace = H5Screate_simple(RANK, dims2, NULL);

    /*
     * Write the data to the hyperslab.
     */
    status = H5Dwrite(dataset, H5T_NATIVE_INT, dataspace, filespace,
               H5P_DEFAULT, data2);
    /*
     * Extend the dataset. Dataset becomes 10 x 5.
     */
    dims[1]   = dims1[1] + dims3[1];
    size[0]   = dims[0];
    size[1]   = dims[1];
    status = H5Dextend (dataset, size);

    /*
     * Select a hyperslab
     */
    filespace = H5Dget_space (dataset);
    offset[0] = 0;
    offset[1] = 3;
    status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
                                 dims3, NULL);

    /*
     * Define memory space.
     */
    dataspace = H5Screate_simple(RANK, dims3, NULL);

    /*
     * Write the data to the hyperslab.
     */
    status = H5Dwrite(dataset, H5T_NATIVE_INT, dataspace, filespace,
                      H5P_DEFAULT, data3);

    /*
     * Resulting dataset
     *
     *  3 3 3 2 2
     *  3 3 3 2 2
     *  3 3 3 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     *  2 0 0 0 0
     */
    /*
     * Close/release resources.
     */
    H5Dclose(dataset);
    H5Sclose(dataspace);
    H5Sclose(filespace);
    H5Fclose(file);

    return 0;
}
```

Example 6. Reading data.

This example shows how to read information the chunked dataset written by Example 5.

```
/*
 *   This example shows how to read data from a chunked dataset.
 *   We will read from the file created by h5_extend_write.c
 */

#include "hdf5.h"

#define FILE        "SDSextendible.h5"
#define DATASETNAME "ExtendibleArray"
#define RANK         2
#define RANKC        1
#define NX           10
#define NY           5

int
main (void)
{
    hid_t       file;                        /* handles */
    hid_t       dataset;
    hid_t       filespace;
    hid_t       memspace;
    hid_t       cparms;
    hsize_t     dims[2];                     /* dataset and chunk dimensions*/
    hsize_t     chunk_dims[2];
    hsize_t     col_dims[1];
    hsize_t     count[2];
    hssize_t    offset[2];

    herr_t      status, status_n;

    int         data_out[NX][NY];  /* buffer for dataset to be read */
    int         chunk_out[2][5];   /* buffer for chunk to be read */
    int         column[10];        /* buffer for column to be read */
    int         rank, rank_chunk;
    hsize_t       i, j;



    /*
     * Open the file and the dataset.
     */
    file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
    dataset = H5Dopen(file, DATASETNAME);

    /*
     * Get dataset rank and dimension.
     */

    filespace = H5Dget_space(dataset);    /* Get filespace handle first. */
    rank      = H5Sget_simple_extent_ndims(filespace);
    status_n  = H5Sget_simple_extent_dims(filespace, dims, NULL);
    printf("dataset rank %d, dimensions %lu x %lu\n",
          rank, (unsigned long)(dims[0]), (unsigned long)(dims[1]));

    /*
     * Get creation properties list.
     */
    cparms = H5Dget_create_plist(dataset); /* Get properties handle first. */

    /*
     * Check if dataset is chunked.
     */
    if (H5D_CHUNKED == H5Pget_layout(cparms))  {

       /*
        * Get chunking information: rank and dimensions
        */
       rank_chunk = H5Pget_chunk(cparms, 2, chunk_dims);
       printf("chunk rank %d, dimensions %lu x %lu\n", rank_chunk,
              (unsigned long)(chunk_dims[0]), (unsigned long)(chunk_dims[1]));
    }

    /*
     * Define the memory space to read dataset.
     */
    memspace = H5Screate_simple(RANK,dims,NULL);

    /*
     * Read dataset back and display.
     */
    status = H5Dread(dataset, H5T_NATIVE_INT, memspace, filespace,
                     H5P_DEFAULT, data_out);
    printf("\n");
    printf("Dataset: \n");
    for (j = 0; j < dims[0]; j++) {
       for (i = 0; i < dims[1]; i++) printf("%d ", data_out[j][i]);
       printf("\n");
    }

    /*
     *     dataset rank 2, dimensions 10 x 5
     *     chunk rank 2, dimensions 2 x 5

     *     Dataset:
     *     1 1 1 3 3
     *     1 1 1 3 3
     *     1 1 1 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     *     2 0 0 0 0
     */

    /*
     * Read the third column from the dataset.
     * First define memory dataspace, then define hyperslab
     * and read it into column array.
     */
    col_dims[0] = 10;
    memspace =  H5Screate_simple(RANKC, col_dims, NULL);

    /*
     * Define the column (hyperslab) to read.
     */
    offset[0] = 0;
    offset[1] = 2;
```

```
count[0]  = 10;
count[1]  = 1;
status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
                        count, NULL);
status = H5Dread(dataset, H5T_NATIVE_INT, memspace, filespace,
          H5P_DEFAULT, column);
printf("\n");
printf("Third column: \n");
for (i = 0; i < 10; i++) {
   printf("%d \n", column[i]);
}

/*
 *     Third column:
 *     1
 *     1
 *     1
 *     0
 *     0
 *     0
 *     0
 *     0
 *     0
 *     0
 */

/*
 * Define the memory space to read a chunk.
 */
memspace = H5Screate_simple(rank_chunk,chunk_dims,NULL);

/*
 * Define chunk in the file (hyperslab) to read.
 */
offset[0] = 2;
offset[1] = 0;
count[0]  = chunk_dims[0];
count[1]  = chunk_dims[1];
status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
                        count, NULL);

/*
 * Read chunk back and display.
 */
status = H5Dread(dataset, H5T_NATIVE_INT, memspace, filespace,
          H5P_DEFAULT, chunk_out);
printf("\n");
printf("Chunk: \n");
for (j = 0; j < chunk_dims[0]; j++) {
   for (i = 0; i < chunk_dims[1]; i++) printf("%d ", chunk_out[j][i]);
   printf("\n");
}
/*
 *  Chunk:
 *  1 1 1 0 0
 *  2 0 0 0 0
 */

/*
 * Close/release resources.
 */
H5Pclose(cparms);
H5Dclose(dataset);
```

```
    H5Sclose(filespace);
    H5Sclose(memspace);
    H5Fclose(file);

    return 0;
```

Example 7. Creating groups.

This example shows how to create and access a group in an HDF5 file and to place a dataset within this group. It also illustrates the usage of the `H5Giterate`, `H5Glink`, and `H5Gunlink` functions.

```
/*
 * This example creates a group in the file and dataset in the group.
 * Hard link to the group object is created and the dataset is accessed
 * under different names.
 * Iterator function is used to find the object names in the root group.
 */


#include "hdf5.h"


#define FILE    "group.h5"
#define RANK    2


herr_t file_info(hid_t loc_id, const char *name, void *opdata);
                                     /* Operator function */
int
main(void)
{

    hid_t    file;
    hid_t    grp;
    hid_t    dataset, dataspace;
    hid_t    plist;

    herr_t   status;
    hsize_t  dims[2];
    hsize_t  cdims[2];

    int      idx;

    /*
     * Create a file.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create a group in the file.
     */
    grp = H5Gcreate(file, "/Data", 0);

    /*
     * Create dataset "Compressed Data" in the group using absolute
     * name. Dataset creation property list is modified to use
     * GZIP compression with the compression effort set to 6.
     * Note that compression can be used only when dataset is chunked.
     */
    dims[0] = 1000;
    dims[1] = 20;
    cdims[0] = 20;
    cdims[1] = 20;
    dataspace = H5Screate_simple(RANK, dims, NULL);
    plist     = H5Pcreate(H5P_DATASET_CREATE);
                H5Pset_chunk(plist, 2, cdims);
                H5Pset_deflate( plist, 6);
    dataset = H5Dcreate(file, "/Data/Compressed_Data", H5T_NATIVE_INT,
                        dataspace, plist);

    /*
     * Close the dataset and the file.
     */
    H5Sclose(dataspace);
    H5Dclose(dataset);
    H5Fclose(file);

    /*
     * Now reopen the file and group in the file.
     */
    file = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);
    grp  = H5Gopen(file, "Data");

    /*
     * Access "Compressed_Data" dataset in the group.
     */
    dataset = H5Dopen(grp, "Compressed_Data");
    if( dataset
```

Example 8. Writing and reading attributes.

This example shows how to create HDF5 attributes, to attach them to a dataset, and to read through all of the attributes of a dataset.

```
/*
 *  This program illustrates the usage of the H5A Interface functions.
 *  It creates and writes a dataset, and then creates and writes array,
 *  scalar, and string attributes of the dataset.
 *  Program reopens the file, attaches to the scalar attribute using
 *  attribute name and reads and displays its value. Then index of the
 *  third attribute is used to read and display attribute values.
 *  The H5Aiterate function is used to iterate through the dataset attributes,
 *  and display their names. The function is also reads and displays the values
 *  of the array attribute.
 */

#include
#include

#define FILE "Attributes.h5"

#define RANK  1   /* Rank and size of the dataset  */
#define SIZE  7

#define ARANK  2   /* Rank and dimension sizes of the first dataset attribute */
#define ADIM1  2
#define ADIM2  3
#define ANAME  "Float attribute"      /* Name of the array attribute */
#define ANAMES "Character attribute" /* Name of the string attribute */

herr_t attr_info(hid_t loc_id, const char *name, void *opdata);
                                     /* Operator function */

int
main (void)
{

   hid_t   file, dataset;       /* File and dataset identifiers */

   hid_t   fid;                 /* Dataspace identifier */
   hid_t   attr1, attr2, attr3; /* Attribute identifiers */
   hid_t   attr;
   hid_t   aid1, aid2, aid3;    /* Attribute dataspace identifiers */
   hid_t   atype;               /* Attribute type */

   hsize_t fdim[] = {SIZE};
   hsize_t adim[] = {ADIM1, ADIM2};  /* Dimensions of the first attribute  */

   float matrix[ADIM1][ADIM2]; /* Attribute data */

   herr_t  ret;                /* Return value */
   uint    i,j;                /* Counters */
   int     idx;                /* Attribute index */
   char    string_out[80];     /* Buffer to read string attribute back */
   int     point_out;          /* Buffer to read scalar attribute back */

   /*
    * Data initialization.
    */
   int vector[] = {1, 2, 3, 4, 5, 6, 7};  /* Dataset data */
   int point = 1;                         /* Value of the scalar attribute */
   char string[] = "ABCD";                /* Value of the string attribute */


   for (i=0; i < ADIM1; i++) {            /* Values of the array attribute */
       for (j=0; j < ADIM2; j++)
       matrix[i][j] = -1.;
   }

   /*
    * Create a file.
    */
   file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /*
    * Create the dataspace for the dataset in the file.
    */
   fid = H5Screate(H5S_SIMPLE);
   ret = H5Sset_extent_simple(fid, RANK, fdim, NULL);

   /*
    * Create the dataset in the file.
    */
   dataset = H5Dcreate(file, "Dataset", H5T_NATIVE_INT, fid, H5P_DEFAULT);

   /*
    * Write data to the dataset.
    */
   ret = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL , H5S_ALL, H5P_DEFAULT, vector);

   /*
    * Create dataspace for the first attribute.
    */
   aid1 = H5Screate(H5S_SIMPLE);
   ret  = H5Sset_extent_simple(aid1, ARANK, adim, NULL);

   /*
    * Create array attribute.
    */
   attr1 = H5Acreate(dataset, ANAME, H5T_NATIVE_FLOAT, aid1, H5P_DEFAULT);

   /*
    * Write array attribute.
    */
   ret = H5Awrite(attr1, H5T_NATIVE_FLOAT, matrix);

   /*
    * Create scalar attribute.
    */
   aid2  = H5Screate(H5S_SCALAR);
   attr2 = H5Acreate(dataset, "Integer attribute", H5T_NATIVE_INT, aid2,
                     H5P_DEFAULT);

   /*
    * Write scalar attribute.
    */
   ret = H5Awrite(attr2, H5T_NATIVE_INT, &point);

   /*
    * Create string attribute.
    */
   aid3  = H5Screate(H5S_SCALAR);
   atype = H5Tcopy(H5T_C_S1);
           H5Tset_size(atype, 4);
   attr3 = H5Acreate(dataset, ANAMES, atype, aid3, H5P_DEFAULT);

   /*
    * Write string attribute.
    */
   ret = H5Awrite(attr3, atype, string);

   /*
    * Close attribute and file dataspaces.
    */
   ret = H5Sclose(aid1);
   ret = H5Sclose(aid2);
   ret = H5Sclose(aid3);
   ret = H5Sclose(fid);

   /*
    * Close the attributes.
    */
   ret = H5Aclose(attr1);
   ret = H5Aclose(attr2);
   ret = H5Aclose(attr3);

   /*
    * Close the dataset.
    */
   ret = H5Dclose(dataset);

   /*
    * Close the file.
    */
   ret = H5Fclose(file);

   /*
    * Reopen the file.
    */
   file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);

   /*
    * Open the dataset.
    */
   dataset = H5Dopen(file,"Dataset");

   /*
    * Attach to the scalar attribute using attribute name, then read and
    * display its value.
    */
   attr = H5Aopen_name(dataset,"Integer attribute");
   ret  = H5Aread(attr, H5T_NATIVE_INT, &point_out);
   printf("The value of the attribute \"Integer attribute\" is %d \n", point_out);
   ret =  H5Aclose(attr);

   /*
    * Attach to the string attribute using its index, then read and display the value.
    */
   attr  = H5Aopen_idx(dataset, 2);
   atype = H5Tcopy(H5T_C_S1);
           H5Tset_size(atype, 4);
   ret   = H5Aread(attr, atype, string_out);
   printf("The value of the attribute with the index 2 is %s \n", string_out);
   ret   = H5Aclose(attr);
   ret   = H5Tclose(atype);
   /*
    * Get attribute info using iteration function.
    */
   idx = H5Aiterate(dataset, NULL, attr_info, NULL);

   /*
    * Close the dataset and the file.
    */
   H5Dclose(dataset);
   H5Fclose(file);

   return 0;
}

/*
 * Operator function.
 */
herr_t
attr_info(hid_t loc_id, const char *name, void *opdata)
{
    hid_t attr, atype, aspace;  /* Attribute, datatype and dataspace identifiers */
    int   rank;
    hsize_t sdim[64];
    herr_t ret;
    int i;
    size_t npoints;             /* Number of elements in the array attribute. */
    float *float_array;         /* Pointer to the array attribute. */
    /*
     * Open the attribute using its name.
     */
    attr = H5Aopen_name(loc_id, name);

    /*
     * Display attribute name.
     */
    printf("\n");
    printf("Name : ");
    puts(name);

    /*
     * Get attribute datatype, dataspace, rank, and dimensions.
     */
    atype  = H5Aget_type(attr);
    aspace = H5Aget_space(attr);
    rank = H5Sget_simple_extent_ndims(aspace);
    ret = H5Sget_simple_extent_dims(aspace, sdim, NULL);

    /*
     *  Display rank and dimension sizes for the array attribute.
     */

    if(rank  0) {
    printf("Rank : %d \n", rank);
    printf("Dimension sizes : ");
    for (i=0; i< rank; i++) printf("%d ", (int)sdim[i]);
    printf("\n");
    }

    /*
     * Read array attribute and display its type and values.
     */
    if (H5T_FLOAT == H5Tget_class(atype)) {
    printf("Type : FLOAT \n");
    npoints = H5Sget_simple_extent_npoints(aspace);
    float_array = (float *)malloc(sizeof(float)*(int)npoints);
    ret = H5Aread(attr, atype, float_array);
    printf("Values : ");
    for( i = 0; i < (int)npoints; i++) printf("%f ", float_array[i]);
    printf("\n");
    free(float_array);
    }

    /*
     * Release all identifiers.
     */
    H5Tclose(atype);
    H5Sclose(aspace);
    H5Aclose(attr);

    return 0;
}
```

Example 9. Creating and storing references to objects.

This example creates a group and two datasets and a named datatype in the group. References to these four objects are stored in the dataset in the root group.

```
#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void) {
    hid_t                 fid1;            /* HDF5 File IDs           */
    hid_t                 dataset; /* Dataset ID                      */
    hid_t                 group;      /* Group ID             */
    hid_t                 sid1;       /* Dataspace ID                          */
    hid_t                 tid1;       /* Datatype ID                  */
    hsize_t               dims1[] = {SPACE1_DIM1};
    hobj_ref_t      *wbuf;      /* buffer to write to disk */
    int       *tu32;      /* Temporary pointer to int data */
    int        i;          /* counting variables */
    const char *write_comment="Foo!"; /* Comments for group */
    herr_t                ret;             /* Generic return value            */

/* Compound datatype */
typedef struct s1_t {
    unsigned int a;
    unsigned int b;
    float c;
} s1_t;

    /* Allocate write buffers */
    wbuf=(hobj_ref_t *)malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32=malloc(sizeof(int)*SPACE1_DIM1);

    /* Create file */
    fid1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a group */
    group=H5Gcreate(fid1,"Group1",-1);

    /* Set group’s comment */
    ret=H5Gset_comment(group,".",write_comment);

    /* Create a dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset1",H5T_STD_U32LE,sid1,H5P_DEFAULT);

    for(i=0; i < SPACE1_DIM1; i++)
```

```
        tu32[i] = i*3;

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create another dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset2",H5T_NATIVE_UCHAR,sid1,H5P_DEFAULT);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create a datatype to refer to */
    tid1 = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));

    /* Insert fields */
    ret=H5Tinsert (tid1, "a", HOFFSET(s1_t,a), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "b", HOFFSET(s1_t,b), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "c", HOFFSET(s1_t,c), H5T_NATIVE_FLOAT);

    /* Save datatype for later */
    ret=H5Tcommit (group, "Datatype1", tid1);

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close group */
    ret = H5Gclose(group);

    /* Create a dataset to store references */
    dataset=H5Dcreate(fid1,"Dataset3",H5T_STD_REF_OBJ,sid1,H5P_DEFAULT);

    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[0],fid1,"/Group1/Dataset1",H5R_OBJECT,-1);

    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[1],fid1,"/Group1/Dataset2",H5R_OBJECT,-1);

    /* Create reference to group */
    ret = H5Rcreate(&wbuf[2],fid1,"/Group1",H5R_OBJECT,-1);

    /* Create reference to named datatype */
    ret = H5Rcreate(&wbuf[3],fid1,"/Group1/Datatype1",H5R_OBJECT,-1);

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close disk dataspace */
    ret = H5Sclose(sid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Close file */
    ret = H5Fclose(fid1);
    free(wbuf);
    free(tu32);
    return 0;
}
```

Example 10. Reading references to objects.

This example opens and reads dataset `Dataset3` from the file created in Example 9. Then the program dereferences the references to dataset `Dataset1`, the group and the named datatype, and opens those objects. The program reads and displays the dataset's data, the group's comment, and the number of members of the compound datatype.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

int
main(void)
{
    hid_t                 fid1;            /* HDF5 File IDs           */
    hid_t                 dataset, /* Dataset ID                      */
              dset2;      /* Dereferenced dataset ID */
    hid_t                 group;      /* Group ID            */
    hid_t                 sid1;       /* Dataspace ID                          */
    hid_t                 tid1;       /* Datatype ID                  */
    hobj_ref_t      *rbuf;      /* buffer to read from disk */
    int                *tu32;      /* temp. buffer read from disk */
    int        i;          /* counting variables */
    char read_comment[10];
    herr_t                ret;             /* Generic return value            */

    /* Allocate read buffers */
    rbuf = malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32 = malloc(sizeof(int)*SPACE1_DIM1);

    /* Open the file */
    fid1 = H5Fopen(FILE1, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dataset=H5Dopen(fid1,"/Dataset3");

    /* Read selection from disk */
    ret=H5Dread(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);

    /* Open dataset object */
    dset2 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);

    ret=H5Sget_simple_extent_npoints(sid1);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);
    printf("Dataset data : \n");
     for (i=0; i < SPACE1_DIM1 ; i++) printf (" %d ", tu32[i]);
    printf("\n");
    printf("\n");

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Open group object */
    group = H5Rdereference(dataset,H5R_OBJECT,&rbuf[2]);

    /* Get group’s comment */
    ret=H5Gget_comment(group,".",10,read_comment);
    printf("Group comment is %s \n", read_comment);
    printf(" \n");
    /* Close group */
    ret = H5Gclose(group);

    /* Open datatype object */
    tid1 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[3]);

    /* Verify correct datatype */
    {
        H5T_class_t tclass;

        tclass= H5Tget_class(tid1);
        if ((tclass == H5T_COMPOUND))
           printf ("Number of compound datatype members is %d \n", H5Tget_nmembers(tid1));
    printf(" \n");
    }

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Close file */
    ret = H5Fclose(fid1);

    /* Free memory buffers */
    free(rbuf);
    free(tu32);
    return 0;
}
```

Example 11. Creating and writing a reference to a region.

This example creates a dataset in the file. Then it creates a dataset to store references to the dataset regions (selections). The first selection is a 6 x 6 hyperslab. The second selection is a point selection in the same dataset. References to both selections are created and stored in the buffer, and then written to the dataset in the file.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK     1
#define SPACE1_DIM1     4

/* Dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

/* Element selection information */
#define POINT1_NPOINTS 10

int
main(void)
{
    hid_t                 fid1;            /* HDF5 File IDs           */
    hid_t                 dset1,   /* Dataset ID                      */
                dset2;      /* Dereferenced dataset ID */
    hid_t                 sid1,       /* Dataspace ID        #1                */
                sid2;       /* Dataspace ID  #2                */
    hsize_t               dims1[] = {SPACE1_DIM1},
                 dims2[] = {SPACE2_DIM1, SPACE2_DIM2};
    hssize_t      start[SPACE2_RANK];     /* Starting location of hyperslab */
    hsize_t               stride[SPACE2_RANK];    /* Stride of hyperslab */
    hsize_t               count[SPACE2_RANK];     /* Element count of hyperslab */
    hsize_t               block[SPACE2_RANK];     /* Block size of hyperslab */
    hssize_t      coord1[POINT1_NPOINTS][SPACE2_RANK];
                                   /* Coordinates for point selection */
    hdset_reg_ref_t      *wbuf;      /* buffer to write to disk */
    int     *dwbuf;      /* Buffer for writing numeric data to disk */
    int        i;          /* counting variables */
    herr_t                ret;             /* Generic return value            */


    /* Allocate write & read buffers */
    wbuf=calloc(sizeof(hdset_reg_ref_t), SPACE1_DIM1);
    dwbuf=malloc(sizeof(int)*SPACE2_DIM1*SPACE2_DIM2);

    /* Create file */
    fid1 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid2 = H5Screate_simple(SPACE2_RANK, dims2, NULL);

    /* Create a dataset */
    dset2=H5Dcreate(fid1,"Dataset2",H5T_STD_U8LE,sid2,H5P_DEFAULT);

    for(i=0; i < SPACE2_DIM1*SPACE2_DIM2; i++)
        dwbuf[i]=i*3;
    /* Write selection to disk */
    ret=H5Dwrite(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,dwbuf);

    /* Close Dataset */
    ret = H5Dclose(dset2);

    /* Create dataspace for the reference dataset */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a dataset */
    dset1=H5Dcreate(fid1,"Dataset1",H5T_STD_REF_DSETREG,sid1,H5P_DEFAULT);

    /* Create references */

    /* Select 6x6 hyperslab for first reference */
    start[0]=2; start[1]=2;
    stride[0]=1; stride[1]=1;
    count[0]=6; count[1]=6;
    block[0]=1; block[1]=1;
    ret = H5Sselect_hyperslab(sid2,H5S_SELECT_SET,start,stride,count,block);

    /* Store first dataset region */
    ret = H5Rcreate(&wbuf[0],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Select sequence of ten points for second reference */
    coord1[0][0]=6; coord1[0][1]=9;
    coord1[1][0]=2; coord1[1][1]=2;
    coord1[2][0]=8; coord1[2][1]=4;
    coord1[3][0]=1; coord1[3][1]=6;
    coord1[4][0]=2; coord1[4][1]=8;
    coord1[5][0]=3; coord1[5][1]=2;
    coord1[6][0]=0; coord1[6][1]=4;
    coord1[7][0]=9; coord1[7][1]=0;
    coord1[8][0]=7; coord1[8][1]=1;
    coord1[9][0]=3; coord1[9][1]=3;
    ret = H5Sselect_elements(sid2,H5S_SELECT_SET,POINT1_NPOINTS,(const hssize_t
**)coord1);

    /* Store second dataset region */
    ret = H5Rcreate(&wbuf[1],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Write selection to disk */
    ret=H5Dwrite(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close all objects */
    ret = H5Sclose(sid1);
    ret = H5Dclose(dset1);
    ret = H5Sclose(sid2);

    /* Close file */
    ret = H5Fclose(fid1);

    free(wbuf);
    free(dwbuf);
    return 0;
}
```

Example 12. Reading a reference to a region.

This example reads a dataset containing dataset region references. It reads data from the dereferenced dataset and displays the number of elements and raw data. Then it reads two selections: a hyperslab selection and a point selection. The program queries a number of points in the hyperslab and the coordinates and displays them. Then it queries a number of selected points and their coordinates and displays the information.

```
#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define NPOINTS 10

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void)
{
    hid_t                 fid1;             /* HDF5 File IDs           */
    hid_t                 dset1,   /* Dataset ID                       */
                dset2;      /* Dereferenced dataset ID */
    hid_t                 sid1,       /* Dataspace ID        #1                */
                sid2;       /* Dataspace ID  #2               */
    hsize_t *   coords;             /* Coordinate buffer */
    hsize_t               low[SPACE2_RANK];   /* Selection bounds */
    hsize_t               high[SPACE2_RANK];     /* Selection bounds */
    hdset_reg_ref_t      *rbuf;      /* buffer to to read disk */
    int    *drbuf;      /* Buffer for reading numeric data from disk */
    int        i, j;          /* counting variables */
    herr_t                ret;              /* Generic return value            */

    /* Output message about test being performed */

    /* Allocate write & read buffers */
    rbuf=malloc(sizeof(hdset_reg_ref_t)*SPACE1_DIM1);
    drbuf=calloc(sizeof(int),SPACE2_DIM1*SPACE2_DIM2);

    /* Open the file */
    fid1 = H5Fopen(FILE2, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dset1=H5Dopen(fid1,"/Dataset1");

    /* Read selection from disk */
    ret=H5Dread(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);

    /* Try to open objects */
    dset2 = H5Rdereference(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);
    ret=H5Sget_simple_extent_npoints(sid1);
    printf(" Number of elements in the dataset is : %d\n",ret);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,drbuf);

    for(i=0; i < SPACE2_DIM1; i++) {
        for (j=0; j < SPACE2_DIM2; j++) printf (" %d ", drbuf[i*SPACE2_DIM2+j]);
        printf("\n"); }

    /* Get the hyperslab selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Verify correct hyperslab selected */
    ret = H5Sget_select_npoints(sid2);
    printf(" Number of elements in the hyperslab is : %d \n", ret);
    ret = H5Sget_select_hyper_nblocks(sid2);
    coords=malloc(ret*SPACE2_RANK*sizeof(hsize_t)*2); /* allocate space for the hyperslab
blocks */
    ret = H5Sget_select_hyper_blocklist(sid2,0,ret,coords);
    printf(" Hyperslab coordinates are : \n");
    printf (" ( %lu , %lu ) ( %lu , %lu ) \n", \
(unsigned long)coords[0],(unsigned long)coords[1],(unsigned long)coords[2],(unsigned
long)coords[3]);
    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Get the element selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[1]);

    /* Verify correct elements selected */
    ret = H5Sget_select_elem_npoints(sid2);
    printf(" Number of selected elements is : %d\n", ret);

    /* Allocate space for the element points */
    coords= malloc(ret*SPACE2_RANK*sizeof(hsize_t));
    ret = H5Sget_select_elem_pointlist(sid2,0,ret,coords);
    printf(" Coordinates of selected elements are : \n");
    for (i=0; i < 2*NPOINTS; i=i+2)
         printf(" ( %lu , %lu ) \n", (unsigned long)coords[i],(unsigned long)coords[i+1]);

    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Close first space */
    ret = H5Sclose(sid1);

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Close Dataset */
    ret = H5Dclose(dset1);

    /* Close file */
    ret = H5Fclose(fid1);
    /* Free memory buffers */
    free(rbuf);
    free(drbuf);
    return 0;
}
```

Last modified: 16 October 1999

HDF5 Tutorial

Release 1.2
October 1999

Hierarchical Data Format (HDF) Group
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Last modified: 13 October 1999

HDF5 Tutorial

Contents

Last Modified: November 8, 1999

1. Introduction

Welcome to the HDF5 Tutorial provided by the HDF User Support Group.

HDF5 is a file format and library for storing scientific data. HDF5 was designed and implemented to address the deficiencies of HDF4.x. It has a more powerful and flexible data model, supports files larger than 2 GB, supports parallel I/O, and is thread-safe. For a short overview presentation of the HDF5 data model, library and tools see:

`http://hdf.ncsa.uiuc.edu/HDF5/HDF5_overview/index.htm`

This tutorial covers the basic HDF5 data objects and file structure, the HDF5 programming model and the API functions necessary for creating and modifying data objects. It also introduces the available HDF5 tools to access HDF5 files.

The examples used in this tutorial, along with a Makefile to compile them can be found in the `./examples/` directory in the online version of this tutorial. You can also download a tar file from NCSA's HDF server with the examples and Makefile (see `http://hdf.ncsa.uiuc.edu/training/other-ex5/examples.tar`). In order to use the Makefile you may have to edit it and update the compiler and compiler options, as well as the path for the HDF5 binary distribution.

Please check the "References" section of this tutorial for where to find other examples of HDF5 Programs.

We hope that the step-by-step examples and instructions will give you a quick start with HDF5.

Please send your comments and suggestions to `hdfhelp@ncsa.uiuc.edu`.

(Note that the HDF5 Tutorial *was designed and implemented to be used online. Since this version has been modified for PDF and PostScript format distribution, it lacks the full interactive capability. For interactive use, see either an installed version of the HDF5 document set that is distributed wih the HDF5 source code and binaries or the HDF web site,* `http://hdf.ncsa.uiuc.edu/training/hdf5/`*.)*

Last Modified: October 8, 1999

2. HDF5 File Organization

An HDF5 file is a container for storing a variety of scientific data, and the two primary HDF5 objects are groups and datasets.

- **HDF5 group:** a grouping structure containing zero or more HDF5 objects, together with supporting metadata.
- **HDF5 dataset:** a multidimensional array of data elements, together with supporting metadata.

Any HDF5 group or dataset may have an associated attribute list. An HDF5 attribute is a user-defined HDF5 structure that provides extra information about an HDF5 object.

Working with groups and group members (datasets for example) is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, objects in an HDF5 file are often described by giving their full (or absolute) path names.

`/` signifies the root group.
`/foo` signifies a member of the root group called *foo*.
`/foo/zoo` signifies a member of the group *foo*, which in turn is a member of the root group.

Last Modified: July 30, 1999

3. The HDF5 API

The HDF5 library provides several interfaces, and is currently implemented in C. The APIs provide routines for accessing HDF5 files and creating and manipulating HDF5 objects. All C routines in the HDF5 library begin with a prefix of the form H5*, where * is a single letter indicating the object on which the operation is to be performed. The APIs are listed below:

API	**DESCRIPTION**
H5	Library Functions: the general-purpose H5 functions.
H5A	Annotation Interface: attribute access and manipulating routines.
H5D	Dataset Interface: dataset access and manipulating routines.
H5E	Error Interface: error handling routines.
H5F	File Interface: file access routines.
H5G	Group Interface: group creating and operating routines.
H5I	Identifier Interface: identifier routines.
H5P	Property List Interface: object property list manipulating routines.
H5R	Reference Interface: reference routines.
H5S	Dataspace Interface: routines for defining dataspaces.
H5T	Data type Interface: routines for creating and manipulating the data type of dataset elements.
H5Z	Compression Interface: compression routine(s).

Last Modified: July 30, 1999

4. Creating an HDF5 File

What is an HDF5 file?

An HDF5 file is a binary file which contains scientific data and supporting metadata. The two primary objects stored in an HDF5 file are groups and datasets. Groups and datasets will be discussed in the other sessions.

To create a file, the program application must specify a file name, file access mode, file creation property list, and file access property list.

- **File Creation Property List:**
  The file creation property list is used to control the file metadata. File metadata contains information about the size of the user-block, the size of various file data structures used by the HDF5 library, etc.

  The user-block is a fixed length block of data located at the beginning of the file which is ignored by the HDF5 library and may be used to store any data information found to be useful to applications.

  For more details, see the HDF5 documentation. In this tutorial, the default file metadata is used.

- **File Access Property List:**
  The file access property list is used to control different methods of performing I/O on files. See the HDF5 User's Guide for details. The default file access property is used in this tutorial.

The steps to create and close an HDF5 file are as follows:

1. Specify the file creation and access property lists if necessary.
2. Create a file.
3. Close the file and close the property lists if necessary.

To create an HDF5 file, the calling program must contain the following calls:

```
file_id = H5Fcreate(filename, access_mode, create_id, access_id);

H5Fclose(file_id);
```

Programming Example

Description

The following example demonstrates how to create and close an HDF5 file. It creates a file called 'file.h5', and then closes the file.
[h5_crtfile.c]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


#include <hdf5.h>
#define FILE "file.h5"

main() {

   hid_t       file_id;   /* file identifier */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Terminate access to the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The include file 'hdf5.h' contains definitions and declarations, and it must be included in any file that uses the HDF5 library.
- H5Fcreate creates an HDF5 file and returns the file identifier.

  ```
  hid_t H5Fcreate (const char *name, unsigned flags, hid_t create_id,
                   hid_t access_id)
  ```

  - The first parameter specifies the name of the file to be created.
  - The second parameter specifies the file access mode. H5F_ACC_TRUNC will truncate a file if it already exists.
  - The third parameter specifies the file creation property list. H5P_DEFAULT indicates that the default file creation property list is used.
  - The last parameter of H5Fcreate specifies the file access property list. H5P_DEFAULT indicates that the default file access property list is used.
- When a file is no longer accessed by a program, H5Fclose must be called to release the resource used by the file. This call is mandatory.

  ```
     herr_t H5Fclose (hid_t file_id)
  ```

- The root group is automatically created when a file is created. Every file has a root group and the path name of the root group is '/'.

File Contents

HDF has developed tools to examine the contents of HDF5 files. The tool used in this tutorial is the HDF5 dumper, h5dump. h5dump is a tool that displays the file contents in human readable form to an ASCII file in DDL. DDL (Data Description Language) is a language that describes HDF5 objects in Backus-Naur Form. To view the file contents, type:

```
h5dump <filename>
```

Figure 4.1 describes the file contents of 'file.h5' using a directed graph. Each HDF5 object is represented by a rectangle and the arrows indicate the structure of the contents. In Fig. 4.2, 'file.h5' contains a group object named '/' (the root group).

Fig. 4.1 *Contents of 'file.h5'*

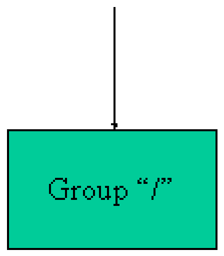


Figure 4.2 is the text-description of 'file.h5' generated by h5dump. The HDF5 file called 'file.h5' contains a group called '/'.

Fig. 4.2 *'file.h5' in DDL*

```
HDF5 "file.h5" {
GROUP "/" {
}
}
```

File Definition in DDL

Figure 4.3 is the simplified DDL file definition for creating an HDF5 file. For simplicity, a simplified DDL is used in this tutorial. A complete and more rigorous DDL can be found in the HDF5 User's Guide. See the “References” section of this tutorial.

Fig. 4.3 *HDF5 File Definition*

The explanation of the symbols used in the DDL:

```
::=                 defined as
<tname>             a token with the name tname
<a> | <b>           one of <a> or <b>
<a>*                zero or more occurrences of <a>
```

The simplified DDL file definition:

```
<file> ::= HDF5 "<file_name>" { <root_group> }

<root_group> ::= GROUP "/" { <group_attribute>* <group_member>* }

<group_attribute> ::= <attribute>

<group_member> ::= <group> | <dataset>
```

Last Modified: August 27, 1999

5. Creating a Dataset

What is a Dataset?

A dataset is a multidimensional array of data elements, together with supporting metadata. To create a dataset, the application program must specify the location to create the dataset, the dataset name, the data type and space of the data array, and the dataset creation properties.

Data Types

A data type is a collection of data type properties, all of which can be stored on disk, and which when taken as a whole, provide complete information for data conversion to or from that data type.

There are two categories of data types in HDF5: atomic and compound data types. An atomic type is a type which cannot be decomposed into smaller units at the API level. A compound data type is a collection of one or more atomic types or small arrays of such types.

Atomic types include integer, float, date and time, string, bit field, and opaque. Figure 5.1 shows the HDF5 data types. Some of the HDF5 predefined atomic data types are listed in Figure 5.2. In this tutorial, we consider only HDF5 predefined integers. For information on data types, see the HDF5 User's Guide.

Fig 5.1 *HDF5 data types*

```
                                 +--  integer
                                 +--  floating point
                 +---- atomic  ----+--  date and time
                 |               +--  character string
HDF5 datatypes --|               +--  bit field
                 |               +--  opaque
                 |
                 +---- compound
```

Fig. 5.2 *Examples of HDF5 predefined data types*

Data Type	**Description**
H5T_STD_I32LE	Four-byte, little-endian, signed two's complement integer
H5T_STD_U16BE	Two-byte, big-endian, unsigned integer
H5T_IEEE_F32BE	Four-byte, big-endian, IEEE floating point
H5T_IEEE_F64LE	Eight-byte, little-endian, IEEE floating point
H5T_C_S1	One-byte, null-terminated string of eight-bit characters

Dataspaces

A dataspace describes the dimensionality of the data array. A dataspace is either a regular N-dimensional array of data points, called a simple dataspace, or a more general collection of data points organized in another manner, called a complex dataspace. Figure 5.3 shows HDF5 dataspaces. In this tutorial, we only consider simple dataspaces.

Fig 5.3 *HDF5 dataspaces*

```
                  +-- simple
HDF5 dataspaces --|
                  +-- complex
```

The dimensions of a dataset can be fixed (unchanging), or they may be unlimited, which means that they are extendible. A dataspace can also describe portions of a dataset, making it possible to do partial I/O operations on selections.

Dataset creation properties

When creating a dataset, HDF5 allows users to specify how raw data is organized on disk and how the raw data is compressed. This information is stored in a dataset creation property list and passed to the dataset interface. The raw data on disk can be stored contiguously (in the same linear way that it is organized in memory), partitioned into chunks and stored externally, etc. In this tutorial, we use the default creation property list; that is, no compression and contiguous storage layout is used. For more information about the creation properties, see the HDF5 User’s Guide.

In HDF5, data types and spaces are independent objects, which are created separately from any dataset that they might be attached to. Because of this the creation of a dataset requires definitions of data type and dataspace. In this tutorial, we use HDF5 predefined data types (integer) and consider only simple dataspaces. Hence, only the creation of dataspace objects is needed.

To create an empty dataset (no data written) the following steps need to be taken:

4. Obtain the location id where the dataset is to be created.
5. Define the dataset characteristics and creation properties.
    - define a data type
    - define a dataspace
    - specify dataset creation properties
6. Create the dataset.
7. Close the data type, dataspace, and the property list if necessary.
8. Close the dataset.

To create a simple dataspace, the calling program must contain the following calls:

```
dataspace_id = H5Screate_simple(rank, dims, maxdims);
H5Sclose(dataspace_id );
```

To create a dataset, the calling program must contain the following calls:

```
dataset_id = H5Dcreate(hid_t loc_id, const char *name, hid_t type_id,
                       hid_t space_id, hid_t create_plist_id);
H5Dclose (dataset_id);
```

Programming Example

Description

The following example shows how to create an empty dataset. It creates a file called 'dset.h5', defines the dataset dataspace, creates a dataset which is a 4x6 integer array, and then closes the dataspace, the dataset, and the file. [`h5_crtdat.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "dset.h5"

main() {

   hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
   hsize_t     dims[2];
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data
                space for the dataset. */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                H5P_DEFAULT);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset_id);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Screate_simple creates a new simple data space and returns a data space identifier.

  ```
  hid_t H5Screate_simple (int rank, const hsize_t * dims,
                          const hsize_t * maxdims)
  ```

  - The first parameter specifies the rank of the dataset.
  - The second parameter specifies the size of the dataset.
  - The third parameter is for the upper limit on the size of the dataset. If it is NULL, the upper limit is the

same as the dimension sizes specified by the second parameter.

- H5Dcreate creates a dataset at the specified location and returns a dataset identifier.

```
hid_t H5Dcreate (hid_t loc_id, const char *name, hid_t type_id,
                 hid_t space_id, hid_t create_plist_id)
```

  - The first parameter is the location identifier.
  - The second parameter is the name of the dataset to create.
  - The third parameter is the data type identifier. H5T_STD_I32BE, a 32-bit Big Endian integer, is an HDF atomic data type.
  - The fourth parameter is the data space identifier.
  - The last parameter specifies the dataset creation property list. H5P_DEFAULT specifies the default dataset creation property list.

- H5Dcreate creates an empty array and initializes the data to 0.
- When a dataset is no longer accessed by a program, H5Dclose must be called to release the resource used by the dataset. This call is mandatory.

```
hid_t H5Dclose (hid_t dataset_id)
```

File Contents

The file contents of 'dset.h5' are shown is **Figure 5.4** and **Figure 5.5**.

Figure 5.4 *The Contents of 'dset.h5'*

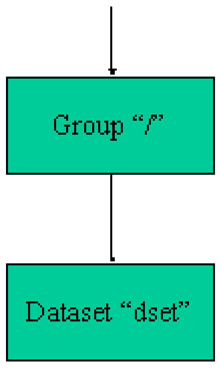


Figure 5.5 *'dset.h5' in DDL*

```
      HDF5 "dset.h5" {
      GROUP "/" {
         DATASET "dset" {
            DATATYPE { H5T_STD_I32BE }
            DATASPACE { SIMPLE ( 4, 6 ) / ( 4, 6 ) }
            DATA {
               0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0
            }
         }
      }
      }
```

Dataset Definition in DDL

The following is the simplified DDL dataset definition:

Fig. 5.6 *HDF5 Dataset Definition*

```
<dataset> ::= DATASET "<dataset_name>" { <data type>
                                         <dataspace>
                                         <data>
                                         <dataset_attribute>* }

<data type> ::= DATATYPE { <atomic_type> }

<dataspace> ::= DATASPACE { SIMPLE <current_dims> / <max_dims> }

<dataset_attribute> ::= <attribute>
```

Last Modified: August 27, 1999

6. Reading to/Writing from a Dataset

Reading to/Writing from a Dataset

During a dataset I/O operation, the library transfers raw data between memory and the file. The memory can have a data type different than the file data type and can also be a different size (memory is a subset of the dataset elements, or vice versa). Therefore, to perform read or write operations, the application program must specify:

- The dataset
- The dataset's data type in memory
- The dataset's dataspace in memory
- The dataset's dataspace in the file
- The transfer properties (The data transfer properties control various aspects of the I/O operations like the number of processes participating in a collective I/O request or hints to the library to control caching of raw data. In this tutorial, we use the default transfer properties.)
- The data buffer

The steps to read to/write from a dataset are as follows:

9. Obtain the dataset identifier.
10. Specify the memory data type.
11. Specify the memory dataspace.
12. Specify the file dataspace.
13. Specify the transfer properties.
14. Perform the desired operation on the dataset.
15. Close the dataset.
16. Close the dataspace/data type, and property list if necessary.

To read to/write from a dataset, the calling program must contain the following call:

```
H5Dread(dataset_id, mem_type_id, mem_space_id, file_space_id,
        xfer_plist_id, buf );
```

or

```
H5Dwrite(dataset_id, mem_type_id, mem_space_id, file_space_id,
         xfer_plist_id, buf);
```

Programming Example

Description

The following example shows how to read and write an existing dataset. It opens the file created in the previous example, obtains the dataset identifier, */dset*, writes the dataset to the file, then reads the dataset back from memory. It then closes the dataset and file.
[`h5_rdwt.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "dset.h5"

main() {

   hid_t       file_id, dataset_id;  /* identifiers */
   herr_t      status;
   int         i, j, dset_data[4][6];

   /* Initialize the dataset. */
   for (i = 0; i < 4; i++)
      for (j = 0; j < 6; j++)
         dset_data[i][j] = i * 6 + j + 1;

   /* Open an existing file. */
   file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

   /* Open an existing dataset. */

   dataset_id = H5Dopen(file_id, "/dset");

   /* Write the dataset. */
   status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset_data);

   status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                    dset_data);

   /* Close the dataset. */
   status = H5Dclose(dataset_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Fopen opens an existing file and returns a file identifier.

  ```
  hid_t H5Fopen (const char *name, unsigned flags, hid_t access_id)
  ```

  - The first argument is the file name.

    - The second argument is the file access mode. H5F_ACC_RDWR allows a file to be read from and written to.

    - The third parameter is the identifier for the file access property list. H5P_DEFAULT specifies the default file access property list.

- H5Dopen opens an existing dataset with the name specified by the second argument at the location specified by the first parameter, and returns an identifier.

```
hid_t H5Dopen (hid_t loc_id, const char *name)
```

- H5Dwrite writes raw data from an application buffer to the specified dataset, converting from the data type and data space of the dataset in memory to the data type and data space of the dataset in the file.

```
herr_t H5Dwrite (hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
                 hid_t file_space_id, hid_t xfer_plist_id, const void * buf)
```

    - The first parameter is the identifier of the dataset.

    - The second parameter is the identifier of the dataset's datatype in memory. H5T_NATIVE_INT is an integer data type for the machine on which the library was compiled.

    - The third parameter is the identifier of the dataset's dataspace in memory. H5S_ALL indicates that the dataset's dataspace in memory is the same as that in the file.

    - The fourth parameter is the identifier of the dataset's dataspace in the file. H5S_ALL indicates that the entire dataspace of the dataset in the file is referenced.

    - The fifth parameter is the identifier of the data transfer propery list. H5P_DEFAULT indicates that the default data transfer property list is used.

    - The last parameter is the data buffer.

- H5Dread reads raw data from the specified dataset to an application buffer, converting from the file datatype and dataspace to the memory datatype and dataspace.

```
herr_t H5Dread (hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
                hid_t file_space_id, hid_t xfer_plist_id, void * buf)
```

    - The first parameter is the identifier of the dataset read from.

    - The second parameter is the identifier of the dataset's memory datatype.

    - The third parameter is the identifier of the dataset's memory dataspace.

    - The fourth parameter is the identifier of the dataset's file dataspace.

    - The fifth parameter is the identifier of the data transfer propery list.

    - The last parameter is the data buffer.

File Contents

Figure 6.1 shows the contents of ’dset.h5’.

Fig. 6.1 *’dset.h5’ in DDL*

```
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE { H5T_STD_I32BE }
      DATASPACE { SIMPLE ( 4, 6 ) / ( 4, 6 ) }
      DATA {
         1, 2, 3, 4, 5, 6,
         7, 8, 9, 10, 11, 12,
         13, 14, 15, 16, 17, 18,
         19, 20, 21, 22, 23, 24
      }
   }
}
}
```

Last Modified: August 27, 1999

7. Creating an Attribute

What is an Attribute?

Attributes are small datasets that can be used to describe the nature and/or the intended usage of the object they are attached to. In this section, we show how to create and read/write an attribute.

Creating an attribute

Creating an attribute is similar to the creation of a dataset. To create an attribute the application must specify the object which the attribute is attached to, the data type and space of the attribute data and the creation properties.

The steps to create an attribute are as follows:

17. Obtain the object identifier that the attribute is to be attached to.
18. Define the characteristics of the attribute and specify creation properties.
    - Define the data type.
    - Define the dataspace.
    - Specify the creation properties.
19. Create the attribute.
20. Close the attribute and data type, dataspace, and creation property list if necessary.

To create an attribute, the calling program must contain the following calls:

```
attr_id = H5Acreate(loc_id, attr_name, type_id, space_id, create_plist);
H5Aclose(attr_id);
```

Reading/Writing an attribute

Attributes may only be read/written as an entire object. No partial I/O is currently supported. Therefore, to perform I/O operations on an attribute, the application needs only to specify the attribute and the attribute's memory data type.

The steps to read/write an attribute are as follows:

1. Obtain the attribute identifier.
2. Specify the attribute's memory data type.
3. Perform the desired operation.
4. Close the memory data type if necessary.

To read/write an attribute, the calling program must contain the following calls:

```
    status = H5Aread(attr_id, mem_type_id, buf);
```

or

```
    status = H5Awrite(attr_id, mem_type_id, buf);
```

Programming Example

Description

This example shows how to create and write a dataset attribute. It opens an existing file 'dset.h5', obtains the id of the dataset "/dset1", defines the attribute's dataspace, creates the dataset attribute, writes the attribute, and then closes the attribute's dataspace, attribute, dataset, and file.
[`h5_crtatt.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "dset.h5"

main() {

   hid_t       file_id, dataset_id, attribute_id, dataspace_id;  /* identifiers
*/
   hsize_t     dims;
   int         attr_data[2];
   herr_t      status;

   /* Initialize the attribute data. */
   attr_data[0] = 100;
   attr_data[1] = 200;

   /* Open an existing file. */
   file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

   /* Open an existing dataset. */
   dataset_id = H5Dopen(file_id, "/dset");

   /* Create the data space for the attribute. */
   dims = 2;
   dataspace_id = H5Screate_simple(1, &dims, NULL);

   /* Create a dataset attribute. */
   attribute_id = H5Acreate(dataset_id, "attr", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

   /* Write the attribute data. */
   status = H5Awrite(attribute_id, H5T_NATIVE_INT, attr_data);

   /* Close the attribute. */
   status = H5Aclose(attribute_id);

   /* Close the dataspace. */
   status = H5Sclose(dataspace_id);
   /* Close to the dataset. */
   status = H5Dclose(dataset_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Acreate creates an attribute which is attached to the object specified by the first parameter, and returns an identifier.

  ```
  hid_t H5Acreate (hid_t loc_id, const char *name, hid_t type_id,
                   hid_t space_id, hid_t create_plist)
  ```

  - The first parameter is the identifier of the object which the attribute is attached to.
  - The second parameter is the name of the attribute to create.
  - The third parameter is the identifier of the attribute's datatype.
  - The fourth parameter is the identifier of the attribute's dataspace.
  - The last parameter is the identifier of the creation property list. H5P_DEFAULT specifies the default creation property list.

- H5Awrite writes the entire attribute, and returns the status of the write.

  ```
  herr_t H5Awrite (hid_t attr_id, hid_t mem_type_id, void *buf)
  ```

  - The first parameter is the identifier of the attribute to write.
  - The second parameter is the identifier of the attribute's memory datatype.
  - The last parameter is the data buffer.

- When an attribute is no longer accessed by a program, H5Aclose must be called to release the attribute from use. This call is mandatory.

  ```
  herr_t H5Aclose (hid_t attr_id)
  ```

File Contents

The contents of ’dset.h5’ and the attribute definition are given in the following:

Fig. 7.1 *’dset.h5’ in DDL*

```
HDF5 "dset.h5" {
 GROUP "/" {
    DATASET "dset" {
       DATATYPE { H5T_STD_I32BE }
       DATASPACE { SIMPLE ( 4, 6 ) / ( 4, 6 ) }
       DATA {
          1, 2, 3, 4, 5, 6,
          7, 8, 9, 10, 11, 12,
          13, 14, 15, 16, 17, 18,
          19, 20, 21, 22, 23, 24
       }
       ATTRIBUTE "attr" {
          DATATYPE { H5T_STD_I32BE }
          DATASPACE { SIMPLE ( 2 ) / ( 2 ) }
          DATA {
             100, 200
          }
       }
    }
 }
 }
```

Attribute Definition in DDL

Fig. 7.2 *HDF5 Attribute Definition*

```
<attribute> ::= ATTRIBUTE "<attr_name>" { <datatype>
                                          <dataspace>
                                          <data>  }
```

Last Modified: August 27, 1999

8. Creating a Group

What is a Group?

An HDF5 group is a structure containing zero or more HDF5 objects. The two primary HDF5 objects are groups and datasets. To create a group, the calling program must:

21. Obtain the location identifier where the group is to be created.
22. Create the group.
23. Close the group.

To create a group, the calling program must contain the following calls:

```
  group_id = H5Gcreate (loc_id, name, size_hint);
  H5Gclose (group_id);
```

Programming Example

Description

The following example shows how to create and close a group. It creates a file called ’group.h5’, creates a group called *MyGroup* in the root group, and then closes the group and file.
[h5_crtgrp.c]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "group.h5"

main() {

   hid_t       file_id, group_id;  /* identifiers */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create a group named "/MyGroup" in the file. */
   group_id = H5Gcreate(file_id, "/MyGroup", 0);

   /* Close the group. */
   status = H5Gclose(group_id);

   /* Terminate access to the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Gcreate creates a new empty group and returns a group identifier.

  ```
  hid_t H5Gcreate (hid_t loc_id, const char *name, size_t size_hint)
  ```

  - The first parameter specifies the location to create the group.
  - The second parameter specifies the name of the group to be created.
  - The third parameter specifies how much file space to reserve to store the names that will appear in the group. If a non-positive value is supplied, then a default size is used. Passing a value of zero is usually adequate since the library is able to dynamically resize the name heap.
- H5Gcreate creates a group named *MyGroup* in the root group of the specified file.
- H5Gclose closes the group. This call is mandatory.

  ```
  herr_t H5Gclose (hid_t group_id)
  ```

File Contents

The contents of 'group.h5' and the definition of the group are given in the following:

Fig. 8.1 *The Contents of 'group.h5'.*

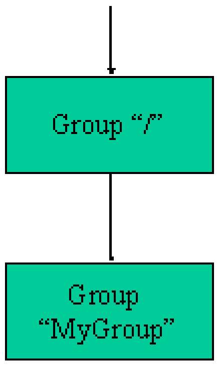


Fig. 8.2 *'group.h5' in DDL*

```
HDF5 "group.h5" {
GROUP "/" {
   GROUP "MyGroup" {
   }
}
}
```

Last Modified: August 27, 1999

9. Creating Groups using Absolute/Relative Names

Absolute vs. Relative Names

Recall that to create an HDF5 object, we have to specify the location where the object is to be created. This location is determined by the identifier of an HDF5 object and the name of the object to be created. The name of the created object can be either an absolute name or a name relative to the specified identifier. In Example 5, we used the file identifier and the absolute name "/MyGroup" to create a group. The file identifier and the name "/" specifies the location where the group "MyGroup" was created.

In this section, we discuss HDF5 names and show how to use absolute/relative names by giving an example of creating groups in a file.

Names

HDF5 object names are a slash-separated list of components. There are few restrictions on names: component names may be any length except zero and may contain any character except slash ("/") and the null terminator. A full name may be composed of any number of component names separated by slashes, with any of the component names being the special name ".". A name which begins with a slash is an absolute name which is accessed beginning with the root group of the file while all other relative names are accessed beginning with the specified group. Multiple consecutive slashes in a full name are treated as single slashes and trailing slashes are not significant. A special case is the name "/" (or equivalent) which refers to the root group.

Functions which operate on names generally take a location identifier which is either a file ID or a group ID and perform the lookup with respect to that location. Some possibilities are:

Location Type	**Object Name**	**Description**
File ID	/foo/bar	The object bar in group foo in the root group.
Group ID	/foo/bar	The object bar in group foo in the root group of the file containing the specified group. In other words, the group ID's only purpose is to supply a file.
File ID	/	The root group of the specified file.
Group ID	/	The root group of the file containing the specified group.
Group ID	foo/bar	The object bar in group foo in the specified group.
File ID	.	The root group of the file.
Group ID	.	The specified group.
Other ID	.	The specified object.

Programming Example

Description

The following example code shows how to create groups using absolute and relative names. It creates three groups: the first two groups are created using the file identifier and the group absolute names, and the third group is created using a group identifier and the name relative to the specified group.
[`h5_crtgrpar.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "groups.h5"

main() {

   hid_t       file_id, group1_id, group2_id, group3_id;  /* identifiers */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create group "MyGroup" in the root group using absolute name. */
   group1_id = H5Gcreate(file_id, "/MyGroup", 0);

   /* Create group "Group_A" in group "MyGroup" using absolute name. */
   group2_id = H5Gcreate(file_id, "/MyGroup/Group_A", 0);

   /* Create group "Group_B" in group "MyGroup" using relative name. */
   group3_id = H5Gcreate(group1_id, "Group_B", 0);

   /* Close groups. */
   status = H5Gclose(group1_id);
   status = H5Gclose(group2_id);
   status = H5Gclose(group3_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Gcreate creates a group at the location specified by a location ID and a name. The location ID can be a file ID or a group ID and the name can be relative or absolute.
- The first H5Gcreate creates the group 'MyGroup' in the root group of the specified file.
- The second H5Gcreate creates the group 'Group_A' in the group 'MyGroup' in the root group of the specified file. Note that the parent group (MyGroup) already exists.
- The third H5Gcreate creates the group 'Group_B' in the specified group.

File Contents

The file contents are shown below:

Fig. 9.1 *The Contents of 'groups.h5'*

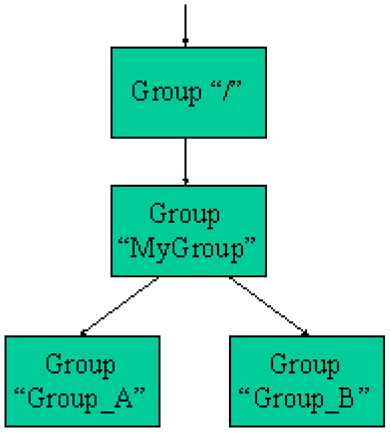


Fig. 9.2 *'groups.h5' in DDL*

```
HDF5 "groups.h5" {
GROUP "/" {
   GROUP "MyGroup" {
      GROUP "Group_A" {
      }
      GROUP "Group_B" {
      }
   }
}
}
```

Last Modified: August 27, 1999

10. Creating Datasets in Groups

Creating datasets in groups

We have shown how to create groups, datasets and attributes. In this section, we show how to create datasets in groups. Recall that H5Dcreate creates a dataset at the location specified by a location identifier and a name. Similar to H5Gcreate, the location identifier can be a file identifier or a group identifier and the name can be relative or absolute. The location identifier and the name together determine the location where the dataset is to be created. If the location identifier and name refers to a group, then the dataset is created in that group.

Programming Example

Description

This example shows how to create a dataset in a particular group. It opens the file created in the previous example and creates two datasets.
[`h5_crtgrpd.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>
#define FILE "groups.h5"

main() {

   hid_t       file_id, group_id, dataset_id, dataspace_id;  /* identifiers */
   hsize_t     dims[2];
   herr_t      status;
   int         i, j, dset1_data[3][3], dset2_data[2][10];

   /* Initialize the first dataset. */
   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
         dset1_data[i][j] = j + 1;

   /* Initialize the second dataset. */
   for (i = 0; i < 2; i++)
      for (j = 0; j < 10; j++)
         dset2_data[i][j] = j + 1;

   /* Open an existing file. */
   file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

   /* Create the data space for the first dataset. */
   dims[0] = 3;
   dims[1] = 3;
   dataspace_id = H5Screate_simple(2, dims, NULL);
   /* Create a dataset in group "MyGroup". */
   dataset_id = H5Dcreate(file_id, "/MyGroup/dset1", H5T_STD_I32BE, dataspace_id
,
                       H5P_DEFAULT);

   /* Write the first dataset. */
   status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset1_data);

   /* Close the data space for the first dataset. */
   status = H5Sclose(dataspace_id);

   /* Close the first dataset. */
   status = H5Dclose(dataset_id);

   /* Open an existing group of the specified file. */
   group_id = H5Gopen(file_id, "/MyGroup/Group_A");

   /* Create the data space for the second dataset. */
   dims[0] = 2;
   dims[1] = 10;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the second dataset in group "Group_A". */
   dataset_id = H5Dcreate(group_id, "dset2", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

   /* Write the second dataset. */
   status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset2_data);

   /* Close the data space for the second dataset. */
   status = H5Sclose(dataspace_id);

   /* Close the second dataset */
   status = H5Dclose(dataset_id);

   /* Close the group. */
   status = H5Gclose(group_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

File Contents

Fig. 10.1 *The Contents of 'groups.h5'*

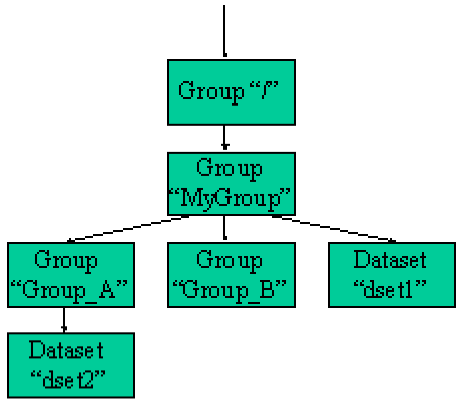


Fig. 10.2 *'groups.h5' in DDL*

```
HDF5 "groups.h5" {
   GROUP "/" {
      GROUP "MyGroup" {
         GROUP "Group_A" {
            DATASET "dset2" {
               DATATYPE { H5T_STD_I32BE }
               DATASPACE { SIMPLE ( 2, 10 ) / ( 2, 10 ) }
               DATA {
                  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
               }
            }
         }
         GROUP "Group_B" {
         }
         DATASET "dset1" {
            DATATYPE { H5T_STD_I32BE }
            DATASPACE { SIMPLE ( 3, 3 ) / ( 3, 3 ) }
            DATA {
               1, 2, 3,
               1, 2, 3,
               1, 2, 3
            }
         }
      }
   }
   }
```

Last Modified: August 27, 1999

Introductory Topics Questions

```
Section 2: HDF File Organization
================================

1. Name and describe the two primary objects that can be stored in an HDF5
   file:

2. What is an attribute?

3. Give the path name for an object called "harry" that is a member of a
   group called "dick," which in turn is a member of the root group.


Section 3: The HDF5 API
=======================

Describe the purpose of each of the following HDF5 APIs:

H5A, H5D, H5E, F5F, H5G, H5T, H5Z



Section 4: Creating an HDF File
===============================

1. What two HDF5 routines must be called in order to create an HDF5 file?

2. What include file must be included in any file that uses the HDF5 library.

3. An HDF5 file is never completely empty because as soon as an HDF5 file
   is created, it automatically contains a certain primary object.  What is
   that object?



Section 5: Creating a Dataset
=============================

1. Name and describe two major datatype categories.

2. List the HDF5 atomic datatypes. Give an example of a predefined datatype.

3. What does the dataspace describe? What are the major characteristics of the
   simple dataspace?

4. What information needs to be passed to the H5Dcreate function, i.e.
   what information is needed to describe a dataset at creation time?
```

```
Section 6: Reading from/Writing to a Dataset
============================================

1. What are six pieces of information which need to be specified for
   reading and writing a dataset?

2. Why are both the memory dataspace and file dataspace needed for
   read/write operations, but only the memory datatype is specified for the
   datatype?

3. What does the line DATASPACE { SIMPLE (4 , 6 ) / ( 4 , 6 ) } in Fig 6.1
   means?


Section 7: Creating an Attribute
================================

1. What is an attribute?

2. Can partial I/O operations be performed on attributes?


Section 8: Creating a Group
===========================

What are the two primary objects that can be included in
a group?


Section 9: Creating Groups using Absolute/Relative Names
========================================================

1. Group names can be specified in two "ways".  What are these
   two types of group names that you can specify?

2. You have a dataset named "moo" in the group "boo", which is
   in the group "foo", which in turn, is in the root group.  How would
   you specify an absolute name to access this dataset?


Section 10: Creating Datasets in Groups
=======================================

Describe a way to access the dataset "moo" described in the previous section
(Section 9, question 2), using a relative and absolute pathname.
```

Last Modified: August 2, 1999

Introductory Topics Questions with Answers

```
Section 2: HDF File Organization
================================

1. Name and describe the two primary objects that can be stored in an HDF5
   file:

Answer:
Group: A grouping structure containing zero or more HDF5 objects, together
       with supporting metadata.

Dataset: A multidimensional array of data elements, together with
         supporting metadata.

2. What is an attribute?

Answer: An HDF attribute is a user-defined HDF5 structure that provides extra
        information about an HDF5 object.

3. Give the path name for an object called "harry" that is a member of a
   group called "dick," which in turn is a member of the root group.

Answer: /dick/harry

Section 3: The HDF5 API
=======================

Describe the purpose of each of the following HDF5 APIs:

H5A, H5D, H5E, F5F, H5G, H5T, H5Z

H5A: Attribute access and manipulation routines.
H5D: Dataset access and manipulation routines.
H5E: Error handling routines.
F5F: File access routines.
H5G: Routines for creating and operating on groups.
H5T: Routines for creating and manipulating the datatypes of dataset elements.
H5Z: Data compression routines.


Section 4: Creating an HDF File
===============================

1. What two HDF5 routines must be called in order to create an HDF5 file?

Answer: H5Fcreate and H5Fclose.

2. What include file must be included in any file that uses the HDF5 library.

Answer: hdf5.h must be included because it contains definitions and
        declarations used by the library.
```

```
3. An HDF5 file is never completely empty because as soon as an HDF5 file
   is created, it automatically contains a certain primary object.  What is
   that object?

Answer: The root group.


Section 5: Creating a Dataset
=============================

1. Name and describe two major datatype categories.

Answer: atomic datatype - An atomic datatype cannot be decomposed into
                          smaller units at the API level.
        compound datatype - A compound datatype is a collection of atomic/
                            compound datatypes, or small arrays of such types.

2. List the HDF5 atomic datatypes. Give an example of a predefined datatype.

Answer: There are six HDF5 atomic datatypes: integer, floating point,
        date and time, character string, bit field, opaque.
        H5T_IEEE_F32LE - 4-byte little-endian, IEEE floating point,
        H5T_NATIVE_INT - native integer

3. What does the dataspace describe? What are the major characteristics of the
   simple dataspace?

Answer: The dataspace describes the dimensionality of the dataset. It is
        characterized by its rank and dimension sizes.

4. What information needs to be passed to the H5Dcreate function, i.e.
   what information is needed to describe a dataset at creation time?

Answer:  dataset location, name, dataspace, datatype, and creation properties.


Section 6: Reading from/Writing to a Dataset
============================================

1. What are six pieces of information which need to be specified for
   reading and writing a dataset?

Answer: A dataset, a dataset’s datatype and dataspace in memory, the
        dataspace in the file, the transfer properties and data buffer.

2. Why are both the memory dataspace and file dataspace needed for
   read/write operations, but only the memory datatype is specified for the
   datatype?

Answer: A dataset’s file datatype is specified at creation time and cannot be
        changed. Both file and memory dataspaces are needed for performing
        subsetting and partial I/O operations.

3. What does the line DATASPACE { SIMPLE (4 , 6 ) / ( 4 , 6 ) } in Fig 6.1
   means?

Answer: It means that the dataset "dset" has a simple dataspace with the
        current dimensions (4,6) and the maximum size of the dimensions (4,6).
```

```
Section 7: Creating an Attribute
================================

1. What is an attribute?

Answer: An attribute is a dataset attached to an object. It describes the
        nature and/or the intended usage of the object.

2. Can partial I/O operations be performed on attributes?

Answer: No


Section 8: Creating a Group
===========================

What are the two primary objects that can be included in
a group?

Answer:  A group and a dataset


Section 9: Creating Groups using Absolute/Relative Names
========================================================

1. Group names can be specified in two "ways".  What are these
   two types of group names that you can specify?

Answer: relative and absolute

2. You have a dataset named "moo" in the group "boo", which is
   in the group "foo", which in turn, is in the root group.  How would
   you specify an absolute name to access this dataset?

Answer: /foo/boo/moo

Section 10: Creating Datasets in Groups
=======================================

Describe a way to access the dataset "moo" described in the previous section
(Section 9, question 2), using a relative and absolute pathname.

Answers: 1. Access the group, "/foo", and get the group ID.
            Access the group "boo" using the group ID obtained in Step 1.
            Access the dataset "moo" using the group ID in Step 2.
               gid = H5Gopen (file_id, "/foo", 0);       /* absolute path */
               gid1 = H5Gopen (gid, "boo", 0);           /* relative path */
               did = H5Dopen (gid1, "moo");              /* relative path */

         2. Access the group, "/foo", and get the group ID.
            Access the dataset "boo/moo", with the group ID just obtained.
               gid = H5Gopen (file_id, "/foo", 0);       /* absolute path */
               did = H5Dopen (gid, "boo/moo");           /* relative path */

         3. Access the dataset with an absolute path.
               did = H5Dopen (file_id, "/foo/boo/moo");  /* absolute path */
```

Last Modified: August 2, 1999

11. Compound Data Types

Creating Compound Data Types

A compound data type is similar to a struct in C or a common block in Fortran. It is a collection of one or more atomic types or small arrays of such types. To create and use a compound data type you need to refer to various properties of the data compound data type:

- It is of class compound.
- It has a fixed total size, in bytes.
- It consists of zero or more members (defined in any order) with unique names and which occupy non-overlapping regions within the datum.
- Each member has its own data type.
- Each member is referenced by an index number between zero and N-1, where N is the number of members in the compound data type.
- Each member has a name which is unique among its siblings in a compound data type.
- Each member has a fixed byte offset, which is the first byte (smallest byte address) of that member in a compound data type.
- Each member can be a small array of up to four dimensions.

Properties of members of a compound data type are defined when the member is added to the compound type and cannot be subsequently modified.

Compound data types must be built out of other data types. First, one creates an empty compound data type and specifies its total size. Then members are added to the compound data type in any order.

Programming Example

Description

This example shows how to create a compound data type, write an array to the file which uses the compound data type, and read back subsets of the members.
[compound.c]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#include "hdf5.h"

#define FILE          "SDScompound.h5"
#define DATASETNAME   "ArrayOfStructures"
#define LENGTH        10
#define RANK          1

int
main(void)
{

    /* First structure  and dataset*/
    typedef struct s1_t {
         int    a;
         float  b;
         double c;
    } s1_t;
    s1_t       s1[LENGTH];
    hid_t      s1_tid;     /* File datatype identifier */

    /* Second structure (subset of s1_t)  and dataset*/
    typedef struct s2_t {
         double c;
         int    a;
    } s2_t;
    s2_t       s2[LENGTH];
    hid_t      s2_tid;    /* Memory datatype handle */

    /* Third "structure" ( will be used to read float field of s1) */
    hid_t      s3_tid;   /* Memory datatype handle */
    float      s3[LENGTH];

    int        i;
    hid_t      file, dataset, space; /* Handles */
    herr_t     status;
    hsize_t    dim[] = {LENGTH};   /* Dataspace dimensions */


    /*
     * Initialize the data
     */
    for (i = 0; i < LENGTH; i++) {
        s1[i].a = i;
        s1[i].b = i*i;
        s1[i].c = 1./(i+1);
    }

    /*
```

```
     * Create the data space.
     */
    space = H5Screate_simple(RANK, dim, NULL);

    /*
     * Create the file.
     */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create the memory data type.
     */
    s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
    H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
    H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
    H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_FLOAT);

    /*
     * Create the dataset.
     */
    dataset = H5Dcreate(file, DATASETNAME, s1_tid, space, H5P_DEFAULT);

    /*
     * Wtite data to the dataset;
     */
    status = H5Dwrite(dataset, s1_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s1);

    /*
     * Release resources
     */
    H5Tclose(s1_tid);
    H5Sclose(space);
    H5Dclose(dataset);
    H5Fclose(file);

    /*
     * Open the file and the dataset.
     */
    file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);

    dataset = H5Dopen(file, DATASETNAME);

    /*
     * Create a data type for s2
     */
    s2_tid = H5Tcreate(H5T_COMPOUND, sizeof(s2_t));

    H5Tinsert(s2_tid, "c_name", HOFFSET(s2_t, c), H5T_NATIVE_DOUBLE);
    H5Tinsert(s2_tid, "a_name", HOFFSET(s2_t, a), H5T_NATIVE_INT);

    /*
     * Read two fields c and a from s1 dataset. Fields in the file
     * are found by their names "c_name" and "a_name".
     */
    status = H5Dread(dataset, s2_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s2);

    /*
     * Display the fields
     */
    printf("\n");
    printf("Field c : \n");
    for( i = 0; i < LENGTH; i++) printf("%.4f ", s2[i].c);
    printf("\n");
    printf("\n");
    printf("Field a : \n");
    for( i = 0; i < LENGTH; i++) printf("%d ", s2[i].a);
    printf("\n");

    /*
     * Create a data type for s3.
     */
    s3_tid = H5Tcreate(H5T_COMPOUND, sizeof(float));

    status = H5Tinsert(s3_tid, "b_name", 0, H5T_NATIVE_FLOAT);

    /*
     * Read field b from s1 dataset. Field in the file is found by its name.
     */
    status = H5Dread(dataset, s3_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s3);

    /*
     * Display the field
     */
    printf("\n");
    printf("Field b : \n");
    for( i = 0; i < LENGTH; i++) printf("%.4f ", s3[i]);
    printf("\n");

    /*
     * Release resources
     */
    H5Tclose(s2_tid);
    H5Tclose(s3_tid);
    H5Dclose(dataset);
    H5Fclose(file);

    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The program outputs the following:

```
Field c :
1.0000 0.5000 0.3333 0.2500 0.2000 0.1667 0.1429 0.1250 0.1111 0.1000

Field a :
0 1 2 3 4 5 6 7 8 9

Field b :
0.0000 1.0000 4.0000 9.0000 16.0000 25.0000 36.0000 49.0000 64.0000 81.0000

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Tcreate creates a new data type of the specified class with the specified number of bytes.

  ```
  hid_t H5Tcreate ( H5T_class_t class, size_t size )
  ```

  - The *class* parameter specifies the data type to create. Currently only the H5T_COMPOUND data type class is supported with this function.

    - The *size* parameter specifies the number of bytes in the data type to create.

- H5Tinsert adds a member to the compound data type specified by *type_id*.

```
   herr_t H5Tinsert ( hid_t type_id, const char * name, off_t offset, hid_t field_id
)
```

    - The *type_id* parameter is the identifier of the compound data type to modify.
    - The *name* parameter is the name of the field to insert. The new member name must be unique within a compound data type.
    - The *offset* parameter is the offset in the memory structure of the field to insert. The library defines the HOFFSET macro to compute the offset of a member within a struct:

      ```
      HOFFSET ( s, m )
      ```

      This macro computes the offset of member *m* within a struct variable *s*.

    - The *field_id* parameter is the data type identifier of the field to insert.

- H5Tclose releases a data type.

```
   herr_t H5Tclose ( hid_t type_id )
```

  The *type_id* parameter is the identifier of the data type to release.

File Contents

```
HDF5 "SDScompound.h5" {
GROUP "/" {
   DATASET "ArrayOfStructures" {
      DATATYPE {
         H5T_STD_I32BE "a_name";
         H5T_IEEE_F32BE "b_name";
         H5T_IEEE_F64BE "c_name";
      }
      DATASPACE { SIMPLE ( 10 ) / ( 10 ) }
      DATA {
         {
            [ 0 ],
            [ 0 ],
            [ 1 ]
         },
         {
            [ 1 ],
            [ 1 ],
            [ 0.5 ]
         },
         {
            [ 2 ],
            [ 4 ],
            [ 0.333333 ]
         },
         {
            [ 3 ],
            [ 9 ],
            [ 0.25 ]
```

```
         },
         {
            [ 4 ],
            [ 16 ],
            [ 0.2 ]
         },
         {
            [ 5 ],
            [ 25 ],
            [ 0.166667 ]
         },
         {
            [ 6 ],
            [ 36 ],
            [ 0.142857 ]
         },
         {
            [ 7 ],
            [ 49 ],
            [ 0.125 ]
         },
         {
            [ 8 ],
            [ 64 ],
            [ 0.111111 ]
         },
         {
            [ 9 ],
            [ 81 ],
            [ 0.1 ]
         }
      }
   }
}
}
```

Last Modified: August 27, 1999

12. Selections using H5Sselect_hyperslab

Selecting a Portion of a Dataspace

Hyperslabs are portions of datasets. A hyperslab selection can be a logically contiguous collection of points in a dataspace, or it can be a regular pattern of points or blocks in a dataspace. You can select a hyperslab to write to/read from with the function H5Sselect_hyperslab.

Programming Example

Description

This example creates a 5 x 6 integer array in a file called sds.h5. It selects a 3 x 4 hyperslab from the dataset, as follows (Dimension 0 is offset by 1 and Dimension 1 is offset by 2):

		X	X	X	X
		X	X	X	X
		X	X	X	X

Then it reads the hyperslab from this file into a 2-dimensional plane (size 7 x 7) of a 3-dimensional array (size 7 x 7 x 3), as follows (with Dimension 0 offset by 3):

X	X	X	X			
X	X	X	X			
X	X	X	X			

[h5_hyperslab.c]

```
/************************************************************

  This example shows how to write and read a hyperslab.  It
  is derived from the h5_read.c and h5_write.c examples in
  the "Introduction to HDF5".

 ************************************************************/

#include "hdf5.h"

#define FILE        "sds.h5"
#define DATASETNAME "IntArray"
#define NX_SUB  3                     /* hyperslab dimensions */
#define NY_SUB  4
#define NX 7                          /* output buffer dimensions */
#define NY 7
#define NZ  3
#define RANK         2
#define RANK_OUT     3

#define X     5                       /* dataset dimensions */
#define Y     6

int
main (void)
{
    hsize_t     dimsf[2];              /* dataset dimensions */
    int         data[X][Y];            /* data to write */

    /*
     * Data  and output buffer initialization.
     */
    hid_t       file, dataset;         /* handles */
    hid_t       dataspace;
    hid_t       memspace;
    hsize_t     dimsm[3];              /* memory space dimensions */
    hsize_t     dims_out[2];           /* dataset dimensions */
    herr_t      status;

    int         data_out[NX][NY][NZ ]; /* output buffer */

    hsize_t     count[2];              /* size of the hyperslab in the file */
    hssize_t    offset[2];             /* hyperslab offset in the file */
    hsize_t     count_out[3];          /* size of the hyperslab in memory */
    hssize_t    offset_out[3];         /* hyperslab offset in memory */
    int         i, j, k, status_n, rank;



/*********************************************************
   This writes data to the HDF5 file.
 *********************************************************/

    /*
     * Data  and output buffer initialization.
     */
    for (j = 0; j < X; j++) {
         for (i = 0; i < Y; i++)
             data[j][i] = i + j;
    }
    /*
     * 0 1 2 3 4 5
     * 1 2 3 4 5 6
     * 2 3 4 5 6 7
     * 3 4 5 6 7 8
     * 4 5 6 7 8 9
     */

    /*
     * Create a new file using H5F_ACC_TRUNC access,
     * the default file creation properties, and the default file
     * access properties.
     */
    file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Describe the size of the array and create the data space for fixed
     * size dataset.
     */
    dimsf[0] = X;
    dimsf[1] = Y;
    dataspace = H5Screate_simple (RANK, dimsf, NULL);

    /*
     * Create a new dataset within the file using defined dataspace and
     * default dataset creation properties.
     */
    dataset = H5Dcreate (file, DATASETNAME, H5T_STD_I32BE, dataspace,
                         H5P_DEFAULT);

    /*
     * Write the data to the dataset using default transfer properties.
     */
    status = H5Dwrite (dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                       H5P_DEFAULT, data);

    /*
     * Close/release resources.
     */
    H5Sclose (dataspace);
    H5Dclose (dataset);
    H5Fclose (file);


/*************************************************************

  This reads the hyperslab from the sds.h5 file just
  created, into a 2-dimensional plane of the 3-dimensional
  array.

 ************************************************************/

    for (j = 0; j < NX; j++) {
         for (i = 0; i < NY; i++) {
             for (k = 0; k < NX; j++) {
         for (i = 0; i < NY; i++) printf("%d ", data_out[j][i][0]);
         printf("\n ");
    }
         printf("\n");
    /*
     * 0 0 0 0 0 0 0
     * 0 0 0 0 0 0 0
     * 0 0 0 0 0 0 0
     * 3 4 5 6 0 0 0
     * 4 5 6 7 0 0 0
     * 5 6 7 8 0 0 0
     * 0 0 0 0 0 0 0
     */

    /*
     * Close and release resources.
     */
    H5Dclose (dataset);
    H5Sclose (dataspace);
    H5Sclose (memspace);
    H5Fclose (file);

}
```

Remarks

- H5Sselect_hyperslab selects a hyperslab region to add to the current selected region for a specified dataspace.

  ```
  herr_t H5Sselect_hyperslab (hid_t space_id, H5S_seloper_t op,
         const hssize_t *start, const hsize_t *stride,
         const hsize_t *count, const hsize_t *block )
  ```

  - The first parameter, *space_id*, is the dataspace identifier for the specified dataspace.
  - The second parameter, *op*, can only be set to H5S_SELECT_SET in the current release. It replaces the existing selection with the parameters from this call. Overlapping blocks are not supported.
  - The *start* array determines the starting coordinates of the hyperslab to select.
  - The *stride* array indicates which elements along a dimension are to be selected.
  - The *count* array determines how many positions to select from the dataspace in each dimension.
  - The *block* array determines the size of the element block selected by the dataspace.

  The *start*, *stride*, *count*, and *block* arrays must be the same size as the rank of the dataspace.

- This example introduces the following H5Dget_* functions:

  H5Dget_space: returns an identifier for a copy of the dataspace of a dataset.
  H5Dget_type: returns an identifier for a copy of the data type of a dataset.

- This example introduces the following H5Sget_* functions used to obtain information about selections:

  H5Sget_simple_extent_dims: returns the size and maximum sizes of each dimension of a dataspace.
  H5Sget_simple_extent_ndims: determines the dimensionality (or rank) of a dataspace.

Last Modified: August 27, 1999

13. Selections using `H5Sselect_elements` and `H5SCopy`

Selecting Independent Points and Copying a Dataspace

You can select independent points to read or write to in a dataspace by use of the H5Sselect_elements function.

The H5Scopy function allows you to make an exact copy of a dataspace, which can help cut down on the number of function calls needed when selecting a dataspace.

Programming Example

Description

This example shows you how to use H5Sselect_elements to select individual points in a dataset and how to use H5Scopy to make a copy of a dataspace. [`h5_copy.c`]

```
/***********************************************************************/
/*                                                                     */
/*  PROGRAM:   h5_copy.c                                               */
/*  PURPOSE:   Shows how to use the H5SCOPY function.                  */
/*  DESCRIPTION:                                                       */
/*             This program creates two files, copy1.h5, and copy2.h5. */
/*             In copy1.h5, it creates a 3x4 dataset called 'Copy1',   */
/*             and write 0's to this dataset.                          */
/*             In copy2.h5, it create a 3x4 dataset called 'Copy2',    */
/*             and write 1's to this dataset.                          */
/*             It closes both files, reopens both files, selects two   */
/*             points in copy1.h5 and writes values to them.  Then it  */
/*             does an H5Scopy from the first file to the second, and  */
/*             writes the values to copy2.h5.  It then closes the      */
/*             files, reopens them, and prints the contents of the     */
/*             two datasets.                                           */
/*                                                                     */
/***********************************************************************/

#include "hdf5.h"
#define FILE1 "copy1.h5"
#define FILE2 "copy2.h5"

#define RANK  2
#define DIM1  3
#define DIM2  4
#define NUMP  2

int main (void)
{
     hid_t   file1, file2, dataset1, dataset2;
     hid_t   mid1, mid2, fid1, fid2;
     hsize_t fdim[] = {DIM1, DIM2};
     hsize_t mdim[] = {DIM1, DIM2};
     hsize_t start[2], stride[2], count[2], block[2];
     int buf1[DIM1][DIM2];
     int buf2[DIM1][DIM2];
     int bufnew[DIM1][DIM2];
     int val[] = {53, 59};
     hsize_t marray[] = {2};
     hssize_t coord[NUMP][RANK];
     herr_t ret;
     uint  i, j;

/***********************************************************************/
/*                                                                     */
/* Create two files containing identical datasets. Write 0’s to one    */
/* and 1’s to the other.                                               */
/*                                                                     */
/***********************************************************************/

     for ( i = 0; i < DIM1; i++ )
         for ( j = 0; j < DIM2; j++ )
             buf1[i][j] = 0;

     for ( i = 0; i < DIM1; i++ )
         for ( j = 0; j < DIM2; j++ )
             buf2[i][j] = 1;

     file1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
     file2 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

     fid1 = H5Screate_simple (RANK, fdim, NULL);
     fid2 = H5Screate_simple (RANK, fdim, NULL);

     dataset1 = H5Dcreate (file1, "Copy1", H5T_NATIVE_INT, fid1, H5P_DEFAULT);
     dataset2 = H5Dcreate (file2, "Copy2", H5T_NATIVE_INT, fid2, H5P_DEFAULT);

     ret = H5Dwrite(dataset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf1);
     ret = H5Dwrite(dataset2, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf2);

     ret = H5Dclose (dataset1);
     ret = H5Dclose (dataset2);

     ret = H5Sclose (fid1);
     ret = H5Sclose (fid2);

     ret = H5Fclose (file1);
     ret = H5Fclose (file2);

/***********************************************************************/
/*                                                                     */
/* Open the two files.  Select two points in one file, write values to */
/* those point locations, then do H5Scopy and write the values to the  */
/* other file.  Close files.                                           */
/*                                                                     */
/***********************************************************************/

     file1 = H5Fopen (FILE1, H5F_ACC_RDWR, H5P_DEFAULT);
     file2 = H5Fopen (FILE2, H5F_ACC_RDWR, H5P_DEFAULT);
     dataset1 = H5Dopen (file1, "Copy1");
     dataset2 = H5Dopen (file2, "Copy2");
     fid1 = H5Dget_space (dataset1);
     mid1 = H5Screate_simple(1, marray, NULL);
     coord[0][0] = 0; coord[0][1] = 3;
     coord[1][0] = 0; coord[1][1] = 1;

     ret = H5Sselect_elements (fid1, H5S_SELECT_SET, NUMP, (const hssize_t **)coord);

     ret = H5Dwrite (dataset1, H5T_NATIVE_INT, mid1, fid1, H5P_DEFAULT, val);

     fid2 = H5Scopy (fid1);

     ret = H5Dwrite (dataset2, H5T_NATIVE_INT, mid1, fid2, H5P_DEFAULT, val);

     ret = H5Dclose (dataset1);
     ret = H5Dclose (dataset2);
     ret = H5Sclose (fid1);
     ret = H5Sclose (fid2);
     ret = H5Fclose (file1);
     ret = H5Fclose (file2);
     ret = H5Sclose (mid1);

/***********************************************************************/
/*                                                                     */
/* Open both files and print the contents of the datasets.             */
/*                                                                     */
/***********************************************************************/

     file1 = H5Fopen (FILE1, H5F_ACC_RDWR, H5P_DEFAULT);
     file2 = H5Fopen (FILE2, H5F_ACC_RDWR, H5P_DEFAULT);
     dataset1 = H5Dopen (file1, "Copy1");
     dataset2 = H5Dopen (file2, "Copy2");

     ret = H5Dread (dataset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                    H5P_DEFAULT, bufnew);

     printf ("\nDataset ’Copy1’ in file ’copy1.h5’ contains: \n");
     for (i=0;i<DIM1; i++) {
        for (j=0;j<DIM2;j++) printf ("%3d  ", bufnew[i][j]);
        printf("\n");
     }

     printf ("\nDataset ’Copy2’ in file ’copy2.h5’ contains: \n");

     ret = H5Dread (dataset2, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                    H5P_DEFAULT, bufnew);

     for (i=0;i<DIM1; i++) {
        for (j=0;j<DIM2;j++) printf ("%3d  ", bufnew[i][j]);
        printf("\n");
     }
     ret = H5Dclose (dataset1);
     ret = H5Dclose (dataset2);
     ret = H5Fclose (file1);
     ret = H5Fclose (file2);

}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- H5Sselect_elements selects array elements to be included in the selection for a dataspace:

```
herr_t H5Sselect_elements (hid_t space_id, H5S_seloper_t op,
                           size_t num_elem, const hssize_t **coord )
```

- The *space_id* parameter is the dataspace identifier.
- The *op* parameter currently can only be set to H5S_SELECT_SET and specifies to replace the existing selection with the parameters from this call.
- The *coord* array is a two-dimensional array of size 'dataspace rank' by the number of elements to be selected, *num_elem*.

- H5Scopy creates an exact copy of a dataspace:

```
hid_t H5Scopy(hid_t space_id )
```

- The *space_id* parameter is the dataspace identifier to copy.

File Contents

Following is the DDL for *copy1.h5* and *copy2.h5*, as viewed with the commands "h5dump copy1.h5" and "h5dump copy2.h5".

Fig. S.1 *'copy1.h5' in DDL*

```
HDF5 "copy1.h5" {
GROUP "/" {
   DATASET "Copy1" {
      DATATYPE { H5T_STD_I32BE }
      DATASPACE { SIMPLE ( 3, 4 ) / ( 3, 4 ) }
      DATA {
         0, 59, 0, 53,
         0, 0, 0, 0,
         0, 0, 0, 0
      }
   }
}
}
```

Fig. S.2 *'copy2.h5' in DDL*

```
HDF5 "copy2.h5" {
GROUP "/" {
   DATASET "Copy2" {
      DATATYPE { H5T_STD_I32BE }
      DATASPACE { SIMPLE ( 3, 4 ) / ( 3, 4 ) }
      DATA {
         1, 59, 1, 53,
         1, 1, 1, 1,
         1, 1, 1, 1
      }
   }
}
}
```

Last Modified: August 27, 1999

14. References to Objects

References to Objects

In HDF5, objects (i.e. groups, datasets, and named data types) are usually accessed by name. This access method was discussed in previous sections. There is another way to access stored objects - by reference.

An object reference is based on the relative file address of the object header in the file and is constant for the life of the object. Once a reference to an object is created and stored in a dataset in the file, it can be used to dereference the object it points to. References are handy for creating a file index or for grouping related objects by storing references to them in one dataset.

Creating and Storing References to Objects

The following steps are involved in creating and storing file references to objects:

24. Create the objects or open them if they already exist in the file.
25. Create a dataset to store the objects' references.
26. Create and store references to the objects in a buffer.
27. Write a buffer with the references to the dataset.

Programming Example

Description

The example below creates a group and two datasets and a named data type in the group. References to these four objects are stored in the dataset in the root group. [`h5_ref2objw.c`]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4
```

```
/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void) {
    hid_t                 fid1;             /* HDF5 File IDs           */
    hid_t                 dataset; /* Dataset ID                       */
    hid_t                 group;      /* Group ID             */
    hid_t                 sid1;       /* Dataspace ID                          */
    hid_t                 tid1;       /* Datatype ID                   */
    hsize_t               dims1[] = {SPACE1_DIM1};
    hobj_ref_t      *wbuf;      /* buffer to write to disk */
    int       *tu32;      /* Temporary pointer to int data */
    int        i;          /* counting variables */
    const char *write_comment="Foo!"; /* Comments for group */
    herr_t                ret;              /* Generic return value             */

/* Compound datatype */
typedef struct s1_t {
    unsigned int a;
    unsigned int b;
    float c;
} s1_t;

    /* Allocate write buffers */
    wbuf=(hobj_ref_t *)malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32=malloc(sizeof(int)*SPACE1_DIM1);

    /* Create file */
    fid1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a group */
    group=H5Gcreate(fid1,"Group1",-1);

    /* Set group's comment */
    ret=H5Gset_comment(group,".",write_comment);

    /* Create a dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset1",H5T_STD_U32LE,sid1,H5P_DEFAULT);

    for(i=0; i < SPACE1_DIM1; i++)
        tu32[i] = i*3;

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create another dataset (inside Group1) */
    dataset=H5Dcreate(group,"Dataset2",H5T_NATIVE_UCHAR,sid1,H5P_DEFAULT);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create a datatype to refer to */
    tid1 = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));

    /* Insert fields */
    ret=H5Tinsert (tid1, "a", HOFFSET(s1_t,a), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "b", HOFFSET(s1_t,b), H5T_NATIVE_INT);

    ret=H5Tinsert (tid1, "c", HOFFSET(s1_t,c), H5T_NATIVE_FLOAT);

    /* Save datatype for later */
    ret=H5Tcommit (group, "Datatype1", tid1);

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close group */
    ret = H5Gclose(group);

    /* Create a dataset to store references */
    dataset=H5Dcreate(fid1,"Dataset3",H5T_STD_REF_OBJ,sid1,H5P_DEFAULT);

    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[0],fid1,"/Group1/Dataset1",H5R_OBJECT,-1);

    /* Create reference to dataset */
    ret = H5Rcreate(&wbuf[1],fid1,"/Group1/Dataset2",H5R_OBJECT,-1);

    /* Create reference to group */
    ret = H5Rcreate(&wbuf[2],fid1,"/Group1",H5R_OBJECT,-1);

    /* Create reference to named datatype */
    ret = H5Rcreate(&wbuf[3],fid1,"/Group1/Datatype1",H5R_OBJECT,-1);

    /* Write selection to disk */
    ret=H5Dwrite(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close disk dataspace */
    ret = H5Sclose(sid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Close file */
    ret = H5Fclose(fid1);
    free(wbuf);
    free(tu32);
    return 0;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The following code,

  ```
  dataset = H5Dcreate ( fid1,"Dataset3",H5T_STD_REF_OBJ,sid1,H5P_DEFAULT );
  ```

  creates a dataset to store references. Notice that the H5T_SDT_REF_OBJ data type is used to specify that references to objects will be stored. The data type H5T_STD_REF_DSETREG is used to store the dataset region references and will be discussed later in this tutorial.

- The next few calls to the H5Rcreate function create references to the objects and store them in the buffer *wbuf*. The signature of the H5Rcreate function is:

```
    herr_t H5Rcreate ( void* buf, hid_t loc_id, const char *name,
                       H5R_type_t ref_type, hid_t space_id )
```

  - The first argument specifies the buffer to store the reference.
  - The second and third arguments specify the name of the referenced object. In our example, the file identifier *fid1* and absolute name of the dataset "/Group1/Dataset1" were used to identify the dataset. One could also use the group identifier of group "Group1" and the relative name of the dataset "Dataset1" to create the same reference.
  - The fourth argument specifies the type of the reference. Our example uses references to the objects (H5R_OBJECT). Another type of reference, reference to the dataset region (H5R_DATASET_REGION), will be discussed later in this tutorial.
  - The fifth argument specifies the space identifier. When references to the objects are created it should be set to -1.

- The H5Dwrite function writes a dataset with the references to the file. Notice that the H5T_SDT_REF_OBJ data type is used to describe the dataset's memory data type.

File Contents

The contents of the "trefer1.h5" file created by this example are as follows:

```
                        Fig    "trefer1.h5"
==============================================================================

HDF5 "trefer1.h5" {
GROUP "/" {
   DATASET "Dataset3" {
      DATATYPE { H5T_REFERENCE }
      DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
      DATA {
         DATASET 0:1696, DATASET 0:2152, GROUP 0:1320, DATATYPE 0:2268
      }
   }
   GROUP "Group1" {
      DATASET "Dataset1" {
         DATATYPE { H5T_STD_U32LE }
         DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
         DATA {
            0, 3, 6, 9
         }
      }
      DATASET "Dataset2" {
         DATATYPE { H5T_STD_U8LE }
         DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
         DATA {
            0, 0, 0, 0
         }
      }
      DATATYPE "Datatype1" {
         H5T_STD_I32BE "a";
         H5T_STD_I32BE "b";
         H5T_IEEE_F32BE "c";
      }
   }
}
}
==================================================================================
```

Notice how the data in dataset "Dataset3" is described. The two numbers with the colon in between represent a unique identifier of the object. These numbers are constant for the life of the object.

Reading References and Accessing Objects Using References

The following steps are involved:

5. Open the dataset with the references and read them. The H5T_STD_REF_OBJ data type must be used to describe the memory data type.

6. Use the read reference to obtain the identifier of the object the reference points to.

7. Open the dereferenced object and perform the desired operations.

8. Close all objects when the task is complete.

Programming Example

Description

The example below opens and reads dataset "Dataset3" from the file created previously. Then the program dereferences the references to dataset "Dataset1", the group and the named data type, and opens those objects. The program reads and displays the dataset's data, the group's comment and the number of members of the compound data type.
[`h5_ref2objr.c`]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <stdlib.h>
#include <hdf5.h>

#define FILE1   "trefer1.h5"

/* dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

int
main(void)
{
    hid_t                 fid1;             /* HDF5 File IDs           */
    hid_t                 dataset, /* Dataset ID                       */
                dset2;      /* Dereferenced dataset ID */
    hid_t                 group;      /* Group ID             */
    hid_t                 sid1;       /* Dataspace ID                          */
    hid_t                 tid1;       /* Datatype ID                   */
    hobj_ref_t      *rbuf;      /* buffer to read from disk */
    int                *tu32;      /* temp. buffer read from disk */
    int        i;          /* counting variables */
    char read_comment[10];
    herr_t                 ret;              /* Generic return value             */

    /* Allocate read buffers */
    rbuf = malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);
    tu32 = malloc(sizeof(int)*SPACE1_DIM1);

    /* Open the file */
    fid1 = H5Fopen(FILE1, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dataset=H5Dopen(fid1,"/Dataset3");

    /* Read selection from disk */
    ret=H5Dread(dataset,H5T_STD_REF_OBJ,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);

    /* Open dataset object */
    dset2 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);

    ret=H5Sget_simple_extent_npoints(sid1);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,tu32);
    printf("Dataset data : \n");
     for (i=0; i < SPACE1_DIM1 ; i++) printf (" %d ", tu32[i]);
    printf("\n");
    printf("\n");

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Open group object */
    group = H5Rdereference(dataset,H5R_OBJECT,&rbuf[2]);

    /* Get group’s comment */
    ret=H5Gget_comment(group,".",10,read_comment);
    printf("Group comment is %s \n", read_comment);
    printf(" \n");
    /* Close group */
    ret = H5Gclose(group);

    /* Open datatype object */
    tid1 = H5Rdereference(dataset,H5R_OBJECT,&rbuf[3]);

    /* Verify correct datatype */
    {
        H5T_class_t tclass;

        tclass= H5Tget_class(tid1);
        if ((tclass == H5T_COMPOUND))
           printf ("Number of compound datatype members is %d \n", H5Tget_nmembers(tid1));
    printf(" \n");
    }

    /* Close datatype */
    ret = H5Tclose(tid1);

    /* Close Dataset */
    ret = H5Dclose(dataset);
    /* Close file */
    ret = H5Fclose(fid1);

    /* Free memory buffers */
    free(rbuf);
    free(tu32);
    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Following is the output of this program:

```
Dataset data :
 0  3  6  9

Group comment is Foo!

Number of compound datatype members is 3
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The H5Dread function was used to read dataset "Dataset3" containing the references to the objects. The H5T_STD_REF_OBJ memory data type was used to read references to memory.

- H5Rdereference obtains the object's identifier. The signature of this function is:

  ```
  hid_t H5Rdereference (hid_t datatset, H5R_type_t ref_type, void *ref)
  ```

  - The first argument is an identifier of the dataset with the references.

  - The second argument specifies the reference type. We used H5R_OBJECT to specify a reference to an object. Another type is H5R_DATASET_REGION to specify a reference to a dataset region. This will be discussed later in this tutorial.

  - The third argument is a buffer to store the reference to be read.

  - The function returns an identifier of the object the reference points to. In our simplified situation we know what type was stored in the dataset. When the type of the object is unknown, then H5Rget_object_type should be used to identify the type of object the reference points to.

Last Modified: August 27, 1999

15. References to Dataset Regions

References to Dataset Regions

Previously you learned about creating, reading, and writing dataset selections. Here you will learn how to store dataset selections in a file, and how to read them back using references to the dataset regions.

A dataset region reference points to the dataset selection by storing the relative file address of the dataset header and the global heap offset of the referenced selection. The selection referenced is located by retrieving the coordinates of the areas in the selection from the global heap. This internal mechanism of storing and retrieving dataset selections is transparent to the user. A reference to the dataset selection (region) is constant for the life of the dataset.

Creating and Storing References to Dataset Regions

The following steps are involved in creating and storing references to the dataset regions:

28. Create a dataset to store the dataset regions (selections).
29. Create selections in the dataset(s). Dataset(s) should already exist in the file.
30. Create references to the selections and store them in a buffer.
31. Write references to the dataset regions in the file.
32. Close all objects.

Programming Example

Description

The example below creates a dataset in the file. Then it creates a dataset to store references to the dataset regions (selections). The first selection is a 6 x 6 hyperslab. The second selection is a point selection in the same dataset. References to both selections are created and stored in the buffer, and then written to the dataset in the file.
[`h5_ref2regw.c`]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK     1
#define SPACE1_DIM1     4
```

```
/* Dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

/* Element selection information */
#define POINT1_NPOINTS 10

int
main(void)
{
    hid_t               fid1;           /* HDF5 File IDs          */
    hid_t               dset1,   /* Dataset ID                    */
              dset2;      /* Dereferenced dataset ID */
    hid_t               sid1,       /* Dataspace ID        #1               */
              sid2;       /* Dataspace ID  #2              */
    hsize_t             dims1[] = {SPACE1_DIM1},
                dims2[] = {SPACE2_DIM1, SPACE2_DIM2};
    hssize_t     start[SPACE2_RANK];     /* Starting location of hyperslab */
    hsize_t             stride[SPACE2_RANK];    /* Stride of hyperslab */
    hsize_t             count[SPACE2_RANK];     /* Element count of hyperslab */
    hsize_t             block[SPACE2_RANK];     /* Block size of hyperslab */
    hssize_t     coord1[POINT1_NPOINTS][SPACE2_RANK];
                                  /* Coordinates for point selection */
    hdset_reg_ref_t      *wbuf;      /* buffer to write to disk */
    int     *dwbuf;      /* Buffer for writing numeric data to disk */
    int        i;          /* counting variables */
    herr_t              ret;            /* Generic return value            */


    /* Allocate write & read buffers */
    wbuf=calloc(sizeof(hdset_reg_ref_t), SPACE1_DIM1);
    dwbuf=malloc(sizeof(int)*SPACE2_DIM1*SPACE2_DIM2);

    /* Create file */
    fid1 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid2 = H5Screate_simple(SPACE2_RANK, dims2, NULL);

    /* Create a dataset */
    dset2=H5Dcreate(fid1,"Dataset2",H5T_STD_U8LE,sid2,H5P_DEFAULT);

    for(i=0; i < SPACE2_DIM1*SPACE2_DIM2; i++)
        dwbuf[i]=i*3;

    /* Write selection to disk */
    ret=H5Dwrite(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,dwbuf);

    /* Close Dataset */
    ret = H5Dclose(dset2);

    /* Create dataspace for the reference dataset */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /* Create a dataset */
    dset1=H5Dcreate(fid1,"Dataset1",H5T_STD_REF_DSETREG,sid1,H5P_DEFAULT);

    /* Create references */

    /* Select 6x6 hyperslab for first reference */
    start[0]=2; start[1]=2;
    stride[0]=1; stride[1]=1;
    count[0]=6; count[1]=6;
    block[0]=1; block[1]=1;
    ret = H5Sselect_hyperslab(sid2,H5S_SELECT_SET,start,stride,count,block);

    /* Store first dataset region */
    ret = H5Rcreate(&wbuf[0],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Select sequence of ten points for second reference */
    coord1[0][0]=6; coord1[0][1]=9;
    coord1[1][0]=2; coord1[1][1]=2;
    coord1[2][0]=8; coord1[2][1]=4;
    coord1[3][0]=1; coord1[3][1]=6;
    coord1[4][0]=2; coord1[4][1]=8;
    coord1[5][0]=3; coord1[5][1]=2;
    coord1[6][0]=0; coord1[6][1]=4;
    coord1[7][0]=9; coord1[7][1]=0;
    coord1[8][0]=7; coord1[8][1]=1;
    coord1[9][0]=3; coord1[9][1]=3;
    ret = H5Sselect_elements(sid2,H5S_SELECT_SET,POINT1_NPOINTS,(const hssize_t
**)coord1);

    /* Store second dataset region */
    ret = H5Rcreate(&wbuf[1],fid1,"/Dataset2",H5R_DATASET_REGION,sid2);

    /* Write selection to disk */
    ret=H5Dwrite(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,wbuf);

    /* Close all objects */
    ret = H5Sclose(sid1);
    ret = H5Dclose(dset1);
    ret = H5Sclose(sid2);

    /* Close file */
    ret = H5Fclose(fid1);

    free(wbuf);
    free(dwbuf);
    return 0;
}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The code,

  ```
  dset1=H5Dcreate(fid1,"Dataset1",H5T_STD_REF_DSETREG,sid1,H5P_DEFAULT);
  ```

  creates a dataset to store references to the dataset(s) regions(selections). Notice that the H5T_STD_REF_DSETREG data type is used.

- This program uses hyperslab and point selections. We used the dataspace handle *sid2* for the calls to H5Sselect_hyperslab and H5Sselect_elements. The handle was created when dataset "Dataset2" was created and it describes the dataset's dataspace. We did not close it when the dataset was closed to decrease the number of function calls used in the example. In a real application program, one should open the dataset and determine its dataspace using the H5Dget_space function.

- H5Rcreate is used to create a dataset region reference and store it in a buffer. The signature of the function is:

```
herr_t H5Rcreate(void *buf, hid_t loc_id, const char *name,
                 H5R_type_t ref_type, hid_t space_id)
```

- The first argument specifies the buffer to store the reference.
- The second and third arguments specify the name of the referenced dataset. In our example the file identifier *fid1* and the absolute name of the dataset "/Dataset2" were used to identify the dataset. The reference to the region of this dataset is stored in the buffer *buf*.
- The fourth argument specifies the type of the reference. Since we are creating references to the dataset regions, the H5R_DATASET_REGION data type is used.
- The fifth argument is a dataspace identifier of the referenced dataset.

File Contents

The contents of the file "trefer2.h5" created by this program are as follows:

```
HDF5 "trefer2.h5" {
GROUP "/" {
   DATASET "Dataset1" {
      DATATYPE { H5T_REFERENCE }
      DATASPACE { SIMPLE ( 4 ) / ( 4 ) }
      DATA {
         DATASET 0:744 {(2,2)-(7,7)}, DATASET 0:744 {(6,9), (2,2), (8,4), (1,6),
          (2,8), (3,2), (0,4), (9,0), (7,1), (3,3)}, NULL, NULL
      }
   }
   DATASET "Dataset2" {
      DATATYPE { H5T_STD_U8LE }
      DATASPACE { SIMPLE ( 10, 10 ) / ( 10, 10 ) }
      DATA {
         0, 3, 6, 9, 12, 15, 18, 21, 24, 27,
         30, 33, 36, 39, 42, 45, 48, 51, 54, 57,
         60, 63, 66, 69, 72, 75, 78, 81, 84, 87,
         90, 93, 96, 99, 102, 105, 108, 111, 114, 117,
         120, 123, 126, 129, 132, 135, 138, 141, 144, 147,
         150, 153, 156, 159, 162, 165, 168, 171, 174, 177,
         180, 183, 186, 189, 192, 195, 198, 201, 204, 207,
         210, 213, 216, 219, 222, 225, 228, 231, 234, 237,
         240, 243, 246, 249, 252, 255, 255, 255, 255, 255,
         255, 255, 255, 255, 255, 255, 255, 255, 255, 255
      }
   }
}
}
```

Notice how raw data of the dataset with the dataset regions is displayed. Each element of the raw data consists of a reference to the dataset (DATASET number1:number2) and its selected region. If the selection is a hyperslab, the corner coordinates of the hyperslab are displayed. For the point selection, the coordinates of each point are displayed. Since only two selections were stored, the third and fourth elements of the dataset "Dataset1" are set to NULL. This was done by the buffer inizialization in the program.

Reading References to Dataset Regions

The following steps are involved in reading references to the dataset regions and referenced dataset regions (selections).

9. Open and read the dataset containing references to the dataset regions. The data type H5T_STD_REF_DSETREG must be used during read operation.

10. Use H5Rdereference to obtain the dataset identifier from the read dataset region reference.

    OR

    Use H5Rget_region to obtain the dataspace identifier for the dataset containing the selection from the read dataset region reference.

11. With the dataspace identifier, the H5S interface functions, H5Sget_select_*, can be used to obtain information about the selection.

12. Close all objects when they are no longer needed.

Programming Example

Description

The following example reads a dataset containing dataset region references. It reads data from the dereferenced dataset and displays the number of elements and raw data. Then it reads two selections: hyperslab and point. The program queries a number of points in the hyperslab and the coordinates and displays them. Then it queries a number of selected points and their coordinates and displays the information.
[`h5_ref2regr.c`]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


#include <stdlib.h>
#include <hdf5.h>

#define FILE2     "trefer2.h5"
#define NPOINTS 10

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

/* 2-D dataset with fixed dimensions */
#define SPACE2_NAME  "Space2"
#define SPACE2_RANK       2
#define SPACE2_DIM1       10
#define SPACE2_DIM2       10

int
main(void)
{
    hid_t                 fid1;            /* HDF5 File IDs          */
    hid_t                 dset1,   /* Dataset ID                     */
              dset2;      /* Dereferenced dataset ID */
    hid_t                  sid1,       /* Dataspace ID        #1                 */
                sid2;       /* Dataspace ID  #2                */
    hsize_t *   coords;             /* Coordinate buffer */
    hsize_t                low[SPACE2_RANK];   /* Selection bounds */
    hsize_t                high[SPACE2_RANK];     /* Selection bounds */
    hdset_reg_ref_t      *rbuf;      /* buffer to to read disk */
    int    *drbuf;      /* Buffer for reading numeric data from disk */
    int        i, j;          /* counting variables */
    herr_t                 ret;             /* Generic return value             */

    /* Output message about test being performed */

    /* Allocate write & read buffers */
    rbuf=malloc(sizeof(hdset_reg_ref_t)*SPACE1_DIM1);
    drbuf=calloc(sizeof(int),SPACE2_DIM1*SPACE2_DIM2);

    /* Open the file */
    fid1 = H5Fopen(FILE2, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dset1=H5Dopen(fid1,"/Dataset1");

    /* Read selection from disk */
    ret=H5Dread(dset1,H5T_STD_REF_DSETREG,H5S_ALL,H5S_ALL,H5P_DEFAULT,rbuf);

    /* Try to open objects */
    dset2 = H5Rdereference(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Check information in referenced dataset */
    sid1 = H5Dget_space(dset2);

    ret=H5Sget_simple_extent_npoints(sid1);
    printf(" Number of elements in the dataset is : %d\n",ret);

    /* Read from disk */
    ret=H5Dread(dset2,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,drbuf);

    for(i=0; i < SPACE2_DIM1; i++) {
        for (j=0; j < SPACE2_DIM2; j++) printf (" %d ", drbuf[i*SPACE2_DIM2+j]);
        printf("\n"); }

    /* Get the hyperslab selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[0]);

    /* Verify correct hyperslab selected */
    ret = H5Sget_select_npoints(sid2);
    printf(" Number of elements in the hyperslab is : %d \n", ret);
    ret = H5Sget_select_hyper_nblocks(sid2);
    coords=malloc(ret*SPACE2_RANK*sizeof(hsize_t)*2); /* allocate space for the hyperslab
blocks */
    ret = H5Sget_select_hyper_blocklist(sid2,0,ret,coords);
    printf(" Hyperslab coordinates are : \n");
    printf (" ( %lu , %lu ) ( %lu , %lu ) \n", \
(unsigned long)coords[0],(unsigned long)coords[1],(unsigned long)coords[2],(unsigned
long)coords[3]);
    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Get the element selection */
    sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[1]);
    /* Verify correct elements selected */
    ret = H5Sget_select_elem_npoints(sid2);
    printf(" Number of selected elements is : %d\n", ret);

    /* Allocate space for the element points */
    coords= malloc(ret*SPACE2_RANK*sizeof(hsize_t));
    ret = H5Sget_select_elem_pointlist(sid2,0,ret,coords);
    printf(" Coordinates of selected elements are : \n");
    for (i=0; i < 2*NPOINTS; i=i+2)
         printf(" ( %lu , %lu ) \n", (unsigned long)coords[i],(unsigned long)coords[i+1]);

    free(coords);
    ret = H5Sget_select_bounds(sid2,low,high);

    /* Close region space */
    ret = H5Sclose(sid2);

    /* Close first space */
    ret = H5Sclose(sid1);

    /* Close dereferenced Dataset */
    ret = H5Dclose(dset2);

    /* Close Dataset */
    ret = H5Dclose(dset1);

    /* Close file */
    ret = H5Fclose(fid1);

    /* Free memory buffers */
    free(rbuf);
    free(drbuf);
    return 0;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Output of this program is :

```
 Number of elements in the dataset is : 100
 0  3  6  9  12  15  18  21  24  27
 30  33  36  39  42  45  48  51  54  57
 60  63  66  69  72  75  78  81  84  87
 90  93  96  99  102  105  108  111  114  117
 120  123  126  129  132  135  138  141  144  147
 150  153  156  159  162  165  168  171  174  177
 180  183  186  189  192  195  198  201  204  207
 210  213  216  219  222  225  228  231  234  237
 240  243  246  249  252  255  255  255  255  255
 255  255  255  255  255  255  255  255  255  255
 Number of elements in the hyperslab is : 36
 Hyperslab coordinates are :
 ( 2 , 2 ) ( 7 , 7 )
 Number of selected elements is : 10
 Coordinates of selected elements are :
 ( 6 , 9 )
 ( 2 , 2 )
 ( 8 , 4 )
 ( 1 , 6 )
 ( 2 , 8 )
 ( 3 , 2 )
 ( 0 , 4 )
 ( 9 , 0 )
 ( 7 , 1 )
 ( 3 , 3 )

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The dataset with the region references was read by H5Dread with the H5T_STD_REF_DSETREG data type specified.
- The read reference can be used to obtain the dataset identifier as we did with the following call:

  ```
  dset2 = H5Rdereference (dset1,H5R_DATASET_REGION,&rbuf[0]);
  ```

  or to obtain spacial information (dataspace and selection) with the call to H5Rget_region:

  ```
  sid2=H5Rget_region(dset1,H5R_DATASET_REGION,&rbuf[0]);
  ```

  The reference to the dataset region has information for both the dataset itself and its selection. In both functions:

  - The first parameter is an identifier of the dataset with the region references.
  - The second parameter specifies the type of reference stored. In this example a reference to the dataset region is stored.
  - The third parameter is a buffer containing the reference of the specified type.
- This example introduces several H5Sget_select* functions used to obtain information about selections:

  H5Sget_select_npoints: returns the number of elements in the hyperslab
  H5Sget_select_hyper_nblocks: returns the number of blocks in the hyperslab
  H5Sget_select_blocklist: returns the "lower left" and "upper right" coordinates of the blocks in the hyperslab selection
  H5Sget_select_bounds: returns the coordinates of the "minimal" block containing a hyperslab selection
  H5Sget_select_elem_npoints: returns the number of points in the element selection
  H5Sget_select_elem_points: returns the coordinates of the element selection

Last Modified: August 27, 1999

16. Chunking and Extendible Datasets

Creating an Extendible Dataset

An extendible dataset is one whose dimensions can grow. In HDF5, it is possible to define a dataset to have certain initial dimensions, then later to increase the size of any of the initial dimensions.

HDF5 requires you to use chunking in order to define extendible datasets. Chunking makes it possible to extend datasets efficiently, without having to reorganize storage excessively.

The following operations are required in order to write an extendible dataset:

33. Declare the dataspace of the dataset to have unlimited dimensions for all dimensions that might eventually be extended.
34. Set dataset creation properties to enable chunking and create a dataset.
35. Extend the size of the dataset.

Programming Example

Description

This example shows how to create a 3 x 3 extendible dataset, write to that dataset, extend the dataset to 10x3, and write to the dataset again. [`h5_extend.c`]

```
/**************************************************************
 *
 *   This example shows how to work with extendible datasets.
 *   In the current version of the library a dataset MUST be
 *   chunked in order to be extendible.
 *
 *   This example is derived from the h5_extend_write.c and
 *   h5_read_chunk.c examples that are in the "Introduction
 *   to HDF5".
 *
 *************************************************************/

#include "hdf5.h"

#define FILE        "ext.h5"
#define DATASETNAME "ExtendibleArray"
#define RANK         2

int
main (void)
```

```
{
    hid_t       file;                          /* handles */
    hid_t       dataspace, dataset;
    hid_t       filespace;
    hid_t       cparms;
    hid_t       memspace;

    hsize_t      dims[2]  = { 3, 3};           /* dataset dimensions
                                                  at creation time */
    hsize_t      dims1[2] = { 3, 3};           /* data1 dimensions */
    hsize_t      dims2[2] = { 7, 1};           /* data2 dimensions */

    hsize_t      maxdims[2] = {H5S_UNLIMITED, H5S_UNLIMITED};
    hsize_t      size[2];
    hssize_t     offset[2];
    hsize_t      i,j;
    herr_t       status, status_n;
    int          data1[3][3] = { {1, 1, 1},      /* data to write */
                                {1, 1, 1},
                                {1, 1, 1} };

    int          data2[7]    = { 2, 2, 2, 2, 2, 2, 2};

    /* Variables used in reading data back */
    hsize_t      chunk_dims[2] ={2, 5};
    hsize_t      chunk_dimsr[2];
    hsize_t      dimsr[2];
    int          data_out[10][3];
    int          rank, rank_chunk;

    /* Create the data space with unlimited dimensions. */
    dataspace = H5Screate_simple (RANK, dims, maxdims);

    /* Create a new file. If file exists its contents will be overwritten. */
    file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Modify dataset creation properties, i.e. enable chunking  */
    cparms = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_chunk ( cparms, RANK, chunk_dims);

    /* Create a new dataset within the file using cparms
       creation properties.  */
    dataset = H5Dcreate (file, DATASETNAME, H5T_NATIVE_INT, dataspace,
                         cparms);

    /* Extend the dataset. This call assures that dataset is 3 x 3.*/
    size[0]   = 3;
    size[1]   = 3;
    status = H5Dextend (dataset, size);

    /* Select a hyperslab  */
    filespace = H5Dget_space (dataset);
    offset[0] = 0;
    offset[1] = 0;
    status = H5Sselect_hyperslab (filespace, H5S_SELECT_SET, offset, NULL,
                                  dims1, NULL);

    /* Write the data to the hyperslab  */
    status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, filespace,
                       H5P_DEFAULT, data1);

    /* Extend the dataset. Dataset becomes 10 x 3  */
    dims[0]   = dims1[0] + dims2[0];
    size[0]   = dims[0];
    size[1]   = dims[1];
    status = H5Dextend (dataset, size);

    /* Select a hyperslab  */
    filespace = H5Dget_space (dataset);
    offset[0] = 3;
    offset[1] = 0;
    status = H5Sselect_hyperslab (filespace, H5S_SELECT_SET, offset, NULL,
                                  dims2, NULL);

    /* Define memory space */
    dataspace = H5Screate_simple (RANK, dims2, NULL);

    /* Write the data to the hyperslab  */
    status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, filespace,
                       H5P_DEFAULT, data2);

    /* Close resources */
    status = H5Dclose (dataset);
    status = H5Sclose (dataspace);
    status = H5Sclose (filespace);
    status = H5Fclose (file);

/****************************************************************
    Read the data back
 ***************************************************************/

    file = H5Fopen (FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
    dataset = H5Dopen (file, DATASETNAME);
    filespace = H5Dget_space (dataset);
    rank = H5Sget_simple_extent_ndims (filespace);
    status_n = H5Sget_simple_extent_dims (filespace, dimsr, NULL);

    cparms = H5Dget_create_plist (dataset);
    if (H5D_CHUNKED == H5Pget_layout (cparms))
    {
       rank_chunk = H5Pget_chunk (cparms, 2, chunk_dimsr);
    }

    memspace = H5Screate_simple (rank,dimsr,NULL);
    status = H5Dread (dataset, H5T_NATIVE_INT, memspace, filespace,
                      H5P_DEFAULT, data_out);
    printf("\n");
    printf("Dataset: \n");
    for (j = 0; j < dimsr[0]; j++)
    {
       for (i = 0; i < dimsr[1]; i++)
           printf("%d ", data_out[j][i]);
       printf("\n");
    }

    status = H5Pclose (cparms);
    status = H5Dclose (dataset);
    status = H5Sclose (filespace);
    status = H5Sclose (memspace);
    status = H5Fclose (file);
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The function H5Pcreate creates a new property as an instance of a property list. The signature of this function is as follows:

  ```
  hid_t H5Pcreate ( H5P_class_t type )
  ```

  - The parameter *type* is the type of property list to create.
    The class types are: H5P_FILE_CREATE, H5P_FILE_ACCESS, H5P_DATASET_CREATE, H5P_DATASET_XFER, and H5P_MOUNT

- The function H5Pset_chunk sets the size of the chunks used to store a chunked layout dataset. The signature of this function is as follows:

  ```
  herr_t H5Pset_chunk ( hid_t plist, int ndims, const hsize_t * dim )
  ```

  - The first parameter, *plist*, is the identifier for the property list to query.
  - The second parameter, *ndims*, is the number of dimensions of each chunk.
  - The third parameter, *dim*, is an array containing the size of each chunk.

  A non-negative value is returned if successful; otherwise a negative value is returned.

- The function H5Dextend extends a dataset that has an unlimited dimension. The signature is as follows:

  ```
  herr_t H5Dextend ( hid_t dataset_id, const hsize_t * size )
  ```

  - The first parameter, *dataset_id*, is the identifier of the dataset.
  - The second parater, *size*, is an array containing the new magnitude of each dimension.

  This function returns a non-negative value if successful; otherwise it returns a negative value.

- The H5Dget_create_plist function returns an identifier for a copy of the dataset creation property list for a dataset.

- The H5Pget_layout function returns the layout of the raw data for a dataset. Valid types are H5D_COMPACT, H5D_CONTIGUOUS, and H5D_CHUNKED.

- The H5Pget_chunk function retrieves the size of chunks for the raw data of a chunked layout dataset. The signature of this function is:

  ```
  int H5Pget_chunk ( hid_t plist, int max_ndims, hsize_t * dims )
  ```

  - The first parameter, *plist*, is the identifier of the property list to query.
  - The second parameter, *max_ndims*, is the size of the *dims* array.
  - The third parameter, *dims*, is the array to store the chunk dimensions

- The H5Pclose function terminates access to a property list. The signature of this function is:

  ```
  herr_t H5Pclose ( hid_t plist )
  ```

  where *plist* is the identifier of the property list to terminate access to.

Last Modified: August 27, 1999

17. Mounting Files

Mounting Files

HDF5 allows you to combine two or more HDF5 files in a manner similar to mounting files in UNIX. The group structure and metadata from one file appear as though they exist in another file. The following steps are involved:

36. Open the files.
37. Choose the "mount point" in the first file (parent). The "mount point" in HDF5 is a group (it can also be the root group).
38. Use the HDF5 API function H5Fmount to mount the second file (child) in the first one.
39. Work with the objects in the second file as if they were members of the "mount point" group in the first file. The previous contents of the "mount point" group are temporarily hidden.
40. Unmount the second file using the H5Funmount function when the work is done.

Programming Example

Description

In the following example we create one file with a group in it, and another file with a dataset. Mounting is used to access the dataset from the second file as a member of a group in the first file. The following picture illustrates this concept.

```
    ’FILE1’                                  ’FILE2’

--------------------                    --------------------
!                  !                    !                  !
!      /           !                    !       /          !
!       |          !                    !        |         !
!       |          !                    !        |         !
!       V          !                    !        V         !
!     --------     !                    !     ----------   !
!     ! Group !    !                    !     ! Dataset!   !
!     ---------    !                    !     ----------   !
!------------------!                    !------------------!
```

After mounting the second file, 'FILE2', under the group in the file, 'FILE1', the parent has the following structure:

```
                                'FILE1'

                           --------------------
                           !                  !
                           !      /           !
                           !       |          !
                           !       |          !
                           !       V          !
                           !     --------     !
                           !     ! Group !    !
                           !     ---------    !
                           !         |        !
                           !         |        !
                           !         V        !
                           !    -----------   !
                           !    ! Dataset !   !
                           !    !----------   !
                           !                  !
                           !------------------!
```

[h5_mount.c]

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include<hdf5.h>

#define FILE1 "mount1.h5"
#define FILE2 "mount2.h5"

#define RANK 2
#define NX 4
#define NY 5

int main(void)
{

   hid_t fid1, fid2, gid;  /* Files and group identifiers */
   hid_t did, tid, sid;    /* Dataset and datatype identifiers */

   herr_t status;
   hsize_t dims[] = {NX,NY}; /* Dataset dimensions */

   int i, j;
   int bm[NX][NY], bm_out[NX][NY]; /* Data buffers */

   /*
    * Initialization of buffer matrix "bm"
    */
   for(i =0; i < NX; i++) {
    for(j = 0; j < NY; j++)
      bm[i][j] = i + j;
   }

   /*
    * Create first file and a group in it.
    */
   fid1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
   gid = H5Gcreate(fid1, "/G", 0);
```

```
/*
 * Close group and file
 */
H5Gclose(gid);
H5Fclose(fid1);

/*
 * Create second file and dataset "D" in it.
 */
fid2 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
dims[0] = NX;
dims[1] = NY;
sid = H5Screate_simple(RANK, dims, NULL);
did = H5Dcreate(fid2, "D", H5T_NATIVE_INT, sid, H5P_DEFAULT);

/*
 * Write data to the dataset.
 */
status = H5Dwrite(did, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, bm);

/*
 * Close all identifiers.
 */
H5Sclose(sid);
H5Dclose(did);
H5Fclose(fid2);

/*
 * Reopen both files
 */
fid1 = H5Fopen(FILE1, H5F_ACC_RDONLY, H5P_DEFAULT);
fid2 = H5Fopen(FILE2, H5F_ACC_RDONLY, H5P_DEFAULT);

/*
 * Mount second file under G in the first file.
 */
H5Fmount(fid1, "/G", fid2, H5P_DEFAULT);

/*
 * Access dataset D in the first file under /G/D name.
 */
did = H5Dopen(fid1,"/G/D");
tid = H5Dget_type(did);
status = H5Dread(did, tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, bm_out);

/*
 * Print out the data.
 */
for(i=0; i < NX; i++){
 for(j=0; j < NY; j++)
     printf("  %d", bm_out[i][j]);
    printf("\n");
}

/*
 * Close all identifers
 */
H5Tclose(tid);
H5Dclose(did);

/*
 * Unmounting second file
 */
```

```
   H5Funmount(fid1, "/G");

   /*
    * Close both files
    */
   H5Fclose(fid1);
   H5Fclose(fid2);

   return 0;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The first part of the program creates a group in one file and creates and writes a dataset to another file.
- Both files are reopened with the H5F_ACC_RDONLY access flag since no objects will be modified. The child should be opened with H5F_ACC_RDWR if the dataset is to be modified.
- The second file is mounted in the first using the H5Fmount function.

  ```
  herr_t  H5Fmount(hid_t loc_id, const char *name, hid_t file_id, hid_t plist_id)
  ```

  - The first two arguments specify the location of the "mount point" (a group). In this example the "mount point" is a group "/G" in the file specified by its handle *fid1*. Since group G is in the root group of the first file, one can also use just "G" to identify it.

    Below is a description of another scenario:
    Suppose group G was a member of group D in the first file (*fid1*). Then mounting point G can be specified in two different ways:

    - *loc_id* is *fid1*
      name is "D/G"
    - *loc_id* is an identifier of the group "D"
      name is just "G"
  - The third argument is an identifier of the file which will be mounted. Only one file can be mounted per "mount point".
  - The fourth argument is an identifier of the property list to be used. Currently, only the default property list, H5P_DEFAULT, can be used.
- In this example we just read data from the dataset D. One can modify data also. If the dataset is modified while the file is mounted, it becomes modified in the original file too after the file is unmounted.
- The file is unmounted with the H5Funmount function:

  ```
  herr_t H5Funmount(hid_t loc_id, const char *name)
  ```

  Arguments to this function specify the location of the "mount point". In our example *loc_id* is an identifier of the first file, and *name* is the name of group G, "/G". One could also use "G" instead of "/G" since G is a member of the root group in the file *fid1*. Notice that H5Funmount does not close files. They are closed with the respective calls to the H5Fclose function. Closing the parent automatically unmounts the child.
- The h5dump utility cannot display files in memory, therefore no output of FILE1 after FILE2 was mounted is provided.

Last Modified: August 27, 1999

18. Iterating over Group Members

How to Iterate Over Group Members

This section discusses how to find names and object types of HDF5 group members.

The HDF5 Group interface has a function, H5Giterate, to iterate over the group members.

Operations on each group member can be performed during the iteration process. The operator function and its data are passed to the iterator as parameters. There are no restrictions on what kind of operations can be performed on group members during the iteration procedure.

The following steps are involved:

41. Write an operator function which will be used during the iteration process. The HDF5 library defines the operator function signature and return values.
42. Open the group to iterate through.
43. Use H5Giterate to iterate through the group or just a few members of the group.

Programming Example

Description

In this example we iterate through the members of the root group. The operator function displays the members' names and their object types. The object type can be a group, dataset, or named datatype. [`h5_iterate.c`]

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <hdf5.h>

#define FILE   "iterate.h5"
#define  FALSE 0

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

herr_t file_info(hid_t loc_id, const char *name, void *opdata);
                                     /* Operator function */
int
main(void) {
    hid_t                   file;            /* HDF5 File IDs           */
    hid_t                   dataset; /* Dataset ID                      */
    hid_t                   group;      /* Group ID             */
    hid_t                   sid;       /* Dataspace ID                  */
    hid_t                   tid;       /* Datatype ID                   */
    hsize_t                 dims[] = {SPACE1_DIM1};
    herr_t                  ret;             /* Generic return value             */

/* Compound datatype */
typedef struct s1_t {
    unsigned int a;
    unsigned int b;
    float c;
} s1_t;

    /* Create file */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid = H5Screate_simple(SPACE1_RANK, dims, NULL);

    /* Create a group */
    group=H5Gcreate(file,"Group1",-1);

    /* Close a group */
    ret = H5Gclose(group);

    /* Create a dataset  */
    dataset=H5Dcreate(file,"Dataset1",H5T_STD_U32LE,sid,H5P_DEFAULT);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create a datatype */
    tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));

    /* Insert fields */
    ret=H5Tinsert (tid, "a", HOFFSET(s1_t,a), H5T_NATIVE_INT);

    ret=H5Tinsert (tid, "b", HOFFSET(s1_t,b), H5T_NATIVE_INT);

    ret=H5Tinsert (tid, "c", HOFFSET(s1_t,c), H5T_NATIVE_FLOAT);

    /* Save datatype for later */
    ret=H5Tcommit (file, "Datatype1", tid);

    /* Close datatype */
    ret = H5Tclose(tid);

    /* Iterate through the file to see members of the root group */

    printf(" Objects in the root group are:\n");
    printf("\n");

    H5Giterate(file, "/", NULL, file_info, NULL);

    /* Close file */
    ret = H5Fclose(file);

    return 0;
}

/*
```

```
 * Operator function.
 */
herr_t file_info(hid_t loc_id, const char *name, void *opdata)
{
    H5G_stat_t statbuf;

    /*
     * Get type of the object and display its name and type.
     * The name of the object is passed to this function by
     * the Library. Some magic :-)
     */
    H5Gget_objinfo(loc_id, name, FALSE, &statbuf);
    switch (statbuf.type) {
    case H5G_GROUP:
         printf(" Object with name %s is a group \n", name);
         break;
    case H5G_DATASET:
         printf(" Object with name %s is a dataset \n", name);
         break;
    case H5G_TYPE:
         printf(" Object with name %s is a named datatype \n", name);
         break;
    default:
         printf(" Unable to identify an object ");
    }
    return 0;
 }

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The output of this program is:

```
 Objects in the root group are:

 Object with name Dataset1 is a dataset
 Object with name Datatype1 is a named datatype
 Object with name Group1 is a group
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Remarks

- The operator function in this example is called *file_info*. The signature of the operator function is as follows:

  ```
  herr_t *(H5G_operator_t) (hid_group_id, const char* name, void *operator_data)
  ```

  - The first parameter is a group identifier for the group being iterated over. It is passed to the operator by the iterator function, H5Giterate.

  - The second parameter is the name of the current object. The name is passed to the operator function by the HDF5 library.

  - The third parameter is the operator data. It is passed to and from the operator by the iterator, H5Giterate. The operator function in this example just prints the name and type of the current object and then exits. This information can also be used to open the object and perform different operations or queries. For example a named datatype object's name can be used to open the datatype and query its properties.

The operator return value defines the behavior of the iterator.

- A zero return value causes the iterator to continue, returning zero when all group members have been processed.
- A positive value causes the iterator to immediately return that value, indicating a short-circuit success. The iterator can be restarted at the next group member.
- A negative value causes the iterator to immediately return that value, indicating failure. The iterator can be restarted at the next group member.

In this example the operator function returns 0, which causes the iterator to continue and go through all group members.

- The function H5Gget_objinfo is used to determine the type of the object. It also returns the modification time, number of hard links, and some other information.

  The signature of this function is as follows:

  ```
  herr_t H5Gget_objinfo(hid_t loc_id, const char * name, hbool_t follow_link,
                        H5G_stat_t *statbuf)
  ```

  - The first two arguments specify the object by its location and name. This example uses the group identifier and name relative to the group to specify the object.
  - The third argument is a flag which indicates whether a symbolic link should be followed or not. A zero value indicates that information should be returned for the link itself, but not about the object it points to. The root group in this example does not have objects that are links, so this flag is not important for our example.
  - The fourth argument is a buffer to return information in. Type information is returned into the field "type" of the H5G_stat_t data structure (statbuf.type). Possible values are: H5G_GROUP, H5G_DATASET, H5G_TYPE, and H5G_LINK.

- The H5Giterate function has the following signature:

  ```
  int H5Giterate(hid_t loc_id, const char *name , idx,
                 H5G_operator_t operator, void * operator_data)
  ```

  - The first parameter is the group for the group being iterated over.
  - The second parameter is the group name.
  - The third parameter specifies that iteration begins with the *idx* object in the group and the next element to be processed is returned in the *idx* parameter. In our example NULL is used to start at the first group member. Since no stopping point is returned in this case, the iterator cannot be restarted if one of the calls to its operator returns a non-zero value.
  - The fourth parameter is an operator function.
  - The fifth argument is the operator data. We used NULL since no data was passed to and from the operator.

Last Modified: August 27, 1999

Utilities (h5dump, h5ls)

The h5dump and h5ls utilities can be used to examine the contents of an hdf5 file.

h5dump

```
h5dump [-h] [-bb] [-header] [-a ] [-d <names>] [-g <names>]
       [-l <names>] [-t <names>] <file>

  -h             Print information on this command.
  -bb            Display the content of the boot block. The default is not to display.
  -header        Display header only; no data is displayed.
  -a <names>     Display the specified attribute(s).
  -d <names>     Display the specified dataset(s).
  -g <names>     Display the specified group(s) and all the members.
  -l <names>     Displays the value(s) of the specified soft link(s).
  -t <names>     Display the specified named data type(s).

  <names> is one or more appropriate object names.
```

h5ls

```
h5ls [OPTIONS] FILE [OBJECTS...]

   OPTIONS
      -h, -?, --help   Print a usage message and exit
      -d, --dump       Print the values of datasets
      -f, --full       Print full path names instead of base names
      -l, --label      Label members of compound datasets
      -r, --recursive  List all groups recursively, avoiding cycles
      -s, --string     Print 1-byte integer datasets as ASCII
      -wN, --width=N   Set the number of columns of output
      -v, --verbose    Generate more verbose output
      -V, --version    Print version number and exit
   FILE
      The file name may include a printf(3C) integer format such as
      "%05d" to open a file family.
   OBJECTS
      The names of zero or more objects about which information should be
      displayed.  If a group is mentioned then information about each of its
      members is displayed.  If no object names are specified then
      information about all of the objects in the root group is displayed.
```

Last Modified: July 30, 1999

Glossary

ATTRIBUTE

An HDF5 attribute is a small dataset that can be used to describe the nature and/or the intended usage of the object it is attached to.

BOOT BLOCK

HDF5 files are composed of a "boot block" describing information required to portably access files on multiple platforms, followed by information about the groups in a file and the datasets in the file. The boot block contains information about the size of offsets and lengths of objects, the number of entries in symbol tables (used to store groups) and additional version information for the file.

DATASET

An HDF5 dataset is a multi-dimensional array of data elements, together with supporting metadata.

DATASPACE

An HDF5 dataspace is an object that describes the dimensionality of the data array. A dataspace is either a regular N-dimensional array of data points, called a simple dataspace, or a more general collection of data points organized in another manner, called a complex dataspace.

DATA TYPE

An HDF5 Data Type is an object that describes the type of the element in an HDF5 multi-dimensional array. There are two categories of datatypes: atomic and compound data types. An atomic type is a type which cannot be decomposed into smaller units at the API level. A compound is a collection of one or more atomic types or small arrays of such types.

DATASET CREATION PROPERTY LIST

The Dataset Creation Property List contains information on how raw data is organized on disk and how the raw data is compressed. The dataset API partitions these terms by layout, compression, and external storage:

Layout:

- H5D_COMPACT: Data is small and can be stored in object header (not implemented yet). This eliminates disk seek/read requests.
- H5D_CONTIGUOUS: (**default**) The data is large, non-extendible, non-compressible, non-sparse, and can be stored externally.
- H5D_CHUNKED: The data is large and can be extended in any dimension. It is partitioned into chunks so each chunk is the same logical size.

Compression: (gzip compression)
External Storage Properties: The data must be contiguous to be stored externally. It allows you to store the data in one or more non-HDF5 files.

DATA TRANSFER PROPERTY LIST

The data transfer property list is used to control various aspects of the I/O, such as caching hints or collective I/O information.

DDL

DDL is a Data Description Language that describes HDF5 objects in Backus-Naur Form.

FILE ACCESS MODES

The file access modes determine whether an existing file will be overwritten. All newly created files are opened for both reading and writing. Possible values are:

```
H5F_ACC_RDWR:   Allow read and write access to file.
H5F_ACC_RDONLY: Allow read-only access to file.
H5F_ACC_TRUNC:  Truncate file, if it already exists, erasing all data
                previously stored in the file.
H5F_ACC_EXCL:   Fail if file already exists.
H5F_ACC_DEBUG:  Print debug information.
H5P_DEFAULT:    Apply default file access and creation properties.
```

FILE ACCESS PROPERTY LIST

File access property lists are used to control different methods of performing I/O on files:

Unbuffered I/O: Local permanent files can be accessed with the functions described in Section 2 of the Posix manual, namely open(), lseek(), read(), write(), and close().
Buffered I/O: Local permanent files can be accessed with the functions declared in the stdio.h header file, namely fopen(), fseek(), fread(), fwrite(), and fclose().
Memory I/O: Local temporary files can be created and accessed directly from memory without ever creating permanent storage. The library uses malloc() and free() to create storage space for the file
Parallel Files using MPI I/O: This driver allows parallel access to a file through the MPI I/O library. The parameters which can be modified are the MPI communicator, the info object, and the access mode. The communicator and info object are saved and then passed to MPI_File_open() during file creation or open. The access_mode controls the kind of parallel access the application intends.
Data Alignment: Sometimes file access is faster if certain things are aligned on file blocks. This can be controlled by setting alignment properties of a file access property list with the H5Pset_alignment() function.

FILE CREATION PROPERTY LIST

The file creation property list is used to control the file metadata. The parameters that can be modified are:

User-Block Size: The "user-block" is a fixed length block of data located at the beginning of the file which is ignored by the HDF5 library and may be used to store any data information found to be useful to applications.
Offset and Length Sizes: The number of bytes used to store the offset and length of objects in the HDF5 file can be controlled with this parameter. Symbol Table Parameters: The size of symbol table B-trees can be controlled by setting the 1/2 rank and 1/2 node size parameters of the B-tree.
Indexed Storage Parameters: The size of indexed storage B-trees can be controlled by setting the 1/2 rank and 1/2 node size parameters of the B-tree.

GROUP

A Group is a structure containing zero or more HDF5 objects, together with supporting metadata. The two primary HDF5 objects are datasets and groups.

HDF5

HDF5 is an abbreviation for Hierarchical Data Format Version 5. This file format is intended to make it easy to write and read scientific data

- by including the information needed to understand the data within the file
- by providing a library of C, FORTRAN, and other language programs that reduce the work required to provide efficient writing and reading - even with parallel IO

HDF5 FILE

An HDF5 file is a container for storing grouped collections of multi-dimensional arrays containing scientific data.

H5DUMP

h5dump is an HDF5 tool that describes the HDF5 file contents in DDL.

HYPERSLAB

A hyperslab is a portion of a dataset. A hyperslab selection can be a logically contiguous collection of points in a dataspace, or it can be a regular pattern of points or blocks in a dataspace.

NAMES

HDF5 object names are a slash-separated list of components. A name which begins with a slash is an absolute name which is accessed beginning with the root group of the file while all other relative names are accessed beginning with the specified group.

PARALLEL I/O (HDF5)

The parallel I/O version of HDF5 supports parallel file access using MPI (Message Passing Interface).

THREADSAFE (HDF5)

A "thread-safe" version of HDF-5 (TSHDF5) is one that can be called from any thread of a multi-threaded program. Any calls to HDF can be made in any order, and each individual HDF call will perform correctly. A calling program does not have to explicitly lock the HDF library in order to do I/O. Applications programmers may assume that the TSHDF5 guarantees the following:

- the HDF-5 library does not create or destroy threads.
- the HDF-5 library uses modest amounts of per-thread private memory.
- the HDF-5 library only locks/unlocks it's own locks (no locks are passed in or returned from HDF), and the internal locking is guaranteed to be deadlock free.

These properties mean that the TSHDF5 library will not interfere with an application's use of threads. A TSHDF5 library is the same library as regular HDF-5 library, with additional code to synchronize access to the HDF-5 library's internal data structures.

Last Modified: September 1, 1999

References

- **HDF Home Page:** `http://hdf.ncsa.uiuc.edu/`
- **HDF5 Home Page and Documentation:** `http://hdf.ncsa.uiuc.edu/HDF5/`
- **HDF5 DDL:** `http://hdf.ncsa.uiuc.edu/HDF5/doc/ddl.html`
- **Introduction to HDF5:** `http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html`
  Overview of HDF5 with example programs.
- **Other Miscellaneous HDF5 Example Programs:** `http://hdf.ncsa.uiuc.edu/training/other-ex5/`

Last Modified: September 28, 1999

Example Programs from This Tutorial

As mentioned in the introduction, this tutorial was designed to be used online in an interactive mode. A tar file containing the example programs supporting this tutorial can be obtained from the following location on the HDF website at NCSA:

```
http://hdf.ncsa.uiuc.edu/training/other-ex5/examples.tar
```

Last Modified: November 10, 1999

A User's Guide for HDF5

Release 1.2
October 1999

Hierarchical Data Format (HDF) Group
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Last modified: 13 October 1999

A User’s Guide for HDF5

1. The File Interface (H5F)

1.1. Introduction

HDF5 files are composed of a "boot block" describing information required to portably access files on multiple platforms, followed by information about the groups in a file and the datasets in the file. The boot block contains information about the size of offsets and lengths of objects, the number of entries in symbol tables (used to store groups) and additional version information for the file.

1.2. File access modes

The HDF5 library assumes that all files are implicitly opened for read access at all times. Passing the `H5F_ACC_RDWR` parameter to `H5Fopen()` allows write access to a file also. `H5Fcreate()` assumes write access as well as read access, passing `H5F_ACC_TRUNC` forces the truncation of an existing file, otherwise H5Fcreate will fail to overwrite an existing file.

1.3. Creating, Opening, and Closing Files

Files are created with the `H5Fcreate()` function, and existing files can be accessed with `H5Fopen()`. Both functions return an object ID which should be eventually released by calling `H5Fclose()`.

`hid_t H5Fcreate (const char *`*`name`*`, uintn `*`flags`*`, hid_t `*`create_properties`*`, hid_t `*`access_properties`*`)`

> This function creates a new file with the specified name in the current directory. The file is opened with read and write permission, and if the `H5F_ACC_TRUNC` flag is set, any current file is truncated when the new file is created. If a file of the same name exists and the `H5F_ACC_TRUNC` flag is not set (or the `H5F_ACC_EXCL` bit is set), this function will fail. Passing `H5P_DEFAULT` for the creation and/or access property lists uses the library's default values for those properties. Creating and changing the values of a property list is documented further below. The return value is an ID for the open file and it should be closed by calling `H5Fclose()` when it's no longer needed. A negative value is returned for failure.

`hid_t H5Fopen (const char *`*`name`*`, uintn `*`flags`*`, hid_t `*`access_properties`*`)`

> This function opens an existing file with read permission and write permission if the `H5F_ACC_RDWR` flag is set. The *access_properties* is a file access property list ID or `H5P_DEFAULT` for the default I/O access parameters. Creating and changing the parameters for access property lists is documented further below. Files which are opened more than once return a unique identifier for each `H5Fopen()` call and can be accessed through all file IDs. The return value is an ID for the open file and it should be closed by calling `H5Fclose()` when it's no longer needed. A negative value is returned for failure.

`herr_t H5Fclose (hid_t `*`file_id`*`)`

> This function releases resources used by a file which was opened by `H5Fcreate()` or `H5Fopen()`. After closing a file the *file_id* should not be used again. This function returns zero for success or a negative value for failure.

`herr_t H5Fflush (hid_t` *`object_id`*`, H5F_scope_t` *`scope`*`)`

> This function will cause all buffers associated with a file to be immediately flushed to the file. The *object_id* can be any object which is associated with a file, including the file itself. *scope* specifies whether the flushing action is to be global or local.

1.4. File Property Lists

Additional parameters to `H5Fcreate()` or `H5Fopen()` are passed through property list objects, which are created with the `H5Pcreate()` function. These objects allow many parameters of a file's creation or access to be changed from the default values. Property lists are used as a portable and extensible method of modifying multiple parameter values with simple API functions. There are two kinds of file-related property lists, namely file creation properties and file access properties.

1.4.1. File Creation Properties

File creation property lists apply to `H5Fcreate()` only and are used to control the file meta-data which is maintained in the boot block of the file. The parameters which can be modified are:

User-Block Size

> The "user-block" is a fixed length block of data located at the beginning of the file which is ignored by the HDF5 library and may be used to store any data information found to be useful to applications. This value may be set to any power of two equal to 512 or greater (i.e. 512, 1024, 2048, etc). This parameter is set and queried with the `H5Pset_userblock()` and `H5Pget_userblock()` calls.

Offset and Length Sizes

> The number of bytes used to store the offset and length of objects in the HDF5 file can be controlled with this parameter. Values of 2, 4 and 8 bytes are currently supported to allow 16-bit, 32-bit and 64-bit files to be addressed. These parameters are set and queried with the `H5Pset_sizes()` and `H5Pget_sizes()` calls.

Symbol Table Parameters

> The size of symbol table B-trees can be controlled by setting the 1/2 rank and 1/2 node size parameters of the B-tree. These parameters are set and queried with the `H5Pset_sym_k()` and `H5Pget_sym_k()` calls.

Indexed Storage Parameters

> The size of indexed storage B-trees can be controlled by setting the 1/2 rank and 1/2 node size parameters of the B-tree. These parameters are set and queried with the `H5Pset_istore_k()` and `H5Pget_istore_k()` calls.

1.4.2. File Access Property Lists

File access property lists apply to `H5Fcreate()` or `H5Fopen()` and are used to control different methods of performing I/O on files.

Unbuffered I/O

> Local permanent files can be accessed with the functions described in Section 2 of the Posix manual, namely `open()`, `lseek()`, `read()`, `write()`, and `close()`. The `lseek64()` function is used on operating systems that support it. This driver is enabled and configured with `H5Pset_sec2()`, and queried with `H5Pget_sec2()`.

Buffered I/O

Local permanent files can be accessed with the functions declared in the `stdio.h` header file, namely `fopen()`, `fseek()`, `fread()`, `fwrite()`, and `fclose()`. The `fseek64()` function is used on operating systems that support it. This driver is enabled and configured with `H5Pset_stdio()`, and queried with `H5Pget_stdio()`.

Memory I/O

Local temporary files can be created and accessed directly from memory without ever creating permanent storage. The library uses `malloc()` and `free()` to create storage space for the file. The total size of the file must be small enough to fit in virtual memory. The name supplied to `H5Fcreate()` is irrelevant, and `H5Fopen()` will always fail.

Parallel Files using MPI I/O

This driver allows parallel access to a file through the MPI I/O library. The parameters which can be modified are the MPI communicator, the info object, and the access mode. The communicator and info object are saved and then passed to `MPI_File_open()` during file creation or open. The access_mode controls the kind of parallel access the application intends. (Note that it is likely that the next API revision will remove the access_mode parameter and have access control specified via the raw data transfer property list of `H5Dread()` and `H5Dwrite()`.) These parameters are set and queried with the `H5Pset_mpi()` and `H5Pget_mpi()` calls.

Data Alignment

Sometimes file access is faster if certain things are aligned on file blocks. This can be controlled by setting alignment properties of a file access property list with the `H5Pset_alignment()` function. Any allocation request at least as large as some threshold will be aligned on an address which is a multiple of some number.

1.5. Examples of using file property lists

1.5.1. Example of using file creation property lists

This following example shows how to create a file with 64-bit object offsets and lengths:

```
hid_t create_plist;
hid_t file_id;

create_plist = H5Pcreate(H5P_FILE_CREATE);
H5Pset_sizes(create_plist, 8, 8);

file_id = H5Fcreate("test.h5", H5F_ACC_TRUNC,
                    create_plist, H5P_DEFAULT);
.
.
.
H5Fclose(file_id);
```

1.5.2. Example of using file creation plist

This following example shows how to open an existing file for independent datasets access by MPI parallel I/O:

```
hid_t access_plist;

hid_t file_id;

access_plist = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_mpi(access_plist, MPI_COMM_WORLD, MPI_INFO_NULL);

 /* H5Fopen must be called collectively */
file_id = H5Fopen("test.h5", H5F_ACC_RDWR, access_plist);
.
.
.
 /* H5Fclose must be called collectively */
H5Fclose(file_id);
```

1.6. Low-level File Drivers

HDF5 is able to access its address space through various types of low-level *file drivers*. For instance, an address space might correspond to a single file on a Unix file system, multiple files on a Unix file system, multiple files on a parallel file system, or a block of memory within the application. Generally, an HDF5 address space is referred to as an "HDF5 file" regardless of how the space is organized at the storage level.

1.6.1. Unbuffered Permanent Files

The *sec2* driver uses functions from section 2 of the Posix manual to access files stored on a local file system. These are the `open()`, `close()`, `read()`, `write()`, and `lseek()` functions. If the operating system supports `lseek64()` then it is used instead of `lseek()`. The library buffers meta data regardless of the low-level driver, but using this driver prevents data from being buffered again by the lowest layers of the HDF5 library.

`H5F_driver_t H5Pget_driver (hid_t` *`access_properties`*`)`

This function returns the constant `H5F_LOW_SEC2` if the *sec2* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_sec2 (hid_t` *`access_properties`*`)`

The file access properties are set to use the *sec2* driver. Any previously defined driver properties are erased from the property list. Additional parameters may be added to this function in the future.

`herr_t H5Pget_sec2 (hid_t` *`access_properties`*`)`

If the file access property list is set to the *sec2* driver then this function returns zero; otherwise it returns a negative value. In the future, additional arguments may be added to this function to match those added to `H5Pset_sec2()`.

1.6.2. Buffered Permanent Files

The *stdio* driver uses the functions declared in the `stdio.h` header file to access permanent files in a local file system. These are the `fopen()`, `fclose()`, `fread()`, `fwrite()`, and `fseek()` functions. If the operating system supports `fseek64()` then it is used instead of `fseek()`. Use of this driver introduces an additional layer of buffering beneath the HDF5 library.

`H5F_driver_t H5Pget_driver(hid_t` *`access_properties`*`)`

> This function returns the constant `H5F_LOW_STDIO` if the *stdio* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_stdio (hid_t` *`access_properties`*`)`

> The file access properties are set to use the *stdio* driver. Any previously defined driver properties are erased from the property list. Additional parameters may be added to this function in the future.

`herr_t H5Pget_stdio (hid_t` *`access_properties`*`)`

> If the file access property list is set to the *stdio* driver then this function returns zero; otherwise it returns a negative value. In the future, additional arguments may be added to this function to match those added to `H5Pset_stdio()`.

1.6.3. Buffered Temporary Files

The *core* driver uses `malloc()` and `free()` to allocated space for a file in the heap. Reading and writing to a file of this type results in mem-to-mem copies instead of disk I/O and as a result is somewhat faster. However, the total file size must not exceed the amount of available virtual memory, and only one HDF5 file handle can access the file (because the name of such a file is insignificant and `H5Fopen()` always fails).

`H5F_driver_t H5Pget_driver (hid_t` *`access_properties`*`)`

> This function returns the constant `H5F_LOW_CORE` if the *core* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_core (hid_t` *`access_properties`*`, size_t` *`block_size`*`)`

> The file access properties are set to use the *core* driver and any previously defined driver properties are erased from the property list. Memory for the file will always be allocated in units of the specified *block_size*. Additional parameters may be added to this function in the future.

`herr_t H5Pget_core (hid_t` *`access_properties`*`, size_t *`*`block_size`*`)`

> If the file access property list is set to the *core* driver then this function returns zero and *block_size* is set to the block size used for the file; otherwise it returns a negative value. In the future, additional arguments may be added to this function to match those added to `H5Pset_core()`.

1.6.4. Parallel Files

This driver uses MPI I/O to provide parallel access to a file.

`H5F_driver_t H5Pget_driver (hid_t` *`access_properties`*`)`

> This function returns the constant `H5F_LOW_MPI` if the *mpi* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_mpi (hid_t` *`access_properties`*`, MPI_Comm` *`comm`*`, MPI_info` *`info`*`)`

> The file access properties are set to use the *mpi* driver and any previously defined driver properties are erased from the property list. Additional parameters may be added to this function in the future.

`herr_t H5Pget_mpi (hid_t` *`access_properties`*`, MPI_Comm *`*`comm`*`, MPI_info *`*`info`*`)`

> If the file access property list is set to the *mpi* driver then this function returns zero and *comm*, and *info* are set to the values stored in the property list; otherwise the function returns a negative value. In the future, additional arguments may be added to this function to match those added to `H5Pset_mpi()`.

1.6.5. File Families

A single HDF5 address space may be split into multiple files which, together, form a file family. Each member of the family must be the same logical size although the size and disk storage reported by `ls`(1) may be substantially smaller. The name passed to `H5Fcreate()` or `H5Fopen()` should include a `printf(3c)` style integer format specifier which will be replaced with the family member number (the first family member is zero).

Any HDF5 file can be split into a family of files by running the file through `split`(1) and numbering the output files. However, because HDF5 is lazy about extending the size of family members, a valid file cannot generally be created by concatenation of the family members. Additionally, `split` and `cat` don’t attempt to generate files with holes. The `h5repart` program can be used to repartition an HDF5 file or family into another file or family and preserves holes in the files.

`h5repart` [`-v`] [`-b` *block_size*[*suffix*]] [`-m` *member_size*[*suffix*]] *source destination*

> This program repartitions an HDF5 file by copying the source file or family to the destination file or family preserving holes in the underlying Unix files. Families are used for the source and/or destination if the name includes a `printf`-style integer format such as "%d". The `-v` switch prints input and output file names on the standard error stream for progress monitoring, `-b` sets the I/O block size (the default is 1kB), and `-m` sets the output member size if the destination is a family name (the default is 1GB). The block and member sizes may be suffixed with the letters `g`, `m`, or `k` for GB, MB, or kB respectively.

`H5F_driver_t H5Pget_driver (hid_t` *`access_properties`*`)`

> This function returns the constant `H5F_LOW_FAMILY` if the *family* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_family (hid_t` *`access_properties`*`, hsize_t` *`memb_size`*`, hid_t` *`member_properties`*`)`

The file access properties are set to use the *family* driver and any previously defined driver properties are erased from the property list. Each member of the file family will use *member_properties* as its file access property list. The *memb_size* argument gives the logical size in bytes of each family member but the actual size could be smaller depending on whether the file contains holes. The member size is only used when creating a new file or truncating an existing file; otherwise the member size comes from the size of the first member of the family being opened. Note: if the size of the `off_t` type is four bytes then the maximum family member size is usually 2^31-1 because the byte at offset 2,147,483,647 is generally inaccessable. Additional parameters may be added to this function in the future.

`herr_t H5Pget_family (hid_t` *`access_properties`*`, hsize_t *`*`memb_size`*`, hid_t *`*`member_properties`*`)`

If the file access property list is set to the *family* driver then this function returns zero; otherwise the function returns a negative value. On successful return, *access_properties* will point to a copy of the member access property list which should be closed by calling `H5Pclose()` when the application is finished with it. If *memb_size* is non-null then it will contain the logical size in bytes of each family member. In the future, additional arguments may be added to this function to match those added to `H5Pset_family()`.

1.6.6. Split Meta/Raw Files

On occasion, it might be useful to separate meta data from raw data. The *split* driver does this by creating two files: one for meta data and another for raw data. The application provides a base file name to `H5Fcreate()` or `H5Fopen()` and this driver appends a file extension which defaults to ".meta" for the meta data file and ".raw" for the raw data file. Each file can have its own file access property list which allows, for instance, a split file with meta data stored with the *core* driver and raw data stored with the *sec2* driver.

`H5F_driver_t H5Pget_driver (hid_t` *`access_properties`*`)`

This function returns the constant `H5F_LOW_SPLIT` if the *split* driver is defined as the low-level driver for the specified access property list.

`herr_t H5Pset_split (hid_t` *`access_properties`*`, const char *`*`meta_extension`*`, hid_t` *`meta_properties`*`, const char *`*`raw_extension`*`, hid_t` *`raw_properties`*`)`

The file access properties are set to use the *split* driver and any previously defined driver properties are erased from the property list. The meta file will have a name which is formed by adding *meta_extension* (or ".meta") to the end of the base name and will be accessed according to the *meta_properties*. The raw file will have a name which is formed by appending *raw_extension* (or ".raw") to the base name and will be accessed according to the *raw_properties*. Additional parameters may be added to this function in the future.

`herr_t H5Pget_split (hid_t` *`access_properties`*`, size_t` *`meta_ext_size`*`, const char *`*`meta_extension`*`, hid_t` *`meta_properties`*`, size_t` *`raw_ext_size`*`, const char *`*`raw_extension`*`, hid_t *`*`raw_properties`*`)`

If the file access property list is set to the *split* driver then this function returns zero; otherwise the function returns a negative value. On successful return, *meta_properties* and *raw_properties* will point to copies of the meta and raw access property lists which should be closed by calling `H5Pclose()` when the application is finished with them, but if the meta and/or raw file has no property list then a negative value is returned for that property list handle. Also, if *meta_extension* and/or *raw_extension* are non-null pointers, at most *meta_ext_size* or *raw_ext_size* characters of the meta or raw file name extension will be copied to the specified buffer. If the actual name is longer than what was requested then the result will not be null terminated (similar to `strncpy()`). In the future, additional arguments may be added to this function to match those added to `H5Pset_split()`.

Last modified: 14 October 1999

2. The Dataset Interface (H5D)

2.1. Introduction

The purpose of the dataset interface is to provide a mechanism to describe properties of datasets and to transfer data between memory and disk. A dataset is composed of a collection of raw data points and four classes of meta data to describe the data points. The interface is hopefully designed in such a way as to allow new features to be added without disrupting current applications that use the dataset interface.

The four classes of meta data are:

Constant Meta Data

> Meta data that is created when the dataset is created and exists unchanged for the life of the dataset. For instance, the datatype of stored array elements is defined when the dataset is created and cannot be subsequently changed.

Persistent Meta Data

> Meta data that is an integral and permanent part of a dataset but can change over time. For instance, the size in any dimension can increase over time if such an increase is allowed when the dataset was created.

Memory Meta Data

> Meta data that exists to describe how raw data is organized in the application's memory space. For instance, the data type of elements in an application array might not be the same as the datatype of those elements as stored in the HDF5 file.

Transport Meta Data

> Meta data that is used only during the transfer of raw data from one location to another. For instance, the number of processes participating in a collective I/O request or hints to the library to control caching of raw data.

Each of these classes of meta data is handled differently by the library although the same API might be used to create them. For instance, the datatype exists as constant meta data and as memory meta data; the same API (the `H5T` API) is used to manipulate both pieces of meta data but they're handled by the dataset API (the `H5D` API) in different manners.

2.2. Storage Layout Properties

The dataset API partitions these terms on three orthogonal axes (layout, compression, and external storage) and uses a *dataset creation property list* to hold the various settings and pass them through the dataset interface. This is similar to the way HDF5 files are created with a file creation property list. A dataset creation property list is always derived from the default dataset creation property list (use `H5Pcreate()` to get a copy of the default property list) by modifying properties with various `H5Pset_`*`property`*`()` functions.

`herr_t H5Pset_layout (hid_t` *`plist_id`*`, H5D_layout_t` *`layout`*`)`

> The storage layout is a piece of constant meta data that describes what method the library uses to organize the raw data on disk. The default layout is contiguous storage.

`H5D_COMPACT` ***(Not yet implemented.)***

The raw data is presumably small and can be stored directly in the object header. Such data is non-extendible, non-compressible, non-sparse, and cannot be stored externally. Most of these restrictions are arbitrary but are enforced because of the small size of the raw data. Storing data in this format eliminates the disk seek/read request normally necessary to read raw data.

`H5D_CONTIGUOUS`

The raw data is large, non-extendible, non-compressible, non-sparse, and can be stored externally. This is the default value for the layout property. The term *large* means that it may not be possible to hold the entire dataset in memory. The non-compressibility is a side effect of the data being large, contiguous, and fixed-size at the physical level, which could cause partial I/O requests to be extremely expensive if compression were allowed.

`H5D_CHUNKED`

The raw data is large and can be extended in any dimension at any time (provided the data space also allows the extension). It may be sparse at the chunk level (each chunk is non-sparse, but there might only be a few chunks) and each chunk can be compressed and/or stored externally. A dataset is partitioned into chunks so each chunk is the same logical size. The chunks are indexed by a B-tree and are allocated on demand (although it might be useful to be able to preallocate storage for parts of a chunked array to reduce contention for the B-tree in a parallel environment). The chunk size must be defined with `H5Pset_chunk()`.

others...

Other layout types may be defined later without breaking existing code. However, to be able to correctly read or modify data stored with one of these new layouts, the application will need to be linked with a new version of the library. This happens automatically on systems with dynamic linking.

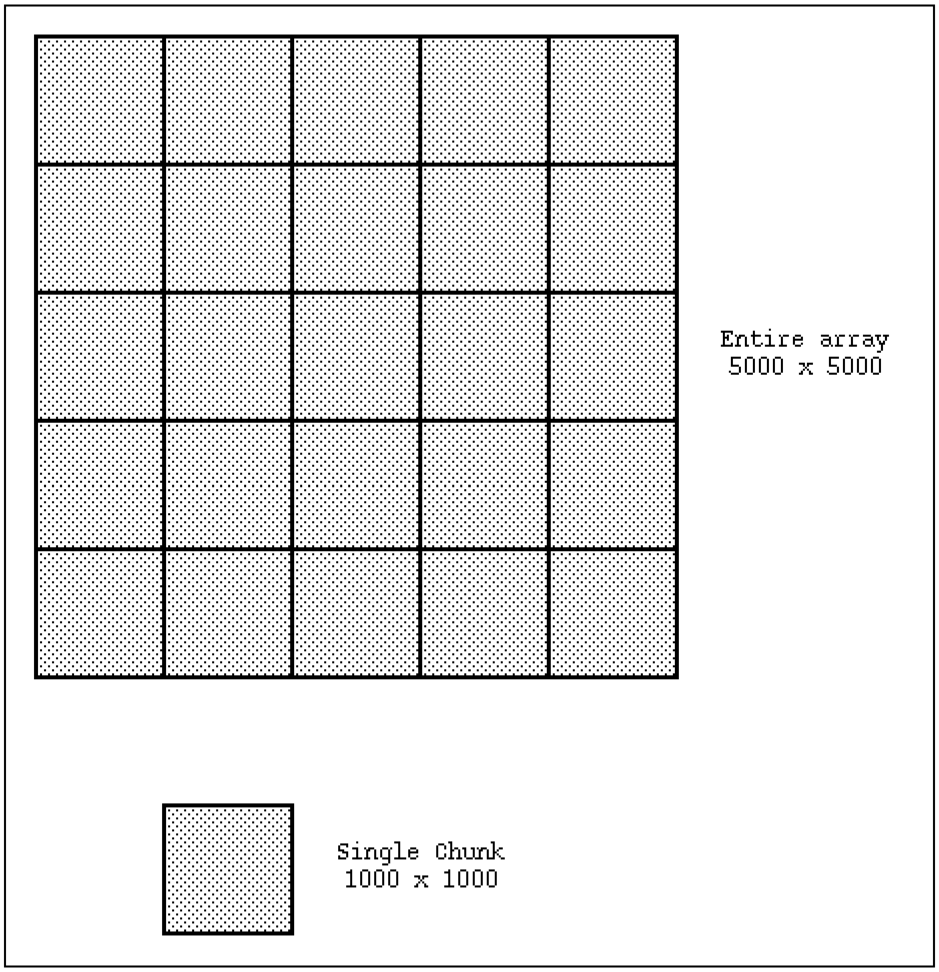
Once the general layout is defined, the user can define properties of that layout. Currently, the only layout that has user-settable properties is the `H5D_CHUNKED` layout, which needs to know the dimensionality and chunk size.

`herr_t H5Pset_chunk (hid_t` *`plist_id`*`, int` *`ndims`*`, hsize_t` *`dim`*`[])`

This function defines the logical size of a chunk for chunked layout. If the layout property is set to `H5D_CHUNKED` and the chunk size is set to *dim*. The number of elements in the *dim* array is the dimensionality, *ndims*. One need not call `H5Dset_layout()` when using this function since the chunked layout is implied.

Example: Chunked Storage

This example shows how a two-dimensional dataset is partitioned into chunks. The library can manage file memory by moving the chunks around, and each chunk could be compressed. The chunks are allocated in the file on demand when data is written to the chunk.



```
size_t hsize[2] = {1000, 1000};
plist = H5Pcreate (H5P_DATASET_CREATE);
H5Pset_chunk (plist, 2, size);
```

Although it is most efficient if I/O requests are aligned on chunk boundaries, this is not a constraint. The application can perform I/O on any set of data points as long as the set can be described by the data space. The set on which I/O is performed is called the *selection*.

2.3. Compression Properties

Some types of storage layout allow data compression which is defined by the functions described here. **Compression is not implemented yet.**

`herr_t H5Pset_compression (hid_t` *`plist_id`*`, H5Z_method_t` *`method`*`)`

`H5Z_method_t H5Pget_compression (hid_t` *`plist_id`*`)`

These functions set and query the compression method that is used to compress the raw data of a dataset. The *plist_id* is a dataset creation property list. The possible values for the compression method are:

`H5Z_NONE`

This is the default and specifies that no compression is to be performed.

`H5Z_DEFLATE`

This specifies that a variation of the Lempel-Ziv 1977 (LZ77) encoding is used, the same encoding used by the free GNU `gzip` program.

`herr_t H5Pset_deflate (hid_t` *`plist_id`*`, int` *`level`*`)`

`int H5Pget_deflate (hid_t` *`plist_id`*`)`

These functions set or query the deflate level of dataset creation property list *plist_id*. The `H5Pset_deflate()` sets the compression method to `H5Z_DEFLATE` and sets the compression level to some integer between one and nine (inclusive). One results in the fastest compression while nine results in the best compression ratio. The default value is six if `H5Pset_deflate()` isn't called. The `H5Pget_deflate()` returns the compression level for the deflate method, or negative if the method is not the deflate method.

2.4. External Storage Properties

Some storage formats may allow storage of data across a set of non-HDF5 files. Currently, only the `H5D_CONTIGUOUS` storage format allows external storage. A set segments (offsets and sizes) in one or more files is defined as an external file list, or *EFL*, and the contiguous logical addresses of the data storage are mapped onto these segments.

`herr_t H5Pset_external (hid_t` *`plist`*`, const char *`*`name`*`, off_t` *`offset`*`, hsize_t` *`size`*`)`

This function adds a new segment to the end of the external file list of the specified dataset creation property list. The segment begins a byte *offset* of file *name* and continues for *size* bytes. The space represented by this segment is adjacent to the space already represented by the external file list. The last segment in a file list may have the size `H5F_UNLIMITED`, in which case the external file may be of unlimited size and no more files can be added to the external files list.

`int H5Pget_external_count (hid_t` *`plist`*`)`

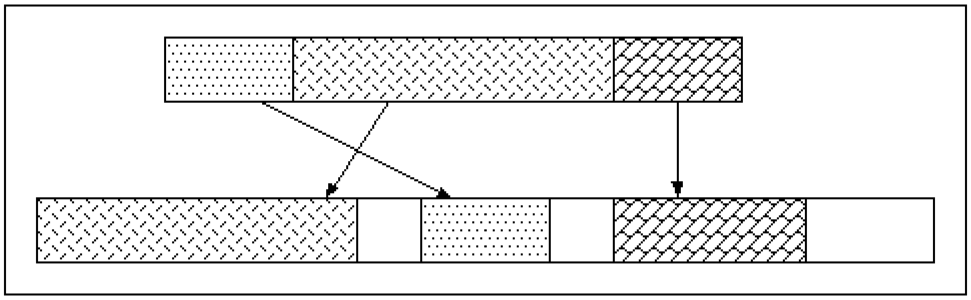
Calling this function returns the number of segments in an external file list. If the dataset creation property list has no external data then zero is returned.

`herr_t H5Pget_external (hid_t` *`plist`*`, int` *`idx`*`, size_t` *`name_size`*`, char *`*`name`*`, off_t *`*`offset`*`, hsize_t *`*`size`*`)`

This is the counterpart for the `H5Pset_external()` function. Given a dataset creation property list and a zero-based index into that list, the file name, byte offset, and segment size are returned through non-null arguments. At most *name_size* characters are copied into the *name* argument which is not null terminated if the file name is longer than the supplied name buffer (this is similar to `strncpy()`).

Example: Multiple Segments

This example shows how a contiguous, one-dimensional dataset is partitioned into three parts and each of those parts is stored in a segment of an external file. The top rectangle represents the logical address space of the dataset while the bottom rectangle represents an external file.

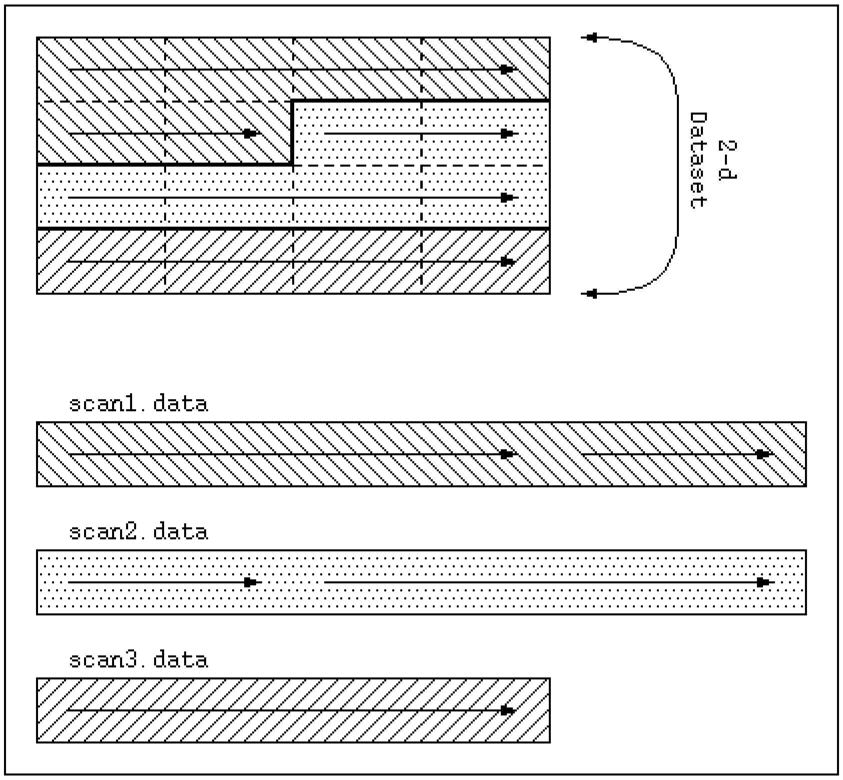


```
plist = H5Pcreate (H5P_DATASET_CREATE);
H5Pset_external (plist, "velocity.data", 3000, 1000);
H5Pset_external (plist, "velocity.data", 0, 2500);
H5Pset_external (plist, "velocity.data", 4500, 1500);
```

One should note that the segments are defined in order of the logical addresses they represent, not their order within the external file. It would also have been possible to put the segments in separate files. Care should be taken when setting up segments in a single file since the library doesn't automatically check for segments that overlap.

Example: Multi-Dimensional

This example shows how a contiguous, two-dimensional dataset is partitioned into three parts and each of those parts is stored in a separate external file. The top rectangle represents the logical address space of the dataset while the bottom rectangles represent external files.



```
plist = H5Pcreate (H5P_DATASET_CREATE);
H5Pset_external (plist, "scan1.data", 0, 24);
H5Pset_external (plist, "scan2.data", 0, 24);
H5Pset_external (plist, "scan3.data", 0, 16);
```

The library maps the multi-dimensional array onto a linear address space like normal, and then maps that address space into the segments defined in the external file list.

The segments of an external file can exist beyond the end of the file. The library reads that part of a segment as zeros. When writing to a segment that exists beyond the end of a file, the file is automatically extended. Using this feature, one can create a segment (or set of segments) which is larger than the current size of the dataset, which allows to dataset to be extended at a future time (provided the data space also allows the extension).

All referenced external data files must exist before performing raw data I/O on the dataset. This is normally not a problem since those files are being managed directly by the application, or indirectly through some other library.

2.5. Datatype

Raw data has a constant datatype which describes the datatype of the raw data stored in the file, and a memory datatype that describes the datatype stored in application memory. Both data types are manipulated with the `H5T` API.

The constant file datatype is associated with the dataset when the dataset is created in a manner described below. Once assigned, the constant datatype can never be changed.

The memory datatype is specified when data is transferred to/from application memory. In the name of data sharability, the memory datatype must be specified, but can be the same type identifier as the constant datatype.

During dataset I/O operations, the library translates the raw data from the constant datatype to the memory datatype or vice versa. Structured datatypes include member offsets to allow reordering of struct members and/or selection of a subset of members and array datatypes include index permutation information to allow things like transpose operations (**the prototype does not support array reordering**) Permutations are relative to some extrinsic descritpion of the dataset.

2.6. Data Space

The dataspace of a dataset defines the number of dimensions and the size of each dimension and is manipulated with the `H5S` API. The *simple* dataspace consists of maximum dimension sizes and actual dimension sizes, which are usually the same. However, maximum dimension sizes can be the constant `H5D_UNLIMITED` in which case the actual dimension size can be incremented with calls to `H5Dextend()`. The maximium dimension sizes are constant meta data while the actual dimension sizes are persistent meta data. Initial actual dimension sizes are supplied at the same time as the maximum dimension sizes when the dataset is created.

The dataspace can also be used to define partial I/O operations. Since I/O operations have two end-points, the raw data transfer functions take two data space arguments: one which describes the application memory data space or subset thereof and another which describes the file data space or subset thereof.

2.7. Setting Constant or Persistent Properties

Each dataset has a set of constant and persistent properties which describe the layout method, pre-compression transformation, compression method, datatype, external storage, and data space. The constant properties are set as described above in a dataset creation property list whose identifier is passed to `H5Dcreate()`.

`hid_t H5Dcreate (hid_t` *`file_id`*`, const char *`*`name`*`, hid_t` *`type_id`*`, hid_t` *`space_id`*`, hid_t` *`create_plist_id`*`)`

A dataset is created by calling `H5Dcreate` with a file identifier, a dataset name, a datatype, a data space, and constant properties. The datatype and data space are the type and space of the dataset as it will exist in the file, which may be different than in application memory. The *create_plist_id* is a `H5P_DATASET_CREATE` property list created with `H5Pcreate()` and initialized with the various functions described above. `H5Dcreate()` returns a dataset handle for success or negative for failure. The handle should eventually be closed by calling `H5Dclose()` to release resources it uses.

`hid_t H5Dopen (hid_t` *`file_id`*`, const char *`*`name`*`)`

An existing dataset can be opened for access by calling this function. A dataset handle is returned for success or a negative value is returned for failure. The handle should eventually be closed by calling `H5Dclose()` to release resources it uses.

`herr_t H5Dclose (hid_t` *`dataset_id`*`)`

This function closes a dataset handle and releases all resources it might have been using. The handle should not be used in subsequent calls to the library.

`herr_t H5Dextend (hid_t` *`dataset_id`*`, hsize_t` *`dim`*`[])`

This function extends a dataset by increasing the size in one or more dimensions. Not all datasets can be extended.

2.8. Querying Constant or Persistent Properties

Constant or persistent properties can be queried with a set of three functions. Each function returns an identifier for a copy of the requested properties. The identifier can be passed to various functions which modify the underlying object to derive a new object; the original dataset is completely unchanged. The return values from these functions should be properly destroyed when no longer needed.

`hid_t H5Dget_type (hid_t` *`dataset_id`*`)`

Returns an identifier for a copy of the dataset permanent datatype or negative for failure.

`hid_t H5Dget_space (hid_t` *`dataset_id`*`)`

Returns an identifier for a copy of the dataset permanent data space, which also contains information about the current size of the dataset if the data set is extendable with `H5Dextend()`.

`hid_t H5Dget_create_plist (hid_t` *`dataset_id`*`)`

Returns an identifier for a copy of the dataset creation property list. The new property list is created by examining various permanent properties of the dataset. This is mostly a catch-all for everything but type and space.

2.9. Setting Memory and Transfer Properties

A dataset also has memory properties which describe memory within the application, and transfer properties that control various aspects of the I/O operations. The memory can have a datatype different than the permanent file datatype (different number types, different struct member offsets, different array element orderings) and can also be a different size (memory is a subset of the permanent dataset elements, or vice versa). The transfer properties might provide caching hints or collective I/O information. Therefore, each I/O operation must specify memory and transfer properties.

The memory properties are specified with *type_id* and *space_id* arguments while the transfer properties are specified with the *transfer_id* property list for the `H5Dread()` and `H5Dwrite()` functions (these functions are described below).

`herr_t H5Pset_buffer (hid_t` *`xfer_plist`*`, size_t` *`max_buf_size`*`, void *`*`tconv_buf`*`, void *`*`bkg_buf`*`)`

`size_t H5Pget_buffer (hid_t` *`xfer_plist`*`, void **`*`tconv_buf`*`, void **`*`bkg_buf`*`)`

Sets or retrieves the maximum size in bytes of the temporary buffer used for datatype conversion in the I/O pipeline. An application-defined buffer can also be supplied as the *tconv_buf* argument, otherwise a buffer will be allocated and freed on demand by the library. A second temporary buffer *bkg_buf* can also be supplied and should be the same size as the *tconv_buf*. The default values are 1MB for the maximum buffer size, and null pointers for each buffer indicating that they should be allocated on demand and freed when no longer needed. The `H5Pget_buffer()` function returns the maximum buffer size or zero on error.

If the maximum size of the temporary I/O pipeline buffers is too small to hold the entire I/O request, then the I/O request will be fragmented and the transfer operation will be strip mined. However, certain restrictions apply to the strip mining.

For instance, when performing I/O on a hyperslab of a simple data space the strip mining is in terms of the slowest varying dimension. So if a 100x200x300 hyperslab is requested, the temporary buffer must be large enough to hold a 1x200x300 sub-hyperslab.

To prevent strip mining from happening, the application should use `H5Pset_buffer()` to set the size of the temporary buffer so it's large enough to hold the entire request.

Example

This example shows how to define a function that sets a dataset transfer property list so that strip mining does not occur. It takes an (optional) dataset transfer property list, a dataset, a data space that describes what data points are being transfered, and a datatype for the data points in memory. It returns a (new) dataset transfer property list with the temporary buffer size set to an appropriate value. The return value should be passed as the fifth argument to `H5Dread()` or `H5Dwrite()`.

```
hid_t
disable_strip_mining (hid_t xfer_plist, hid_t dataset,
                      hid_t space, hid_t mem_type)
{
    hid_t file_type;          /* File datatype */
    size_t type_size;         /* Sizeof larger type */
    size_t size;              /* Temp buffer size */
    hid_t xfer_plist;         /* Return value */

    file_type = H5Dget_type (dataset);
    type_size = MAX(H5Tget_size(file_type), H5Tget_size(mem_type));
    H5Tclose (file_type);
    size = H5Sget_npoints(space) * type_size;
    if (xfer_plist<0) xfer_plist = H5Pcreate (H5P_DATASET_XFER);
    H5Pset_buffer(xfer_plist, size, NULL, NULL);
    return xfer_plist;
}
```

2.10. Querying Memory or Transfer Properties

Unlike constant and persistent properties, a dataset cannot be queried for it's memory or transfer properties. Memory properties cannot be queried because the application already stores those properties separate from the buffer that holds the raw data, and the buffer may hold multiple segments from various datasets and thus have more than one set of memory properties. The transfer properties cannot be queried from the dataset because they're associated with the transfer itself and not with the dataset (but one can call `H5Pget_`*`property`*`()` to query transfer properties from a tempalate).

2.11. Raw Data I/O

All raw data I/O is accomplished through these functions which take a dataset handle, a memory datatype, a memory data space, a file data space, transfer properties, and an application memory buffer. They translate data between the memory datatype and space and the file datatype and space. The data spaces can be used to describe partial I/O operations.

`herr_t H5Dread (hid_t` *`dataset_id`*`, hid_t` *`mem_type_id`*`, hid_t` *`mem_space_id`*`, hid_t` *`file_space_id`*`, hid_t` *`xfer_plist_id`*`, void *`*`buf`*`/*out*/)`

Reads raw data from the specified dataset into *buf* converting from file datatype and space to memory datatype and space.

herr_t H5Dwrite (hid_t *dataset_id*, hid_t *mem_type_id*, hid_t *mem_space_id*, hid_t *file_space_id*, hid_t *xfer_plist_id*, const void **buf*)

> Writes raw data from an application buffer *buf* to the specified dataset converting from memory datatype and space to file datatype and space.

In the name of sharability, the memory datatype must be supplied. However, it can be the same identifier as was used to create the dataset or as was returned by `H5Dget_type()`; the library will not implicitly derive memory datatypes from constant datatypes.

For complete reads of the dataset one may supply `H5S_ALL` as the argument for the file data space. If `H5S_ALL` is also supplied as the memory data space then no data space conversion is performed. This is a somewhat dangerous situation since the file data space might be different than what the application expects.

2.12. Examples

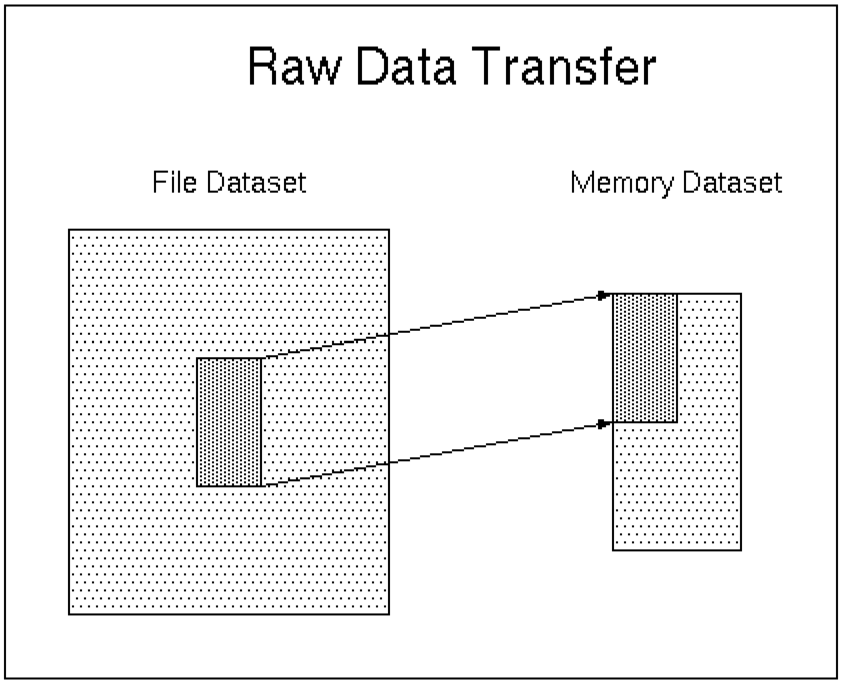
The examples in this section illustrate some common dataset practices.

This example shows how to create a dataset which is stored in memory as a two-dimensional array of native `double` values but is stored in the file in Cray `float` format using LZ77 compression. The dataset is written to the HDF5 file and then read back as a two-dimensional array of `float` values.

Example 1

```
hid_t file, data_space, dataset, properties;
double dd[500][600];
float ff[500][600];
hsize_t dims[2], chunk_size[2];

/* Describe the size of the array */
dims[0] = 500;
dims[1] = 600;
data_space = H5Screate_simple (2, dims);


/*
 * Create a new file using with read/write access,
 * default file creation properties, and default file
 * access properties.
 */
file = H5Fcreate ("test.h5", H5F_ACC_RDWR, H5P_DEFAULT,
                  H5P_DEFAULT);

/*
 * Set the dataset creation plist to specify that
 * the raw data is to be partitioned into 100x100 element
 * chunks and that each chunk is to be compressed with
 * LZ77.
 */
chunk_size[0] = chunk_size[1] = 100;
properties = H5Pcreate (H5P_DATASET_CREATE);
H5Pset_chunk (properties, 2, chunk_size);
H5Pset_compression (properties, H5D_COMPRESS_LZ77);

/*
 * Create a new dataset within the file.  The datatype
 * and data space describe the data on disk, which may
 * be different than the format used in the application’s
 * memory.
 */
dataset = H5Dcreate (file, "dataset", H5T_CRAY_FLOAT,
                     data_space, properties);

/*
 * Write the array to the file.  The datatype and data
 * space describe the format of the data in the ‘dd’
 * buffer.  The raw data is translated to the format
 * required on disk defined above.  We use default raw
 * data transfer properties.
 */
H5Dwrite (dataset, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
          H5P_DEFAULT, dd);

/*
 * Read the array as floats.  This is similar to writing
 * data except the data flows in the opposite direction.
 */
H5Dread (dataset, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL,
         H5P_DEFAULT, ff);

H5Dclose (dataset);
H5Sclose (data_space);
H5Pclose (properties);
H5Fclose (file);
```

This example uses the file created in Example 1 and reads a hyperslab of the 500x600 file dataset. The hyperslab size is 100x200 and it is located beginning at element <200,200>. We read the hyperslab into an 200x400 array in memory beginning at element <0,0> in memory. Visually, the transfer looks something like this:



Example 2

```
hid_t file, mem_space, file_space, dataset;
double dd[200][400];
hssize_t offset[2];
hsize size[2];

/*
 * Open an existing file and its dataset.
 */
file = H5Fopen ("test.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
dataset = H5Dopen (file, "dataset");

/*
 * Describe the file data space.
 */
offset[0] = 200; /*offset of hyperslab in file*/
offset[1] = 200;
size[0] = 100;   /*size of hyperslab*/
size[1] = 200;
file_space = H5Dget_space (dataset);
H5Sset_hyperslab (file_space, 2, offset, size);

/*
 * Describe the memory data space.
 */
size[0] = 200;  /*size of memory array*/
size[1] = 400;
mem_space = H5Screate_simple (2, size);

offset[0] = 0;  /*offset of hyperslab in memory*/
offset[1] = 0;
size[0] = 100;  /*size of hyperslab*/
size[1] = 200;
H5Sset_hyperslab (mem_space, 2, offset, size);

/*
 * Read the dataset.
 */
H5Dread (dataset, H5T_NATIVE_DOUBLE, mem_space,
         file_space, H5P_DEFAULT, dd);

/*
 * Close/release resources.
 */
H5Dclose (dataset);
H5Sclose (mem_space);
H5Sclose (file_space);
H5Fclose (file);
```

If the file contains a compound data structure one of whose members is a floating point value (call it "delta") but the application is interested in reading an array of floating point values which are just the "delta" values, then the application should cast the floating point array as a struct with a single "delta" member.

Example 3

```
hid_t file, dataset, type;
double delta[200];

/*
 * Open an existing file and its dataset.
 */
file = H5Fopen ("test.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
dataset = H5Dopen (file, "dataset");

/*
 * Describe the memory datatype, a struct with a single
 * "delta" member.
 */
type = H5Tcreate (H5T_COMPOUND, sizeof(double));
H5Tinsert (type, "delta", 0, H5T_NATIVE_DOUBLE);

/*
 * Read the dataset.
 */
H5Dread (dataset, type, H5S_ALL, H5S_ALL,
         H5P_DEFAULT, dd);

/*
 * Close/release resources.
 */
H5Dclose (dataset);
H5Tclose (type);
H5Fclose (file);
```

Last modified: 14 October 1999

3. The Datatype Interface (H5T)

3.1. Introduction

The datatype interface provides a mechanism to describe the storage format of individual data points of a data set and is hopefully designed in such a way as to allow new features to be easily added without disrupting applications that use the datatype interface. A dataset (the H5D interface) is composed of a collection or raw data points of homogeneous type organized according to the data space (the H5S interface).

A datatype is a collection of datatype properties, all of which can be stored on disk, and which when taken as a whole, provide complete information for data conversion to or from that datatype. The interface provides functions to set and query properties of a datatype.

A *data point* is an instance of a *datatype*, which is an instance of a *type class*. We have defined a set of type classes and properties which can be extended at a later time. The atomic type classes are those which describe types which cannot be decomposed at the datatype interface level; all other classes are compound.

3.2. General Datatype Operations

The functions defined in this section operate on datatypes as a whole. New datatypes can be created from scratch or copied from existing datatypes. When a datatype is no longer needed its resources should be released by calling `H5Tclose()`.

Datatypes come in two flavors: named datatypes and transient datatypes. A named datatype is stored in a file while the transient flavor is independent of any file. Named datatypes are always read-only, but transient types come in three varieties: modifiable, read-only, and immutable. The difference between read-only and immutable types is that immutable types cannot be closed except when the entire library is closed (the predefined types like `H5T_NATIVE_INT` are immutable transient types).

`hid_t H5Tcreate (H5T_class_t` *`class`*`, size_t` *`size`*`)`

> Datatypes can be created by calling this function, where *class* is a datatype class identifier. However, the only class currently allowed is `H5T_COMPOUND` to create a new empty compound datatype where *size* is the total size in bytes of an instance of this datatype. Other datatypes are created with `H5Tcopy()`. All functions that return datatype identifiers return a negative value for failure.

`hid_t H5Topen (hid_t` *`location`*`, const char *`*`name`*`)`

> A named datatype can be opened by calling this function, which returns a datatype identifier. The identifier should eventually be released by calling `H5Tclose()` to release resources. The named datatype returned by this function is read-only or a negative value is returned for failure. The *location* is either a file or group identifier.

`herr_t H5Tcommit (hid_t` *`location`*`, const char *`*`name`*`, hid_t` *`type`*`)`

> A transient datatype (not immutable) can be committed to a file and turned into a named datatype by calling this function. The *location* is either a file or group identifier and when combined with *name* refers to a new named datatype.

`htri_t H5Tcommitted (hid_t` *`type`*`)`

A type can be queried to determine if it is a named type or a transient type. If this function returns a positive value then the type is named (that is, it has been committed perhaps by some other application). Datasets which return committed datatypes with `H5Dget_type()` are able to share the datatype with other datasets in the same file.

`hid_t H5Tcopy (hid_t` *`type`*`)`

This function returns a modifiable transient datatype which is a copy of *type* or a negative value for failure. If *type* is a dataset identifier then the type returned is a modifiable transient copy of the datatype of the specified dataset.

`herr_t H5Tclose (hid_t` *`type`*`)`

Releases resources associated with a datatype. The datatype identifier should not be subsequently used since the results would be unpredictable. It is illegal to close an immutable transient datatype.

`htri_t H5Tequal (hid_t` *`type1`*`, hid_t` *`type2`*`)`

Determines if two types are equal. If *type1* and *type2* are the same then this function returns `TRUE`, otherwise it returns `FALSE` (an error results in a negative return value).

`herr_t H5Tlock (hid_t` *`type`*`)`

A transient datatype can be locked, making it immutable (read-only and not closable). The library does this to all predefined types to prevent the application from inadvertently modifying or deleting (closing) them, but the application is also allowed to do this for its own datatypes. Immutable datatypes are closed when the library closes (either by `H5close()` or by normal program termination).

3.3. Properties of Atomic Types

An atomic type is a type which cannot be decomposed into smaller units at the API level. All atomic types have a common set of properties which are augmented by properties specific to a particular type class. Some of these properties also apply to compound datatypes, but we discuss them only as they apply to atomic datatypes here. The properties and the functions that query and set their values are:

`H5T_class_t H5Tget_class (hid_t` *`type`*`)`

This property holds one of the class names: `H5T_INTEGER, H5T_FLOAT, H5T_TIME, H5T_STRING, or H5T_BITFIELD`. This property is read-only and is set when the datatype is created or copied (see `H5Tcreate()`, `H5Tcopy()`). If this function fails it returns `H5T_NO_CLASS` which has a negative value (all other class constants are non-negative).

`size_t H5Tget_size (hid_t` *`type`*`)`

`herr_t H5Tset_size (hid_t` *`type`*`, size_t` *`size`*`)`

This property is total size of the datum in bytes, including padding which may appear on either side of the actual value. If this property is reset to a smaller value which would cause the significant part of the data to extend beyond the edge of the datatype then the `offset` property is decremented a bit at a time. If the offset reaches zero and the significant part of the data still extends beyond the edge of the datatype then the `precision` property is decremented a bit at a time. Decreasing the size of a datatype may fail if the `H5T_FLOAT` bit fields would extend beyond the significant part of the type. Adjusting the size of an `H5T_STRING` automatically adjusts the precision as well. On error, `H5Tget_size()` returns zero which is never a valid size.

`H5T_order_t H5Tget_order (hid_t` *`type`*`)`

`herr_t H5Tset_order (hid_t` *`type`*`, H5T_order_t` *`order`*`)`

All atomic datatypes have a byte order which describes how the bytes of the datatype are layed out in memory. If the lowest memory address contains the least significant byte of the datum then it is said to be *little-endian* or `H5T_ORDER_LE`. If the bytes are in the oposite order then they are said to be *big-endian* or `H5T_ORDER_BE`. Some datatypes have the same byte order on all machines and are `H5T_ORDER_NONE` (like character strings). If `H5Tget_order()` fails then it returns `H5T_ORDER_ERROR` which is a negative value (all successful return values are non-negative).

`size_t H5Tget_precision (hid_t` *`type`*`)`

`herr_t H5Tset_precision (hid_t` *`type`*`, size_t` *`precision`*`)`

Some datatypes occupy more bytes than what is needed to store the value. For instance, a `short` on a Cray is 32 significant bits in an eight-byte field. The `precision` property identifies the number of significant bits of a datatype and the `offset` property (defined below) identifies its location. The `size` property defined above represents the entire size (in bytes) of the datatype. If the precision is decreased then padding bits are inserted on the MSB side of the significant bits (this will fail for `H5T_FLOAT` types if it results in the sign, mantissa, or exponent bit field extending beyond the edge of the significant bit field). On the other hand, if the precision is increased so that it "hangs over" the edge of the total size then the `offset` property is decremented a bit at a time. If the `offset` reaches zero and the significant bits still hang over the edge, then the total size is increased a byte at a time. The precision of an `H5T_STRING` is read-only and is always eight times the value returned by `H5Tget_size()`. `H5Tget_precision()` returns zero on failure since zero is never a valid precision.

`size_t H5Tget_offset (hid_t` *`type`*`)`

`herr_t H5Tset_offset (hid_t` *`type`*`, size_t` *`offset`*`)`

While the `precision` property defines the number of significant bits, the `offset` property defines the location of those bits within the entire datum. The bits of the entire data are numbered beginning at zero at the least significant bit of the least significant byte (the byte at the lowest memory address for a little-endian type or the byte at the highest address for a big-endian type). The `offset` property defines the bit location of the least signficant bit of a bit field whose length is `precision`. If the offset is increased so the significant bits "hang over" the edge of the datum, then the `size` property is automatically incremented. The offset is a read-only property of an `H5T_STRING` and is always zero. `H5Tget_offset()` returns zero on failure which is also a valid offset, but is guaranteed to succeed if a call to `H5Tget_precision()` succeeds with the same arguments.

`herr_t H5Tget_pad (hid_t` *`type`*`, H5T_pad_t *`*`lsb`*`, H5T_pad_t *`*`msb`*`)`

`herr_t H5Tset_pad (hid_t` *`type`*`, H5T_pad_t` *`lsb`*`, H5T_pad_t` *`msb`*`)`

The bits of a datum which are not significant as defined by the `precision` and `offset` properties are called *padding*. Padding falls into two categories: padding in the low-numbered bits is *lsb* padding and padding in the high-numbered bits is *msb* padding (bits are numbered according to the description for the `offset` property). Padding bits can always be set to zero (`H5T_PAD_ZERO`) or always set to one (`H5T_PAD_ONE`). The current pad types are returned through arguments of `H5Tget_pad()` either of which may be null pointers.

3.3.1. Properties of Integer Atomic Types

Integer atomic types (`class=H5T_INTEGER`) describe integer number formats. Such types include the following information which describes the type completely and allows conversion between various integer atomic types.

`H5T_sign_t H5Tget_sign (hid_t` *`type`*`)`

`herr_t H5Tset_sign (hid_t` *`type`*`, H5T_sign_t` *`sign`*`)`

Integer data can be signed two's complement (`H5T_SGN_2`) or unsigned (`H5T_SGN_NONE`). Whether data is signed or not becomes important when converting between two integer datatypes of differing sizes as it determines how values are truncated and sign extended.

3.3.2. Properties of Floating-point Atomic Types

The library supports floating-point atomic types (`class=H5T_FLOAT`) as long as the bits of the exponent are contiguous and stored as a biased positive number, the bits of the mantissa are contiguous and stored as a positive magnitude, and a sign bit exists which is set for negative values. Properties specific to floating-point types are:

`herr_t H5Tget_fields (hid_t` *`type`*`, size_t *`*`spos`*`, size_t *`*`epos`*`, size_t *`*`esize`*`, size_t *`*`mpos`*`, size_t *`*`msize`*`)`

`herr_t H5Tset_fields (hid_t` *`type`*`, size_t` *`spos`*`, size_t` *`epos`*`, size_t` *`esize`*`, size_t` *`mpos`*`, size_t` *`msize`*`)`

A floating-point datum has bit fields which are the exponent and mantissa as well as a mantissa sign bit. These properties define the location (bit position of least significant bit of the field) and size (in bits) of each field. The bit positions are numbered beginning at zero at the beginning of the significant part of the datum (see the descriptions of the `precision` and `offset` properties). The sign bit is always of length one and none of the fields are allowed to overlap. When expanding a floating-point type one should set the precision first; when decreasing the size one should set the field positions and sizes first.

`size_t H5Tget_ebias (hid_t` *`type`*`)`

`herr_t H5Tset_ebias (hid_t` *`type`*`, size_t` *`ebias`*`)`

The exponent is stored as a non-negative value which is `ebias` larger than the true exponent. `H5Tget_ebias()` returns zero on failure which is also a valid exponent bias, but the function is guaranteed to succeed if `H5Tget_precision()` succeeds when called with the same arguments.

`H5T_norm_t H5Tget_norm (hid_t` *`type`*`)`

`herr_t H5Tset_norm (hid_t` *`type`*`, H5T_norm_t` *`norm`*`)`

This property determines the normalization method of the mantissa.

- If the value is `H5T_NORM_MSBSET` then the mantissa is shifted left (if non-zero) until the first bit after the radix point is set and the exponent is adjusted accordingly. All bits of the mantissa after the radix point are stored.

- If its value is `H5T_NORM_IMPLIED` then the mantissa is shifted left (if non-zero) until the first bit after the radix point is set and the exponent is adjusted accordingly. The first bit after the radix point is not stored since it's always set.

- If its value is `H5T_NORM_NONE` then the fractional part of the mantissa is stored without normalizing it.

`H5T_pad_t H5Tget_inpad (hid_t *type*)`

`herr_t H5Tset_inpad (hid_t *type*, H5T_pad_t *inpad*)`

If any internal bits (that is, bits between the sign bit, the mantissa field, and the exponent field but within the precision field) are unused, then they will be filled according to the value of this property. The *inpad* argument can be `H5T_PAD_ZERO` if the internal padding should always be set to zero, or `H5T_PAD_ONE` if it should always be set to one. `H5Tget_inpad()` returns `H5T_PAD_ERROR` on failure which is a negative value (successful return is always non-negative).

3.3.3. Properties of Date and Time Atomic Types

Dates and times (`class=H5T_TIME`) are stored as character strings in one of the ISO-8601 formats like "*1997-12-05 16:25:30*"; as character strings using the Unix asctime(3) format like "*Thu Dec 05 16:25:30 1997*"; as an integer value by juxtaposition of the year, month, and day-of-month, hour, minute and second in decimal like *19971205162530*; as an integer value in Unix time(2) format; or other variations.

3.3.4. Properties of Character String Atomic Types

Fixed-length character string types are used to store textual information. The `offset` property of a string is always zero and the `precision` property is eight times as large as the value returned by `H5Tget_size()` (since precision is measured in bits while size is measured in bytes). Both properties are read-only.

`H5T_cset_t H5Tget_cset (hid_t *type*)`

`herr_t H5Tset_cset (hid_t *type*, H5T_cset_t *cset*)`

HDF5 is able to distinguish between character sets of different nationalities and to convert between them to the extent possible. The only character set currently supported is `H5T_CSET_ASCII`.

`H5T_str_t H5Tget_strpad (hid_t *type*)`

`herr_t H5Tset_strpad (hid_t *type*, H5T_str_t *strpad*)`

The method used to store character strings differs with the programming language: C usually null terminates strings while Fortran left-justifies and space-pads strings. This property defines the storage mechanism and can be

`H5T_STR_NULLTERM`

A C-style string which is guaranteed to be null terminated. When converting from a longer string the value will be truncated and then a null character appended.

`H5T_STR_NULLPAD`

A C-style string which is padded with null characters but not necessarily null terminated. Conversion from a long string to a shorter `H5T_STR_NULLPAD` string will truncate but not null terminate. Conversion from a short value to a longer value will append null characters as with `H5T_STR_NULLTERM`.

`H5T_STR_SPACEPAD`

A Fortran-style string which is padded with space characters. This is the same as `H5T_STR_NULLPAD` except the padding character is a space instead of a null.

H5Tget_strpad() returns H5T_STR_ERROR on failure, a negative value (all successful return values are non-negative).

3.3.5. Properties of Bit Field Atomic Types

Converting a bit field (class=H5T_BITFIELD) from one type to another simply copies the significant bits. If the destination is smaller than the source then bits are truncated. Otherwise new bits are filled according to the msb padding type.

3.3.6. Character and String Datatype Issues

The H5T_NATIVE_CHAR and H5T_NATIVE_UCHAR datatypes are actually numeric data (1-byte integers). If the application wishes to store character data, then an HDF5 *string* datatype should be derived from H5T_C_S1 instead.

Motivation

HDF5 defines at least three classes of datatypes: integer data, floating point data, and character data. However, the C language defines only integer and floating point datatypes; character data in C is overloaded on the 8- or 16-bit integer types and character strings are overloaded on arrays of those integer types which, by convention, are terminated with a zero element. In C, the variable unsigned char s[256] is either an array of numeric data, a single character string with at most 255 characters, or an array of 256 characters, depending entirely on usage. For uniformity with the other H5T_NATIVE_ types, HDF5 uses the numeric interpretation of H5T_NATIVE_CHAR and H5T_NATIVE_UCHAR.

Usage

To store unsigned char s[256] data as an array of integer values, use the HDF5 datatype H5T_NATIVE_UCHAR and a data space that describes the 256-element array. Some other application that reads the data will then be able to read, say, a 256-element array of 2-byte integers and HDF5 will perform the numeric translation. To store unsigned char s[256] data as a character string, derive a fixed length string datatype from H5T_C_S1 by increasing its size to 256 characters. Some other application that reads the data will be able to read, say, a space padded string of 16-bit characters and HDF5 will perform the character and padding translations.

```
hid_t s256 = H5Tcopy(H5T_C_S1);
             H5Tset_size(s256, 256);
```

To store unsigned char s[256] data as an array of 256 ASCII characters, use an HDF5 data space to describe the array and derive a one-character string type from H5T_C_S1. Some other application will be able to read a subset of the array as 16-bit characters and HDF5 will perform the character translations. The H5T_STR_NULLPAD is necessary because if H5T_STR_NULLTERM were used (the default) then the single character of storage would be for the null terminator and no useful data would actually be stored (unless the length were incremented to more than one character).

```
hid_t s1 = H5Tcopy(H5T_C_S1);
           H5Tset_strpad(s1, H5T_STR_NULLPAD);
```

Summary

The C language uses the term char to represent one-byte numeric data and does not make character strings a first-class datatype. HDF5 makes a distinction between integer and character data and maps the C signed char (H5T_NATIVE_CHAR) and unsigned char (H5T_NATIVE_UCHAR) datatypes to the HDF5 integer type class.

3.4. Properties of Opaque Types

Opaque types (`class=H5T_OPAQUE`) provide the application with a mechanism for describing data which cannot be otherwise described by HDF5. The only properties associated with opaque types are a size in bytes and an ASCII tag which is manipulated with `H5Tset_tag()` and `H5Tget_tag()` functions. The library contains no predefined conversion functions but the application is free to register conversions between any two opaque types or between an opaque type and some other type.

3.5. Properties of Compound Types

A compound datatype is similar to a `struct` in C or a common block in Fortran: it is a collection of one or more atomic types or small arrays of such types. Each *member* of a compound type has a name which is unique within that type, and a byte offset that determines the first byte (smallest byte address) of that member in a compound datum. A compound datatype has the following properties:

`H5T_class_t H5Tget_class (hid_t` *`type`*`)`

All compound datatypes belong to the type class `H5T_COMPOUND`. This property is read-only and is defined when a datatype is created or copied (see `H5Tcreate()` or `H5Tcopy()`).

`size_t H5Tget_size (hid_t` *`type`*`)`

Compound datatypes have a total size in bytes which is returned by this function. All members of a compound datatype must exist within this size. A value of zero is returned for failure; all successful return values are positive.

`int H5Tget_nmembers (hid_t` *`type`*`)`

A compound datatype consists of zero or more members (defined in any order) with unique names and which occupy non-overlapping regions within the datum. In the functions that follow, individual members are referenced by an index number between zero and *N*-1, inclusive, where *N* is the value returned by this function. `H5Tget_nmembers()` returns -1 on failure.

`char *H5Tget_member_name (hid_t` *`type`*`, int` *`membno`*`)`

Each member has a name which is unique among its siblings in a compound datatype. This function returns a pointer to a null-terminated copy of the name allocated with `malloc()` or the null pointer on failure. The caller is responsible for freeing the memory returned by this function.

`size_t H5Tget_member_offset (hid_t` *`type`*`, int` *`membno`*`)`

The byte offset of member number *membno* with respect to the beginning of the containing compound datum is returned by this function. A zero is returned on failure which is also a valid offset, but this function is guaranteed to succeed if a call to `H5Tget_member_dims()` succeeds when called with the same *type* and *membno* arguments.

`int H5Tget_member_dims (hid_t` *`type`*`, int` *`membno`*`, int` *`dims`*`[4], int` *`perm`*`[4])`

Each member can be a small array of up to four dimensions, making it convenient to describe things like transposition matrices. The dimensionality of the member is returned (or negative for failure) and the size in each dimension is returned through the *dims* argument. The *perm* argument describes how the array’s elements are mapped to the linear address space of memory with respect to some reference order (the reference order is specified in natural language documentation which describes the compound datatype). The application which "invented" the type will often use the identity permutation and other applications will use a permutation that causes the elements to be rearranged to the desired order. Only the first few elements of *dims* and *perm* are initialized according to the dimensionality of the member. Scalar members have dimensionality zero. **The only permutations supported at this time are the identity permutation and the transpose permutation (in the 4d case, {0,1,2,3} and {3,2,1,0}).**

`hid_t H5Tget_member_type (hid_t` *`type`*`, int` *`membno`*`)`

Each member has its own datatype, a copy of which is returned by this function. The returned datatype identifier should be released by eventually calling `H5Tclose()` on that type.

Properties of members of a compound datatype are defined when the member is added to the compound type (see `H5Tinsert()`) and cannot be subsequently modified. This makes it imposible to define recursive data structures.

3.6. Predefined Atomic Datatypes

The library predefines a modest number of datatypes having names like `H5T_`*`arch_base`* where *arch* is an architecture name and *base* is a programming type name. New types can be derived from the predifined types by copying the predefined type (see `H5Tcopy()`) and then modifying the result.

Architecture Name	Description
`IEEE`	This architecture defines standard floating point types in various byte orders.
`STD`	This is an architecture that contains semi-standard datatypes like signed two’s complement integers, unsigned integers, and bitfields in various byte orders.
`UNIX`	Types which are specific to Unix operating systems are defined in this architecture. The only type currently defined is the Unix date and time types (`time_t`).
`C` `FORTRAN`	Types which are specific to the C or Fortran programming languages are defined in these architectures. For instance, `H5T_C_STRING` defines a base string type with null termination which can be used to derive string types of other lengths.
`NATIVE`	This architecture contains C-like datatypes for the machine on which the library was compiled. The types were actually defined by running the `H5detect` program when the library was compiled. In order to be portable, applications should almost always use this architecture to describe things in memory.
`CRAY`	Cray architectures. These are word-addressable, big-endian systems with non-IEEE floating point.
`INTEL`	All Intel and compatible CPU’s including 80286, 80386, 80486, Pentium, Pentium-Pro, and Pentium-II. These are little-endian systems with IEEE floating-point.
`MIPS`	All MIPS CPU’s commonly used in SGI systems. These are big-endian systems with IEEE floating-point.
`ALPHA`	All DEC Alpha CPU’s, little-endian systems with IEEE floating-point.

The base name of most types consists of a letter, a precision in bits, and an indication of the byte order. The letters are:

B	Bitfield
D	Date and time
F	Floating point
I	Signed integer
R	References
S	Character string
U	Unsigned integer

The byte order is a two-letter sequence:

BE	Big endian
LE	Little endian
VX	Vax order

Example	Description
`H5T_IEEE_F64LE`	Eight-byte, little-endian, IEEE floating-point
`H5T_IEEE_F32BE`	Four-byte, big-endian, IEEE floating point
`H5T_STD_I32LE`	Four-byte, little-endian, signed two's complement integer
`H5T_STD_U16BE`	Two-byte, big-endian, unsigned integer
`H5T_UNIX_D32LE`	Four-byte, little-endian, time_t
`H5T_C_S1`	One-byte, null-terminated string of eight-bit characters
`H5T_INTEL_B64`	Eight-byte bit field on an Intel CPU
`H5T_CRAY_F64`	Eight-byte Cray floating point
`H5T_STD_ROBJ`	Reference to an entire object in a file

The `NATIVE` architecture has base names which don’t follow the same rules as the others. Instead, native type names are similar to the C type names. Here are some examples:

Example	Corresponding C Type
`H5T_NATIVE_CHAR`	`char`
`H5T_NATIVE_SCHAR`	`signed char`
`H5T_NATIVE_UCHAR`	`unsigned char`
`H5T_NATIVE_SHORT`	`short`
`H5T_NATIVE_USHORT`	`unsigned short`
`H5T_NATIVE_INT`	`int`
`H5T_NATIVE_UINT`	`unsigned`
`H5T_NATIVE_LONG`	`long`
`H5T_NATIVE_ULONG`	`unsigned long`
`H5T_NATIVE_LLONG`	`long long`
`H5T_NATIVE_ULLONG`	`unsigned long long`
`H5T_NATIVE_FLOAT`	`float`
`H5T_NATIVE_DOUBLE`	`double`
`H5T_NATIVE_LDOUBLE`	`long double`
`H5T_NATIVE_HSIZE`	`hsize_t`
`H5T_NATIVE_HSSIZE`	`hssize_t`
`H5T_NATIVE_HERR`	`herr_t`
`H5T_NATIVE_HBOOL`	`hbool_t`

Example: A 128-bit integer

To create a 128-bit, little-endian signed integer type one could use the following (increasing the precision of a type automatically increases the total size):

```
hid_t new_type = H5Tcopy (H5T_NATIVE_INT);
H5Tset_precision (new_type, 128);
H5Tset_order (new_type, H5T_ORDER_LE);
```

Example: An 80-character string

To create an 80-byte null terminated string type one might do this (the offset of a character string is always zero and the precision is adjusted automatically to match the size):

```
hid_t str80 = H5Tcopy (H5T_C_S1);
H5Tset_size (str80, 80);
```

3.7. Defining Compound Datatypes

Unlike atomic datatypes which are derived from other atomic datatypes, compound datatypes are created from scratch. First, one creates an empty compound datatype and specifies it’s total size. Then members are added to the compound datatype in any order.

Usually a C struct will be defined to hold a data point in memory, and the offsets of the members in memory will be the offsets of the struct members from the beginning of an instance of the struct.

`HOFFSET(s,m)`

This macro computes the offset of member *m* within a struct *s*.

`offsetof(s,m)`

This macro defined in `stddef.h` does exactly the same thing as the `HOFFSET()` macro.

Each member must have a descriptive name which is the key used to uniquely identify the member within the compound datatype. A member name in an HDF5 datatype does not necessarily have to be the same as the name of the member in the C struct, although this is often the case. Nor does one need to define all members of the C struct in the HDF5 compound datatype (or vice versa).

Example: A simple struct

An HDF5 datatype is created to describe complex numbers whose type is defined by the `complex_t` struct.

```
typedef struct {
   double re;   /*real part*/
   double im;   /*imaginary part*/
} complex_t;

hid_t complex_id = H5Tcreate (H5T_COMPOUND, sizeof tmp);
H5Tinsert (complex_id, "real", HOFFSET(complex_t,re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (complex_id, "imaginary", HOFFSET(complex_t,im),
           H5T_NATIVE_DOUBLE);
```

Member alignment is handled by the `HOFFSET` macro. However, data stored on disk does not require alignment, so unaligned versions of compound data structures can be created to improve space efficiency on disk. These unaligned compound datatypes can be created by computing offsets by hand to eliminate inter-member padding, or the members can be packed by calling `H5Tpack()` (which modifies a datatype directly, so it is usually preceded by a call to `H5Tcopy()`):

Example: A packed struct

This example shows how to create a disk version of a compound datatype in order to store data on disk in as compact a form as possible. Packed compound datatypes should generally not be used to describe memory as they may violate alignment constraints for the architecture being used. Note also that using a packed datatype for disk storage may involve a higher data conversion cost.

```
hid_t complex_disk_id = H5Tcopy (complex_id);
H5Tpack (complex_disk_id);
```

Example: A flattened struct

Compound datatypes that have a compound datatype member can be handled two ways. This example shows that the compound datatype can be flattened, resulting in a compound type with only atomic members.

```
typedef struct {
   complex_t x;
   complex_t y;
} surf_t;

hid_t surf_id = H5Tcreate (H5T_COMPOUND, sizeof tmp);
H5Tinsert (surf_id, "x-re", HOFFSET(surf_t,x.re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (surf_id, "x-im", HOFFSET(surf_t,x.im),
           H5T_NATIVE_DOUBLE);
H5Tinsert (surf_id, "y-re", HOFFSET(surf_t,y.re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (surf_id, "y-im", HOFFSET(surf_t,y.im),
           H5T_NATIVE_DOUBLE);
```

Example: A nested struct

However, when the `complex_t` is used often it becomes inconvenient to list its members over and over again. So the alternative approach to flattening is to define a compound datatype and then use it as the type of the compound members, as is done here (the typedefs are defined in the previous examples).

```
hid_t complex_id, surf_id; /*hdf5 datatypes*/

complex_id = H5Tcreate (H5T_COMPOUND, sizeof c);
H5Tinsert (complex_id, "re", HOFFSET(complex_t,re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (complex_id, "im", HOFFSET(complex_t,im),
           H5T_NATIVE_DOUBLE);

surf_id = H5Tcreate (H5T_COMPOUND, sizeof s);
H5Tinsert (surf_id, "x", HOFFSET(surf_t,x), complex_id);
H5Tinsert (surf_id, "y", HOFFSET(surf_t,y), complex_id);
```

3.8. Enumeration Datatypes

3.8.1. Introduction

An HDF enumeration datatype is a 1:1 mapping between a set of symbols and a set of integer values, and an order is imposed on the symbols by their integer values. The symbols are passed between the application and library as character strings and all the values for a particular enumeration type are of the same integer type, which is not necessarily a native type.

3.8.2. Creation

Creation of an enumeration datatype resembles creation of a compound datatype: first an empty enumeration type is created, then members are added to the type, then the type is optionally locked.

`hid_t H5Tcreate(H5T_class_t` *`type_class`*`, size_t` *`size`*`)`

This function creates a new empty enumeration datatype based on a native signed integer type. The first argument is the constant `H5T_ENUM` and the second argument is the size in bytes of the native integer on which the enumeration type is based. If the architecture does not support a native signed integer of the specified size then an error is returned.

```
/* Based on a native signed short */
hid_t hdf_en_colors = H5Tcreate(H5T_ENUM, sizeof(short));
```

`hid_t H5Tenum_create(hid_t` *`base`*`)`

This function creates a new empty enumeration datatype based on some integer datatype *base* and is a generalization of the `H5Tcreate()` function. This function is useful when creating an enumeration type based on some non-native integer datatype, but it can be used for native types as well.

```
/* Based on a native unsigned short */
hid_t hdf_en_colors_1 = H5Tenum_create(H5T_NATIVE_USHORT);

/* Based on a MIPS 16-bit unsigned integer */
hid_t hdf_en_colors_2 = H5Tenum_create(H5T_MIPS_UINT16);

/* Based on a big-endian 16-bit unsigned integer */
hid_t hdf_en_colors_3 = H5Tenum_create(H5T_STD_U16BE);
```

`herr_t H5Tenum_insert(hid_t` *`etype`*`, const char *`*`symbol`*`, void *`*`value`*`)`

Members are inserted into the enumeration datatype *etype* with this function. Each member has a symbolic name *symbol* and some integer representation *value*. The *value* argument must point to a value of the same datatype as specified when the enumeration type was created. The order of member insertion is not important but all symbol names and values must be unique within a particular enumeration type.

```
short val;
H5Tenum_insert(hdf_en_colors, "RED",   (val=0,&val));
H5Tenum_insert(hdf_en_colors, "GREEN", (val=1,&val));
H5Tenum_insert(hdf_en_colors, "BLUE",  (val=2,&val));
H5Tenum_insert(hdf_en_colors, "WHITE", (val=3,&val));
H5Tenum_insert(hdf_en_colors, "BLACK", (val=4,&val));
```

`herr_t H5Tlock(hid_t` *`etype`*`)`

This function locks a datatype so it cannot be modified or freed unless the entire HDF5 library is closed. Its use is completely optional but using it on an application datatype makes that datatype act like a predefined datatype.

```
H5Tlock(hdf_en_colors);
```

3.8.3. Integer Operations

Because an enumeration datatype is derived from an integer datatype, any operation which can be performed on integer datatypes can also be performed on enumeration datatypes. This includes:

`H5Topen()`	`H5Tcreate()`	`H5Tcopy()`	`H5Tclose()`
`H5Tequal()`	`H5Tlock()`	`H5Tcommit()`	`H5Tcommitted()`
`H5Tget_class()`	`H5Tget_size()`	`H5Tget_order()`	`H5Tget_pad()`
`H5Tget_precision()`	`H5Tget_offset()`	`H5Tget_sign()`	`H5Tset_size()`
`H5Tset_order()`	`H5Tset_precision()`	`H5Tset_offset()`	`H5Tset_pad()`
`H5Tset_sign()`			

In addition, the new function `H5Tget_super()` will be defined for all datatypes that are derived from existing types (currently just enumeration types).

`hid_t H5Tget_super(hid_t` *`type`*`)`

Return the datatype from which *type* is derived. When *type* is an enumeration datatype then the returned value will be an integer datatype but not necessarily a native type. One use of this function would be to create a new enumeration type based on the same underlying integer type and values but with possibly different symbols.

```
hid_t itype = H5Tget_super(hdf_en_colors);
hid_t hdf_fr_colors = H5Tenum_create(itype);
H5Tclose(itype);

short val;
H5Tenum_insert(hdf_fr_colors, "ouge",  (val=0,&val));
H5Tenum_insert(hdf_fr_colors, "vert",  (val=1,&val));
H5Tenum_insert(hdf_fr_colors, "bleu",  (val=2,&val));
H5Tenum_insert(hdf_fr_colors, "blanc", (val=3,&val));
H5Tenum_insert(hdf_fr_colors, "noir",  (val=4,&val));
H5Tlock(hdf_fr_colors);
```

3.8.4. Type Functions

A small set of functions is available for querying properties of an enumeration type. These functions are likely to be used by browsers to display datatype information.

`int H5Tget_nmembers(hid_t` *`etype`*`)`

When given an enumeration datatype *etype* this function returns the number of members defined for that type. This function is already implemented for compound datatypes.

`char *H5Tget_member_name(hid_t` *`etype`*`, int` *`membno`*`)`

Given an enumeration datatype *etype* this function returns the symbol name for the member indexed by *membno*. Members are numbered from zero to *N*-1 where *N* is the return value from `H5Tget_nmembers()`. The members are stored in no particular order. This function is already implemented for compound datatypes. If an error occurs then the null pointer is returned. The return value should be freed by calling `free()`.

`herr_t H5Tget_member_value(hid_t` *`etype`*`, int` *`membno`*`, void *`*`value`*`/*out*/)`

Given an enumeration datatype *etype* this function returns the value associated with the member indexed by *membno* (as described for `H5Tget_member_name()`). The value returned is in the domain of the underlying integer datatype which is often a native integer type. The application should ensure that the memory pointed to by *value* is large enough to contain the result (the size can be obtained by calling `H5Tget_size()` on either the enumeration type or the underlying integer type when the type is not known by the C compiler.

```
int i, n = H5Tget_nmembers(hdf_en_colors);
for (i=0; i<n; i++) {
    char *symbol = H5Tget_member_name(hdf_en_colors, i);
    short val;
    H5Tget_member_value(hdf_en_colors, i, &val);
    printf("#%d %20s = %d\n", i, symbol, val);
    free(symbol);
}
```

Output:

```
#0                 BLACK = 4
#1                  BLUE = 2
#2                 GREEN = 1
#3                   RED = 0
#4                 WHITE = 3
```

3.8.5. Data Functions

In addition to querying about the enumeration type properties, an application may want to make queries about enumerated data. These functions perform efficient mappings between symbol names and values.

`herr_t H5Tenum_valueof(hid_t` *`etype`*`, const char *`*`symbol`*`, void *`*`value`*`/*out*/)`

Given an enumeration datatype *etype* this function returns through *value* the bit pattern associated with the symbol name *symbol*. The *value* argument should point to memory which is large enough to hold the result, which is returned as the underlying integer datatype specified when the enumeration type was created, often a native integer type.

`herr_t H5Tenum_nameof(hid_t` *`etype`*`, void *`*`value`*`, char *`*`symbol`*`, size_t` *`size`*`)`

This function translates a bit pattern pointed to by *value* to a symbol name according to the mapping defined in the enumeration datatype *etype* and stores at most *size* characters of that name (counting the null terminator) to the *symbol* buffer. If the name is longer than the result buffer then the result is not null terminated and the function returns failure. If *value* points to a bit pattern which is not in the domain of the enumeration type then the first byte of the *symbol* buffer is set to zero and the function fails.

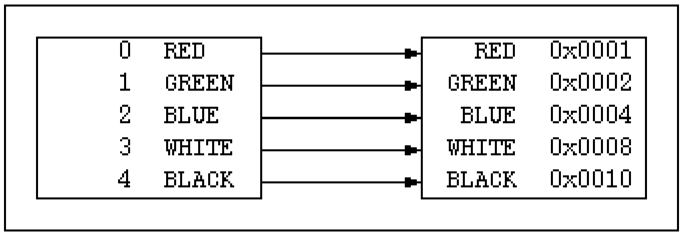
```
short data[1000] = {4, 2, 0, 0, 5, 1, ...};
int i;
char symbol[32];

for (i=0; i<1000; i++) {
    if (H5Tenum_nameof(hdf_en_colors, data+i, symbol,
                       sizeof symbol))<0) {
        if (symbol[0]) {
            strcpy(symbol+sizeof(symbol)-4, "...");
        } else {
            strcpy(symbol, "UNKNOWN");
        }
    }
    printf("%d %s\n", data[i], symbol);
}
printf("}\n");
```

Output:

```
4 BLACK
2 BLUE
0 RED
0 RED
5 UNKNOWN
1 GREEN
...
```

3.8.6. Conversion

Enumerated data can be converted from one type to another provided the destination enumeration type contains all the symbols of the source enumeration type. The conversion operates by matching up the symbol names of the source and destination enumeration types to build a mapping from source value to destination value. For instance, if we are translating from an enumeration type that defines a sequence of integers as the values for the colors to a type that defines a different bit for each color then the mapping might look like this:



That is, a source value of `2` which corresponds to `BLUE` would be mapped to `0x0004`. The following code snippet builds the second datatype, then converts a raw data array from one datatype to another, and then prints the result.

```
/* Create a new enumeration type */
short val;
hid_t bits = H5Tcreate(H5T_ENUM, sizeof val);
H5Tenum_insert(bits, "RED",   (val=0x0001,&val));
H5Tenum_insert(bits, "GREEN", (val=0x0002,&val));
H5Tenum_insert(bits, "BLUE",  (val=0x0004,&val));
H5Tenum_insert(bits, "WHITE", (val=0x0008,&val));
H5Tenum_insert(bits, "BLACK", (val=0x0010,&val));

/* The data */
short data[6] = {1, 4, 2, 0, 3, 5};

/* Convert the data from one type to another */
H5Tconvert(hdf_en_colors, bits, 5, data, NULL);

/* Print the data */
for (i=0; i<6; i++) {
    printf("0x%04x\n", (unsigned)(data[i]));
}
```

Output:

```
0x0002
0x0010
0x0004
0x0001
0x0008
0xffff
```

If the source data stream contains values which are not in the domain of the conversion map then an overflow exception is raised within the library, causing the application defined overflow handler to be invoked (see `H5Tset_overflow()`). If no overflow handler is defined then all bits of the destination value will be set.

The HDF library will not provide conversions between enumerated data and integers although the application is free to do so (this is a policy we apply to all classes of HDF datatypes). However, since enumeration types are derived from integer types it is permissible to treat enumerated data as integers and perform integer conversions in that context.

3.8.7. Symbol Order

Symbol order is determined by the integer values associated with each symbol. When the integer datatype is a native type, testing the relative order of two symbols is an easy process: simply compare the values of the symbols. If only the symbol names are available then the values must first be determined by calling `H5Tenum_valueof()`.

```
short val1, val2;
H5Tenum_valueof(hdf_en_colors, "WHITE", &val1);
H5Tenum_valueof(hdf_en_colors, "BLACK", &val2);
if (val1 < val2) ...
```

When the underlying integer datatype is not a native type then the easiest way to compare symbols is to first create a similar enumeration type that contains all the same symbols but has a native integer type (HDF type conversion features can be used to convert the non-native values to native values). Once we have a native type we can compare symbol order as just described. If `foreign` is some non-native enumeration type then a native type can be created as follows:

```
int n = H5Tget_nmembers(foreign);
hid_t itype = H5Tget_super(foreign);
void *val = malloc(n * MAX(H5Tget_size(itype), sizeof(int)));
char *name = malloc(n * sizeof(char*));
int i;

/* Get foreign type information */
for (i=0; i<n; i++) {
    name[i] = H5Tget_member_name(foreign, i);
    H5Tget_member_value(foreign, i,
                        (char*)val+i*H5Tget_size(foreign));
}

/* Convert integer values to new type */
H5Tconvert(itype, H5T_NATIVE_INT, n, val, NULL);

/* Build a native type */
hid_t native = H5Tenum_create(H5T_NATIVE_INT);
for (i=0; i<n; i++) {
    H5Tenum_insert(native, name[i], ((int*)val)[i]);
    free(name[i]);
}
free(name);
free(val);
```

It is also possible to convert enumerated data to a new type that has a different order defined for the symbols. For instance, we can define a new type, `reverse` that defines the same five colors but in the reverse order.

```
short val;
int i;
char sym[8];
short data[5] = {0, 1, 2, 3, 4};

hid_t reverse = H5Tenum_create(H5T_NATIVE_SHORT);
H5Tenum_insert(reverse, "BLACK", (val=0,&val));
H5Tenum_insert(reverse, "WHITE", (val=1,&val));
H5Tenum_insert(reverse, "BLUE",  (val=2,&val));
H5Tenum_insert(reverse, "GREEN", (val=3,&val));
H5Tenum_insert(reverse, "RED",   (val=4,&val));

/* Print data */
for (i=0; i<5; i++) {
    H5Tenum_nameof(hdf_en_colors, data+i, sym, sizeof sym);
    printf ("%d %s\n", data[i], sym);
}

puts("Converting...");
H5Tconvert(hdf_en_colors, reverse, 5, data, NULL);

/* Print data */
for (i=0; i<5; i++) {
    H5Tenum_nameof(reverse, data+i, sym, sizeof sym);
    printf ("%d %s\n", data[i], sym);
}
```

Output:

```
0 RED
1 GREEN
2 BLUE
3 WHITE
4 BLACK
Converting...
4 RED
3 GREEN
2 BLUE
1 WHITE
0 BLACK
```

3.8.8. Equality

The order that members are inserted into an enumeration type is unimportant; the important part is the associations between the symbol names and the values. Thus, two enumeration datatypes will be considered equal if and only if both types have the same symbol/value associations and both have equal underlying integer datatypes. Type equality is tested with the `H5Tequal()` function.

3.8.9. Interacting with C’s `enum` Type

Although HDF enumeration datatypes are similar to C `enum` datatypes, there are some important differences:

Difference	Motivation/Implications
Symbols are unquoted in C but quoted in HDF.	This allows the application to manipulate symbol names in ways that are not possible with C.
The C compiler automatically replaces all symbols with their integer values but HDF requires explicit calls to do the same.	C resolves symbols at compile time while HDF resolves symbols at run time.
The mapping from symbols to integers is *N*:1 in C but 1:1 in HDF.	HDF can translate from value to name uniquely and large `switch` statements are not necessary to print values in human-readable format.
A symbol must appear in only one C `enum` type but may appear in multiple HDF enumeration types.	The translation from symbol to value in HDF requires the datatype to be specified while in C the datatype is not necessary because it can be inferred from the symbol.
The underlying integer value is always a native integer in C but can be a foreign integer type in HDF.	This allows HDF to describe data that might reside on a foreign architecture, such as data stored in a file.
The sign and size of the underlying integer datatype is chosen automatically by the C compiler but must be fully specified with HDF.	Since HDF doesn’t require finalization of a datatype, complete specification of the type must be supplied before the type is used. Requiring that information at the time of type creation was a design decision to simplify the library.

The examples below use the following C datatypes:

```
/* English color names */
typedef enum {
    RED,
    GREEN,
    BLUE,
    WHITE,
    BLACK
} c_en_colors;

/* Spanish color names, reverse order */
typedef enum {
    NEGRO
    BLANCO,
    AZUL,
    VERDE,
    ROJO,
} c_sp_colors;

/* No enum definition for French names */
```

Creating HDF Types from C Types

An HDF enumeration datatype can be created from a C `enum` type simply by passing pointers to the C `enum` values to `H5Tenum_insert()`. For instance, to create HDF types for the `c_en_colors` type shown above:

```
c_en_colors val;
hid_t hdf_en_colors = H5Tcreate(H5T_ENUM, sizeof(c_en_colors));
H5Tenum_insert(hdf_en_colors, "RED",   (val=RED,  &val));
H5Tenum_insert(hdf_en_colors, "GREEN", (val=GREEN,&val));
H5Tenum_insert(hdf_en_colors, "BLUE",  (val=BLUE, &val));
H5Tenum_insert(hdf_en_colors, "WHITE", (val=WHITE,&val));
H5Tenum_insert(hdf_en_colors, "BLACK", (val=BLACK,&val));
```

Name Changes between Applications

Occassionally two applicatons wish to exchange data but they use different names for the constants they exchange. For instance, an English and a Spanish program may want to communicate color names although they use different symbols in the C `enum` definitions. The communication is still possible although the applications must agree on common terms for the colors. The following example shows the Spanish code to read the values assuming that the applications have agreed that the color information will be exchanged using Enlish color names:

```
c_sp_colors val, data[1000];
hid_t hdf_sp_colors = H5Tcreate(H5T_ENUM, sizeof(c_sp_colors));
H5Tenum_insert(hdf_sp_colors, "RED",   (val=ROJO,   &val));
H5Tenum_insert(hdf_sp_colors, "GREEN", (val=VERDE,  &val));
H5Tenum_insert(hdf_sp_colors, "BLUE",  (val=AZUL,   &val));
H5Tenum_insert(hdf_sp_colors, "WHITE", (val=BLANCO, &val));
H5Tenum_insert(hdf_sp_colors, "BLACK", (val=NEGRO,  &val));

H5Dread(dataset, hdf_sp_colors, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);
```

Symbol Ordering across Applications

Since symbol ordering is completely determined by the integer values assigned to each symbol in the `enum` definition, ordering of `enum` symbols cannot be preserved across files like with HDF enumeration types. HDF can convert from one application’s integer values to the other’s so a symbol in one application’s C `enum` gets mapped to the same symbol in the other application’s C `enum`, but the relative order of the symbols is not preserved.

For example, an application may be defined to use the definition of `c_en_colors` defined above where `WHITE` is less than `BLACK`, but some other application might define the colors in some other order. If each application defines an HDF enumeration type based on that application’s C `enum` type then HDF will modify the integer values as data is communicated from one application to the other so that a `RED` value in the first application is also a `RED` value in the other application.

A case of this reordering of symbol names was also shown in the previous code snippet (as well as a change of language), where HDF changed the integer values so 0 (`RED`) in the input file became 4 (`ROJO`) in the `data` array. In the input file, `WHITE` was less than `BLACK`; in the application the opposite is true.

In fact, the ability to change the order of symbols is often convenient when the enumeration type is used only to group related symbols that don’t have any well defined order relationship.

Internationalization

The HDF enumeration type conversion features can also be used to provide internationalization of debugging output. A program written with the `c_en_colors` datatype could define a separate HDF datatype for languages such as English, Spanish, and French and cast the enumerated value to one of these HDF types to print the result.

```
c_en_colors val, *data=...;

hid_t hdf_sp_colors = H5Tcreate(H5T_ENUM, sizeof val);
H5Tenum_insert(hdf_sp_colors, "ROJO",   (val=RED,   &val));
H5Tenum_insert(hdf_sp_colors, "VERDE",  (val=GREEN, &val));
H5Tenum_insert(hdf_sp_colors, "AZUL",   (val=BLUE,  &val));
H5Tenum_insert(hdf_sp_colors, "BLANCO", (val=WHITE, &val));
H5Tenum_insert(hdf_sp_colors, "NEGRO",  (val=BLACK, &val));

hid_t hdf_fr_colors = H5Tcreate(H5T_ENUM, sizeof val);
H5Tenum_insert(hdf_fr_colors, "OUGE",  (val=RED,   &val));
H5Tenum_insert(hdf_fr_colors, "VERT",  (val=GREEN, &val));
H5Tenum_insert(hdf_fr_colors, "BLEU",  (val=BLUE,  &val));
H5Tenum_insert(hdf_fr_colors, "BLANC", (val=WHITE, &val));
H5Tenum_insert(hdf_fr_colors, "NOIR",  (val=BLACK, &val));

void
nameof(lang_t language, c_en_colors val, char *name, size_t size)
{
    switch (language) {
    case ENGLISH:
        H5Tenum_nameof(hdf_en_colors, &val, name, size);
        break;
    case SPANISH:
        H5Tenum_nameof(hdf_sp_colors, &val, name, size);
        break;
    case FRENCH:
        H5Tenum_nameof(hdf_fr_colors, &val, name, size);
        break;
    }
}
```

3.8.10. Goals That Have Been Met

The main goal of enumeration types is to provide communication of enumerated data using symbolic equivalence. That is, a symbol written to a dataset by one application should be read as the same symbol by some other application.

Architecture Independence	Two applications shall be able to exchange enumerated data even when the underlying integer values have different storage formats. HDF accomplishes this for enumeration types by building them upon integer types.
Preservation of Order Relationship	The relative order of symbols shall be preserved between two applications that use equivalent enumeration datatypes. Unlike numeric values that have an implicit ordering, enumerated data has an explicit order defined by the enumeration datatype and HDF records this order in the file.
Order Independence	An application shall be able to change the relative ordering of the symbols in an enumeration datatype. This is accomplished by defining a new type with different integer values and converting data from one type to the other.
Subsets	An application shall be able to read enumerated data from an archived dataset even after the application has defined additional members for the enumeration type. An application shall

	be able to write to a dataset when the dataset contains a superset of the members defined by the application. Similar rules apply for in-core conversions between enumerated datatypes.
Targetable	An application shall be able to target a particular architecture or application when storing enumerated data. This is accomplished by allowing non-native underlying integer types and converting the native data to non-native data.
Efficient Data Transfer	An application that defines a file dataset that corresponds to some native C enumerated data array shall be able to read and write to that dataset directly using only Posix read and write functions. HDF already optimizes this case for integers, so the same optimization will apply to enumerated data.
Efficient Storage	Enumerated data shall be stored in a manner which is space efficient. HDF stores the enumerated data as integers and allows the application to chose the size and format of those integers.

3.9. Variable-length Datatypes

3.9.1. Overview And Justification

Variable-length (VL) datatypes are sequences of an existing datatype (atomic, VL, or compound) which are not fixed in length from one dataset location to another. In essence, they are similar to C character strings -- a sequence of a type which is pointed to by a particular type of *pointer* -- although they are implemented more closely to FORTRAN strings by including an explicit length in the pointer instead of using a particular value to terminate the sequence.

VL datatypes are useful to the scientific community in many different ways, some of which are listed below:

- Ragged arrays: Multi-dimensional ragged arrays can be implemented with the last (fastest changing) dimension being ragged by using a VL datatype as the type of the element stored. (Or as a field in a compound datatype.)

- Fractal arrays: If a compound datatype has a VL field of another compound type with VL fields (a *nested* VL datatype), this can be used to implement ragged arrays of ragged arrays, to whatever nesting depth is required for the user.

- Polygon lists: A common storage requirement is to efficiently store arrays of polygons with different numbers of vertices. VL datatypes can be used to efficiently and succinctly describe an array of polygons with different numbers of vertices.

- Character strings: Perhaps the most common use of VL datatypes will be to store C-like VL character strings in dataset elements or as attributes of objects.

- Indices: An array of VL object references could be used as an index to all the objects in a file which contain a particular sequence of dataset values. Perhaps an array something like the following:

```
Value1: Object1, Object3,  Object9
Value2: Object0, Object12, Object14, Object21, Object22
Value3: Object2
Value4: <none>
Value5: Object1, Object10, Object12
    .
    .
```

- Object Tracking: An array of VL dataset region references can be used as a method of tracking objects or features appearing in a sequence of datasets. Perhaps an array of them would look like:

```
Feature1: Dataset1:Region,  Dataset3:Region,  Dataset9:Region
Feature2: Dataset0:Region,  Dataset12:Region, Dataset14:Region,
          Dataset21:Region, Dataset22:Region
Feature3: Dataset2:Region
Feature4: <none>
Feature5: Dataset1:Region,  Dataset10:Region, Dataset12:Region
    .
    .
```

3.9.2. Variable-length Datatype Memory Management

With each element possibly being of different sequence lengths for a dataset with a VL datatype, the memory for the VL datatype must be dynamically allocated. Currently there are two methods of managing the memory for VL datatypes: the standard C malloc/free memory allocation routines or a method of calling user-defined memory management routines to allocate or free memory. Since the memory allocated when reading (or writing) may be complicated to release, an HDF5 routine is provided to traverse a memory buffer and free the VL datatype information without leaking memory.

Variable-length datatypes cannot be divided

VL datatypes are designed so that they cannot be subdivided by the library with selections, etc. This design was chosen due to the complexities in specifying selections on each VL element of a dataset through a selection API that is easy to understand. Also, the selection APIs work on dataspaces, not on datatypes. At some point in time, we may want to create a way for dataspaces to have VL components to them and we would need to allow selections of those VL regions, but that is beyond the scope of this document.

What happens if the library runs out of memory while reading?

It is possible for a call to `H5Dread` to fail while reading in VL datatype information if the memory required exceeds that which is available. In this case, the `H5Dread` call will fail gracefully and any VL data which has been allocated prior to the memory shortage will be returned to the system via the memory management routines detailed below. It may be possible to design a *partial read* API function at a later date, if demand for such a function warrants.

Strings as variable-length datatypes

Since character strings are a special case of VL data that is implemented in many different ways on different machines and in different programming languages, they are handled somewhat differently from other VL datatypes in HDF5.

HDF5 has native VL strings for each language API, which are stored the same way on disk, but are exported through each language API in a natural way for that language. When retrieving VL strings from a dataset, users may choose to have them stored in memory as a native VL string or in HDF5's `hvl_t` struct for VL datatypes.

VL strings may be created in one of two ways: by creating a VL datatype with a base type of `H5T_NATIVE_ASCII`, `H5T_NATIVE_UNICODE`, etc., or by creating a string datatype and setting its length to `H5T_STRING_VARIABLE`. The second method is used to access native VL strings in memory. The library will convert between the two types, but they are stored on disk using different datatypes and have different memory representations.

Multi-byte character representations, such as UNICODE or *wide* characters in C/C++, will need the appropriate character and string datatypes created so that they can be described properly through the datatype API. Additional conversions between these types and the current ASCII characters will also be required.

Variable-width character strings (which might be compressed data or some other encoding) are not currently handled by this design. We will evaluate how to implement them based on user feedback.

3.9.3. Variable-length Datatype API

Creation

VL datatypes are created with the `H5Tvlen_create()` function as follows:

> *type_id* = `H5Tvlen_create`(*hid_t* `base_type_id`);

The base datatype will be the datatype that the sequence is composed of, characters for character strings, vertex coordinates for polygon lists, etc. The base datatype specified for the VL datatype can be of any HDF5 datatype, including another VL datatype, a compound datatype, or an atomic datatype.

Query base datatype of VL datatype

It may be necessary to know the base datatype of a VL datatype before memory is allocated, etc. The base datatype is queried with the `H5Tget_super()` function, described in the H5T documentation.

Query minimum memory required for VL information

It order to predict the memory usage that `H5Dread` may need to allocate to store VL data while reading the data, the `H5Dget_vlen_size()` function is provided:

> *herr_t* `H5Dget_vlen_buf_size`(*hid_t* `dataset_id`, *hid_t* `type_id`, *hid_t* `space_id`, *hsize_t* `*size`)

(This function is not implemented in Release 1.2.)

This routine checks the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory. The `*size` value is modified according to how many bytes are required to store the VL data in memory.

Specifying how to manage memory for the VL datatype

The memory management method is determined by dataset transfer properties passed into the `H5Dread` and `H5Dwrite` functions with the dataset transfer property list.

Default memory management is set by using `H5P_DEFAULT` for the dataset transfer property list identifier. If `H5P_DEFAULT` is used with `H5Dread`, the system `malloc` and `free` calls will be used for allocating and freeing memory. In such a case, `H5P_DEFAULT` should also be passed as the property list identifier to `H5Dvlen_reclaim`.

The rest of this subsection is relevant only to those who choose *not* to use default memory management.

The user can choose whether to use the system `malloc` and `free` calls or user-defined, or custom, memory management functions. If user-defined memory management functions are to be used, the memory allocation and free routines must be defined via `H5Pset_vlen_mem_manager()`, as follows:

> *herr_t* `H5Pset_vlen_mem_manager`(*hid_t* `plist_id`, *H5MM_allocate_t* `alloc`, *void* `*alloc_info`, *H5MM_free_t* `free`, *void* `*free_info`)

The `alloc` and `free` parameters identify the memory management routines to be used. If the user has defined custom memory management routines, `alloc` and/or `free` should be set to make those routine calls (i.e., the name of the routine is used as the value of the parameter); if the user prefers to use the system’s `malloc` and/or `free`, the `alloc` and `free` parameters, respectively, should be set to `NULL`

The prototypes for the user-defined functions would appear as follows:

> `typedef` *void* `*(*H5MM_allocate_t)`(*size_t* `size`, *void* `*info`) ;

`typedef` *void* (`*H5MM_free_t`)(*void* `*mem`, *void* `*free_info`) ;

The `alloc_info` and `free_info` parameters can be used to pass along any required information to the user's memory management routines.

In summary, if the user has defined custom memory management routines, the name(s) of the routines are passed in the `alloc` and `free` parameters and the custom routines' parameters are passed in the `alloc_info` and `free_info` parameters. If the user wishes to use the system `malloc` and `free` functions, the `alloc` and/or `free` parameters are set to `NULL` and the `alloc_info` and `free_info` parameters are ignored.

Recovering memory from VL buffers read in

The complex memory buffers created for a VL datatype may be reclaimed with the `H5Dvlen_reclaim()` function call, as follows:

herr_t `H5Dvlen_reclaim`(*hid_t* `type_id`, *hid_t* `space_id`, *hid_t* `plist_id`, *void* `*buf`);

The `type_id` must be the datatype stored in the buffer, `space_id` describes the selection for the memory buffer to free the VL datatypes within, `plist_id` is the dataset transfer property list which was used for the I/O transfer to create the buffer, and `buf` is the pointer to the buffer to free the VL memory within. The VL structures (`hvl_t`) in the user's buffer are modified to zero out the VL information after it has been freed.

If nested VL datatypes were used to create the buffer, this routine frees them from the bottom up, releasing all the memory without creating memory leaks.

3.9.4. Code Examples

The following example creates the following one-dimensional array of size 4 of variable-length datatype.

```
0 10 20 30
  11 21 31
     22 32
        33
```

Each element of the VL datatype is of H5T_NATIVE_UINT type.

The array is stored in the dataset and then read back into memory. Default memory management routines are used for writing the VL data. Custom memory management routines are used for reading the VL data and reclaiming memory space.

Example: Variable-length Datatypes

```

#include

#define FILE   "tvltypes.h5"
#define MAX(X,Y)        ((X)>(Y)?(X):(Y))

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME  "Space1"
#define SPACE1_RANK       1
#define SPACE1_DIM1       4

void *vltypes_alloc_custom(size_t size, void *info);
void vltypes_free_custom(void *mem, void *info);

/****************************************************************
```

```
**
**  vltypes_alloc_custom():  VL datatype custom memory
**      allocation routine.  This routine just uses malloc to
**      allocate the memory and increments the amount of memory
**      allocated.
**
****************************************************************/
void *vltypes_alloc_custom(size_t size, void *info)
{
    void *ret_value=NULL;       /* Pointer to return */
    int *mem_used=(int *)info;  /* Get the pointer to the memory used */
    size_t extra;               /* Extra space needed */

    /*
     *  This weird contortion is required on the DEC Alpha to keep the
     *  alignment correct.
     */
    extra=MAX(sizeof(void *),sizeof(int));

    if((ret_value=malloc(extra+size))!=NULL) {
        *(int *)ret_value=size;
        *mem_used+=size;
    } /* end if */
    ret_value=((unsigned char *)ret_value)+extra;
    return(ret_value);
}

/****************************************************************
**
**  vltypes_free_custom(): VL datatype custom memory
**      allocation routine.  This routine just uses free to
**      release the memory and decrements the amount of memory
**      allocated.
**
****************************************************************/
void vltypes_free_custom(void *_mem, void *info)
{
    unsigned char *mem;
    int *mem_used=(int *)info;  /* Get the pointer to the memory used */
    size_t extra;               /* Extra space needed */

    /*
     *  This weird contortion is required on the DEC Alpha to keep the
     *  alignment correct.
     */
    extra=MAX(sizeof(void *),sizeof(int));

    if(_mem!=NULL) {
        mem=((unsigned char *)_mem)-extra;
        *mem_used-=*(int *)mem;
        free(mem);
    } /* end if */
}

int main(void)
{
    hvl_t wdata[SPACE1_DIM1];   /* Information to write */
    hvl_t rdata[SPACE1_DIM1];   /* Information read in */
    hid_t                 fid1;            /* HDF5 File IDs           */
    hid_t                 dataset; /* Dataset ID                      */
    hid_t                 sid1;      /* Dataspace ID                          */
    hid_t                 tid1;      /* Datatype ID                   */
    hid_t       xfer_pid;   /* Dataset transfer property list ID */
    hsize_t                dims1[] = {SPACE1_DIM1};
    uint       i,j;        /* counting variables */
    int         mem_used=0; /* Memory used during allocation */
    herr_t                 ret;             /* Generic return value            */


    /*
     * Allocate and initialize VL data to write
     */
    for(i=0; i<SPACE1_DIM1; i++) {

        wdata[i].p=malloc((i+1)*sizeof(unsigned int));
        wdata[i].len=i+1;
        for(j=0; j<(i+1); j++)
            ((unsigned int *)wdata[i].p)[j]=i*10+j;
    } /* end for */

    /*
     * Create file.
     */
    fid1 = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create dataspace for datasets.
     */
    sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

    /*
     * Create a datatype to refer to.
     */
    tid1 = H5Tvlen_create (H5T_NATIVE_UINT);

    /*
     * Create a dataset.
     */
    dataset=H5Dcreate(fid1,"Dataset1",tid1,sid1,H5P_DEFAULT);

    /*
     * Write dataset to disk.
     */
    ret=H5Dwrite(dataset,tid1,H5S_ALL,H5S_ALL,H5P_DEFAULT,wdata);

    /*
     * Change to the custom memory allocation routines for reading VL data
     */
    xfer_pid=H5Pcreate(H5P_DATASET_XFER);

    ret=H5Pset_vlen_mem_manager(xfer_pid,vltypes_alloc_custom,
                                &mem_used,vltypes_free_custom,&mem_used);

    /*
     * Read dataset from disk. vltypes_alloc_custom and
     * will be used to manage memory.
     */
    ret=H5Dread(dataset,tid1,H5S_ALL,H5S_ALL,xfer_pid,rdata);

    /*
     * Display data read in
     */
    for(i=0; i<SPACE1_DIM1; i++) {
        printf("%d-th element length is %d \n", i, (unsigned) rdata[i].len);
        for(j=0; j<rdata[i].len; j++) {
            printf(" %d ",((unsigned int *)rdata[i].p)[j] );
            }
        printf("\n");
    } /* end for */

    /*
     * Reclaim the read VL data. vltypes_free_custom will be used
     * to reclaim the space.
     */
    ret=H5Dvlen_reclaim(tid1,sid1,xfer_pid,rdata);


    /*
     * Reclaim the write VL data.  C language free function will be used
     * to reclaim space.
     */
    ret=H5Dvlen_reclaim(tid1,sid1,H5P_DEFAULT,wdata);

    /*
     * Close Dataset
     */
    ret = H5Dclose(dataset);

    /*
     * Close datatype
     */
    ret = H5Tclose(tid1);

    /*
     * Close disk dataspace
     */
    ret = H5Sclose(sid1);

    /*
     * Close dataset transfer property list
     */
    ret = H5Pclose(xfer_pid);

    /*
     * Close file
     */
    ret = H5Fclose(fid1);

}
```

And the output from this sample code would be as follows:

Example: Variable-length Datatypes, Sample Output

```
0-th element length is 1
0
1-th element length is 2
10  11
2-th element length is 3
20  21  22
3-th element length is 4
30  31  32  33
```

For further samples of VL datatype code, see the tests in `test/tvltypes.c` in the HDF5 distribution.

3.10. Sharing Datatypes among Datasets

If a file has lots of datasets which have a common datatype then the file could be made smaller by having all the datasets share a single datatype. Instead of storing a copy of the datatype in each dataset object header, a single datatype is stored and the object headers point to it. The space savings is probably only significant for datasets with a compound datatype since the simple datatypes can be described with just a few bytes anyway.

To create a bunch of datasets that share a single datatype just create the datasets with a committed (named) datatype.

Example: Shared Types

To create two datasets that share a common datatype one just commits the datatype, giving it a name, and then uses that datatype to create the datasets.

```
hid_t t1 = ...some transient type...;
H5Tcommit (file, "shared_type", t1);
hid_t dset1 = H5Dcreate (file, "dset1", t1, space, H5P_DEFAULT);
hid_t dset2 = H5Dcreate (file, "dset2", t1, space, H5P_DEFAULT);
```

And to create two additional datasets later which share the same type as the first two datasets:

```
hid_t dset1 = H5Dopen (file, "dset1");
hid_t t2 = H5Dget_type (dset1);
hid_t dset3 = H5Dcreate (file, "dset3", t2, space, H5P_DEFAULT);
hid_t dset4 = H5Dcreate (file, "dset4", t2, space, H5P_DEFAULT);
```

3.11. Data Conversion

The library is capable of converting data from one type to another and does so automatically when reading or writing the raw data of a dataset, attribute data, or fill values. The application can also change the type of data stored in an array.

In order to insure that data conversion exceeds disk I/O rates, common data conversion paths can be hand-tuned and optimized for performance. The library contains very efficient code for conversions between most native datatypes and a few non-native datatypes, but if a hand-tuned conversion function is not available, then the library falls back to a slower but more general conversion function. The application programmer can define additional conversion functions when the libraries repertoire is insufficient. In fact, if an application does define a conversion function which would be of general interest, we request that the function be submitted to the HDF5 development team for inclusion in the library.

Note: The HDF5 library contains a deliberately limited set of conversion routines. It can convert from one integer format to another, from one floating point format to another, and from one struct to another. It can also perform byte swapping when the source and destination types are otherwise the same. The library does not contain any functions for converting data between integer and floating point formats. It is anticipated that some users will find it necessary to develop float to integer or integer to float conversion functions at the application level; users are invited to submit those functions to be considered for inclusion in future versions of the library.

A conversion path contains a source and destination datatype and each path contains a *hard* conversion function and/or a *soft* conversion function. The only difference between hard and soft functions is the way in which the library chooses which function applies: A hard function applies to a specific conversion path while a soft function may apply to multiple paths. When both hard and soft functions apply to a conversion path, then the hard function is favored and when multiple soft functions apply, the one defined last is favored.

A data conversion function is of type `H5T_conv_t` which is defined as:

```
typedef herr_t (*H5T_conv_t)(hid_t src_type,
                             hid_t dest_type,
                                H5T_cdata_t *cdata,
                                size_t nelmts,
                                void *buffer,
                             void *background);
```

The conversion function is called with the source and destination datatypes (*src_type* and *dst_type*), path-constant data (*cdata*), the number of instances of the datatype to convert (*nelmts*), a buffer which initially contains an array of data having the source type and on return will contain an array of data having the destination type (*buffer*), and a temporary or background buffer (*background*). Functions return a negative value on failure and some other value on success.

The `command` field of the *cdata* argument determines what happens within the conversion function. It's values can be:

`H5T_CONV_INIT`

This command is to hard conversion functions when they're registered or soft conversion functions when the library is determining if a conversion can be used for a particular path. The *src_type* and *dst_type* are the end-points of the path being queried and *cdata* is all zero. The library should examine the source and destination types and return zero if the conversion is possible and negative otherwise (hard conversions need not do this since they've presumably been registered only on paths they support). If the conversion is possible the library may allocate and initialize private data and assign the pointer to the `priv` field of *cdata* (or private data can be initialized later). It should also initialize the `need_bkg` field described below. The *buf* and *background* pointers will be null pointers.

`H5T_CONV_CONV`

This command indicates that data points should be converted. The conversion function should initialize the `priv` field of *cdata* if it wasn't initialize during the `H5T_CONV_INIT` command and then convert *nelmts* instances of the *src_type* to the *dst_type*. The *buffer* serves as both input and output. The *background* buffer is supplied according to the value of the `need_bkg` field of *cdata* (the values are described below).

`H5T_CONV_FREE`

The conversion function is about to be removed from some path and the private data (the *`cdata`*`-priv` pointer) should be freed and set to null. All other pointer arguments are null, the *src_type* and *dst_type* are invalid (negative), and the *nelmts* argument is zero.

Others...

Other commands might be implemented later and conversion functions that don't support those commands should return a negative value.

Whether a background buffer is supplied to a conversion function, and whether the background buffer is initialized depends on the value of *`cdata`*`-need_bkg` which the conversion function should have initialized during the H5T_CONV_INIT command. It can have one of these values:

`H5T_BKG_NONE`

No background buffer will be supplied to the conversion function. This is the default.

`H5T_BKG_TEMP`

A background buffer will be supplied but it will not be initialized. This is useful for those functions requiring some extra buffer space as the buffer can probably be allocated more efficiently by the library (the application can supply the buffer as part of the dataset transfer property list).

`H5T_BKG_YES`

An initialized background buffer is passed to the conversion function. The buffer is initialized with the current values of the destination for the data which is passed in through the *buffer* argument. It can be used to "fill in between the cracks". For instance, if the destination type is a compound datatype and we are initializing only part of the compound datatype from the source type then the background buffer can be used to initialize the other part of the destination.

The `recalc` field of *cdata* is set when the conversion path table changes. It can be used by conversion function that cache other conversion paths so they know when their cache needs to be recomputed.

Once a conversion function is written it can be registered and unregistered with these functions:

`herr_t H5Tregister(H5T_pers_t` *`pers`*`, const char *`*`name`*`, hid_t` *`src_type`*`, hid_t` *`dest_type`*`, H5T_conv_t` *`func`*`)`

Once a conversion function is written, the library must be notified so it can be used. The function can be registered as a hard (`H5T_PERS_HARD`) or soft (`H5T_PERS_SOFT`) conversion depending on the value of *pers*, displacing any previous conversions for all applicable paths. The *name* is used only for debugging but must be supplied. If *pers* is `H5T_PERS_SOFT` then only the type classes of the *src_type* and *dst_type* are used. For instance, to register a general soft conversion function that can be applied to any integer to integer conversion one could say: `H5Tregister(H5T_PERS_SOFT, "i2i", H5T_NATIVE_INT, H5T_NATIVE_INT, convert_i2i)`. One special conversion path called the "no-op" conversion path is always defined by the library and used as the conversion function when no data transformation is necessary. The application can redefine this path by specifying a new hard conversion function with a negative value for both the source and destination datatypes, but the library might not call the function under certain circumstances.

`herr_t H5Tunregister (H5T_pers_t` *`pers`*`, const char *`*`name`*`, hid_t` *`src_type`*`, hid_t` *`dest_type`*`, H5T_conv_t` *`func`*`)`

Any conversion path or function that matches the critera specified by a call to this function is removed from the type conversion table. All fields have the same interpretation as for `H5Tregister()` with the added feature that any (or all) may be wild cards. The `H5T_PERS_DONTCARE` constant should be used to indicate a wild card for the *pers* argument. The wild card *name* is the null pointer or empty string, the wild card for the *src_type* and *dest_type* arguments is any negative value, and the wild card for the *func* argument is the null pointer. The special no-op conversion path is never removed by this function.

Example: A conversion function

Here's an example application-level function that converts Cray `unsigned short` to any other 16-bit unsigned big-endian integer. A cray `short` is a big-endian value which has 32 bits of precision in the high-order bits of a 64-bit word.

```
1 typedef struct {
2     size_t dst_size;
3     int direction;
4 } cray_ushort2be_t;
5
6 herr_t
```

```
cray_ushort2be (hid_t src, hid_t dst,
                H5T_cdata_t *cdata,
                size_t nelmts, void *buf,
                const void *background)
{
    unsigned char *src = (unsigned char *)buf;
    unsigned char *dst = src;
    cray_ushort2be_t *priv = NULL;

    switch (cdata->command) {
    case H5T_CONV_INIT:
        /*
         * We are being queried to see if we handle this
         * conversion.  We can handle conversion from
         * Cray unsigned short to any other big-endian
         * unsigned integer that doesn’t have padding.
         */
        if (!H5Tequal (src, H5T_CRAY_USHORT) ||
            H5T_ORDER_BE != H5Tget_order (dst) ||
            H5T_SGN_NONE != H5Tget_signed (dst) ||
            8*H5Tget_size (dst) != H5Tget_precision (dst)) {
            return -1;
        }

        /*
         * Initialize private data.  If the destination size
         * is larger than the source size, then we must
         * process the elements from right to left.
         */
        cdata->priv = priv = malloc (sizeof(cray_ushort2be_t));
        priv->dst_size = H5Tget_size (dst);
        if (priv->dst_size>8) {
            priv->direction = -1;
        } else {
            priv->direction = 1;
        }
        break;

    case H5T_CONV_FREE:
        /*
         * Free private data.
         */
        free (cdata->priv);
        cdata->priv = NULL;
        break;

    case H5T_CONV_CONV:
        /*
         * Convert each element, watch out for overlap src
         * with dst on the left-most element of the buffer.
         */
        priv = (cray_ushort2be_t *)(cdata->priv);
        if (priv->direction<0) {
            src += (nelmts - 1) * 8;
            dst += (nelmts - 1) * dst_size;
        }
        for (i=0; i<n; i++) {
            if (src==dst && dst_size<4) {
                for (j=0; j<dst_size; j++) {
                    dst[j] = src[j+4-dst_size];
                }
            } else {
                for (j=0; j<4 && j<dst_size; j++) {
                    dst[dst_size-(j+1)] = src[3-j];
                }
                for (j=4; j<dst_size; j++) {
                    dst[dst_size-(j+1)] = 0;
                }
            }
            src += 8 * direction;
            dst += dst_size * direction;
        }
        break;

    default:
        /*
         * Unknown command.
         */
        return -1;
    }
    return 0;
}
```

The *background* argument is ignored since it's generally not applicable to atomic datatypes.

Example: Soft Registration

The convesion function described in the previous example applies to more than one conversion path. Instead of enumerating all possible paths, we register it as a soft function and allow it to decide which paths it can handle.

```
H5Tregister(H5T_PERS_SOFT, "cus2be",
            H5T_NATIVE_INT, H5T_NATIVE_INT,
            cray_ushort2be);
```

This causes it to be consulted for any conversion from an integer type to another integer type. The first argument is just a short identifier which will be printed with the datatype conversion statistics.

NOTE: The idea of a master soft list and being able to query conversion functions for their abilities tries to overcome problems we saw with AIO. Namely, that there was a dichotomy between generic conversions and specific conversions that made it very difficult to write a conversion function that operated on, say, integers of any size and order as long as they don't have zero padding. The AIO mechanism required such a function to be explicitly registered (like `H5Tregister_hard()`) for each an every possible conversion path whether that conversion path was actually used or not.

Last modified: 14 October 1999

4. The Dataspace Interface (H5S)

4.1. Introduction

The dataspace interface (H5S) provides a mechanism to describe the positions of the elements of a dataset and is designed in such a way as to allow new features to be easily added without disrupting applications that use the dataspace interface. A dataset (defined with the dataset interface) is composed of a collection of raw data points of homogeneous type, defined in the datatype (H5T) interface, organized according to the dataspace with this interface.

A dataspace describes the locations that dataset elements are located at. A dataspace is either a regular N-dimensional array of data points, called a *simple* dataspace, or a more general collection of data points organized in another manner, called a *complex* dataspace. A *scalar* dataspace is a special case of the *simple* data space and is defined to be a 0-dimensional single data point in size. Currently only *scalar* and *simple* dataspaces are supported with this version of the H5S interface. *Complex* dataspaces will be defined and implemented in a future version. *Complex* dataspaces are intended to be used for such structures which are awkward to express in *simple* dataspaces, such as irregularly gridded data or adaptive mesh refinement data. This interface provides functions to set and query properties of a dataspace.

Operations on a dataspace include defining or extending the extent of the dataspace, selecting portions of the dataspace for I/O and storing the dataspaces in the file. The extent of a dataspace is the range of coordinates over which dataset elements are defined and stored. Dataspace selections are subsets of the extent (up to the entire extent) which are selected for some operation.

For example, a 2-dimensional dataspace with an extent of 10 by 10 may have the following very simple selection:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	-	-	-	-	-	-	-	-	-	-
1	-	X	X	X	-	-	-	-	-	-
2	-	X	X	X	-	-	-	-	-	-
3	-	X	X	X	-	-	-	-	-	-
4	-	X	X	X	-	-	-	-	-	-
5	-	X	X	X	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-

Example 1: Contiguous rectangular selection

Or, a more complex selection may be defined:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	-	-	-	-	-	-	-	-	-	-
1	-	X	X	X	-	-	X	-	-	-
2	-	X	-	X	-	-	-	-	-	-
3	-	X	-	X	-	-	X	-	-	-
4	-	X	-	X	-	-	-	-	-	-
5	-	X	X	X	-	-	X	-	-	-
6	-	-	-	-	-	-	-	-	-	-
7	-	-	X	X	X	X	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-

Example 2: Non-contiguous selection

Selections within dataspaces have an offset within the extent which is used to locate the selection within the extent of the dataspace. Selection offsets default to 0 in each dimension, but may be changed to move the selection within a dataspace. In example 2 above, if the offset was changed to 1,1, the selection would look like this:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-
2	-	-	X	X	X	-	-	X	-	-
3	-	-	X	-	X	-	-	-	-	-
4	-	-	X	-	X	-	-	X	-	-
5	-	-	X	-	X	-	-	-	-	-
6	-	-	X	X	X	-	-	X	-	-
7	-	-	-	-	-	-	-	-	-	-
8	-	-	-	X	X	X	X	-	-	-
9	-	-	-	-	-	-	-	-	-	-

Example 3: Non-contiguous selection with 1,1 offset

Selections also have a linearization ordering of the points selected (defaulting to "C" order, ie. last dimension changing fastest). The linearization order may be specified for each point or it may be chosen by the axis of the dataspace. For example, with the default "C" ordering, example 1’s selected points are iterated through in this order: (1,1), (1,2), (1,3), (2,1), (2,2), etc. With "FORTRAN" ordering, example 1’s selected points would be iterated through in this order: (1,1), (2,1), (3,1), (4,1), (5,1), (1,2), (2,2), etc.

A dataspace may be stored in the file as a permanent object, to allow many datasets to use a commonly defined dataspace. Dataspaces with extendable extents (ie. unlimited dimensions) are not able to be stored as permanent dataspaces.

Dataspaces may be created using an existing permanent dataspace as a container to locate the new dataspace within. These dataspaces are complete dataspaces and may be used to define datasets. A dataspaces with a "parent" can be queried to determine the parent dataspace and the location within the parent. These dataspaces must currently be the same number of dimensions as the parent dataspace.

4.2. General Dataspace Operations

The functions defined in this section operate on dataspaces as a whole. New dataspaces can be created from scratch or copied from existing data spaces. When a dataspace is no longer needed its resources should be released by calling `H5Sclose()`.

`hid_t H5Screate(H5S_class_t` *`type`*`)`

This function creates a new dataspace of a particular *type*. The types currently supported are H5S_SCALAR and H5S_SIMPLE; others are planned to be added later.

`hid_t H5Sopen(hid_t` *`location`*`, const char *`*`name`*`)`

This function opens a permanent dataspace for use in an application. The *location* argument is a file or group ID and *name* is an absolute or relative path to the permanent dataspace. The dataspace ID which is returned is a handle to a permanent dataspace which can't be modified.

`hid_t H5Scopy (hid_t` *`space`*`)`

This function creates a new dataspace which is an exact copy of the dataspace *space*.

`hid_t H5Ssubspace (hid_t` *`space`*`)`

This function uses the currently defined selection and offset in *space* to create a dataspace which is located within *space*. The *space* dataspace must be a sharable dataspace located in the file, not a dataspace for a dataset. The relationship of the new dataspace within the existing dataspace is preserved when the new dataspace is used to create datasets. Currently, only subspaces which are equivalent to simple dataspaces (ie. rectangular contiguous areas) are allowed. A subspace is not "simplified" or reduced in the number of dimensions used if the selection is "flat" in one dimension, they always have the same number of dimensions as their parent dataspace.

`herr_t H5Scommit (hid_t` *`location`*`, const char *`*`name`*`, hid_t` *`space`*`)`

The dataspaces specified with *space* is stored in the file specified by *location*. *Location* may be either a file or group handle and *name* is an absolute or relative path to the location to store the dataspace. After this call, the dataspace is permanent and can't be modified.

`herr_t H5Sclose (hid_t` *`space`*`)`

Releases resources associated with a dataspace. Subsequent use of the dataspace identifier after this call is undefined.

4.3. Dataspace Extent Operations

These functions operate on the extent portion of a dataspace.

`herr_t H5Sset_extent_simple (hid_t` *`space`*`, int` *`rank`*`, const hsize_t *`*`current_size`*`, const hsize_t *`*`maximum_size`*`)`

Sets or resets the size of an existing dataspace, where *rank* is the dimensionality, or number of dimensions, of the dataspace. *current_size* is an array of size *rank* which contains the new size of each dimension in the dataspace. *maximum_size* is an array of size *rank* which contains the maximum size of each dimension in the dataspace. Any previous extent is removed from the dataspace, the dataspace type is set to H5S_SIMPLE and the extent is set as specified.

`herr_t H5Sset_extent_none (hid_t` *`space`*`)`

Removes the extent from a dataspace and sets the type to H5S_NO_CLASS.

`herr_t H5Sextent_copy (hid_t` *`dest_space`*`, hid_t` *`source_space`*`)`

Copies the extent from *source_space* to *dest_space*, which may change the type of the dataspace. Returns non-negative on success, negative on failure.

`hsize_t H5Sget_simple_extent_npoints (hid_t` *`space`*`)`

This function determines the number of elements in a dataspace. For example, a simple 3-dimensional dataspace with dimensions 2, 3 and 4 would have 24 elements. Returns the number of elements in the dataspace, negative on failure.

`int H5Sget_simple_extent_ndims (hid_t` *`space`*`)`

This function determines the dimensionality (or rank) of a dataspace. Returns the number of dimensions in the dataspace, negative on failure.

`herr_t H5Sget_simple_extent_dims (hid_t` *`space`*`, hsize_t *`*`dims`*`, hsize_t *`*`max`*`)`

The function retrieves the size of the extent of the dataspace *space* by placing the size of each dimension in the array *dims*. Also retrieves the size of the maximum extent of the dataspace, placing the results in *max*. Returns non-negative on success, negative on failure.

4.4. Dataspace Selection Operations

Selections are maintained separately from extents in dataspaces and operations on the selection of a dataspace do not affect the extent of the dataspace. Selections are independent of extent type and the boundaries of selections are reconciled with the extent at the time of the data transfer. Selection offsets apply a selection to a location within an extent, allowing the same selection to be moved within the extent without requiring a new selection to be specified. Offsets default to 0 when the dataspace is created. Offsets are applied when an I/O transfer is performed (and checked during calls to H5Sselect_valid). Selections have an iteration order for the points selected, which can be any permutation of the dimensions involved (defaulting to 'C' array order) or a specific order for the selected points, for selections composed of single array elements with H5Sselect_elements. Selections can also be copied or combined together in various ways with H5Sselect_op. Further methods of selecting portions of a dataspace may be added in the future.

`herr_t H5Sselect_hyperslab (hid_t` *`space`*`, h5s_selopt_t` *`op`*`, const hssize_t *` *`start`*`, const hsize_t *` *`stride`*`, const hsize_t *` *`count`*`, const hsize_t *` *`block`*`)`

This function selects a hyperslab region to add to the current selected region for the *space* dataspace. The *start*, *stride*, *count* and *block* arrays must be the same size as the rank of the dataspace. The selection operator *op* determines how the new selection is to be combined with the already existing selection for the dataspace. Currently, The following operators are supported:

`H5S_SELECT_SET`	Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
`H5S_SELECT_OR`	Adds the new selection to the existing selection.

The *start* array determines the starting coordinates of the hyperslab to select. The *stride* array chooses array locations from the dataspace with each value in the *stride* array determining how many elements to move in each dimension. Setting a value in the *stride* array to 1 moves to each element in that dimension of the dataspace, setting a value of 2 in a location in the *stride* array moves to every other element in that dimension of the dataspace. In other words, the *stride* determines the number of elements to move from the *start* location in each dimension. Stride values of 0 are not allowed. If the *stride* parameter is `NULL`, a contiguous hyperslab is selected (as if each value in the *stride* array was set to all 1's). The *count* array determines how many blocks to select from the dataspace, in each dimension. The *block* array determines the size of the element block selected from the dataspace. If the *block* parameter is set to `NULL`, the block size defaults to a single element in each dimension (as if the *block* array was set to all 1's).

For example, in a 2-dimensional dataspace, setting *start* to [1,1], *stride* to [4,4], *count* to [3,7] and *block* to [2,2] selects 21 2x2 blocks of array elements starting with location (1,1) and selecting blocks at locations (1,1), (5,1), (9,1), (1,5), (5,5), etc.

Regions selected with this function call default to 'C' order iteration when I/O is performed.

`herr_t H5Sselect_elements (hid_t` *`space`*`, h5s_selopt_t` *`op`*`, const size_t` *`num_elements`*`, const hssize_t *`*`coord`*`[])`

This function selects array elements to be included in the selection for the *space* dataspace. The number of elements selected must be set with the *num_elements*. The *coord* array is a two-dimensional array of size <dataspace rank> by <*num_elements*> in size (ie. a list of coordinates in the array). The order of the element coordinates in the *coord* array also specifies the order that the array elements are iterated through when I/O is performed. Duplicate coordinate locations are not checked for.

The selection operator *op* determines how the new selection is to be combined with the already existing selection for the dataspace. The following operators are supported:

`H5S_SELECT_SET`	Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
`H5S_SELECT_OR`	Adds the new selection to the existing selection.

When operators other than H5S_SELECT_SET are used to combine a new selection with an existing selection, the selection ordering is reset to 'C' array ordering.

`herr_t H5Sselect_all (hid_t` *`space`*`)`

This function selects the special H5S_SELECT_ALL region for the *space* dataspace. H5S_SELECT_ALL selects the entire dataspace for any dataspace is is applied to.

`herr_t H5Sselect_none (hid_t` *`space`*`)`

This function resets the selection region for the *space* dataspace not to include any elements.

`herr_t H5Sselect_op (hid_t` *`space1`*`, h5s_selopt_t` *`op`*`, hid_t` *`space2`*`)`

Uses *space2* to perform an operation on *space1*. The valid operations for *op* are:

H5S_SELECT_COPY

Copies the selection from *space2* into *space1*, removing any previously defined selection for *space1*. The selection order and offset are also copied to *space1*

H5S_SELECT_UNION

Performs a set union of the selection of the dataspace *space2* with the selection from the dataspace *space1*, with the result being stored in *space1*. The selection order for *space1* is reset to 'C' order.

H5S_SELECT_INTERSECT

Performs an set intersection of the selection from *space2* with *space1*, with the result being stored in *space1*. The selection order for *space1* is reset to 'C' order.

H5S_SELECT_DIFFERENCE

Performs a set difference of the selection from *space2* with *space1*, with the result being stored in *space1*. The selection order for *space1* is reset to 'C' order.

`herr_t H5Sselect_order (hid_t` *`space`*`, hsize_t` *`perm_vector`*`[])`

This function selects the order to iterate through the dimensions of a dataspace when performing I/O on a selection. If a specific order has already been selected for the selection with H5Sselect_elements, this function will remove it and use a dimension oriented ordering on the selected elements. The elements of the *perm_vector* array must be unique and between 0 and the rank of the dataspace, minus 1. The order of the elements in *perm_vector* specify the order to iterate through the selection for each dimension of the dataspace. To iterate through a 3-dimensional dataspace selection in 'C' order, specify the elements of the *perm_vector* as [0, 1, 2], for FORTRAN order they would be [2, 1, 0]. Other orderings, such as [1, 2, 0] are also possible, but may execute slower.

`htri_t H5Sselect_valid (hid_t space)`

This function verifies that the selection for a dataspace is within the extent of the dataspace, if the currently set offset for the dataspace is used. Returns TRUE if the selection is contained within the extent, FALSE if it is not contained within the extent and FAIL on error conditions (such as if the selection or extent is not defined).

`hsize_t H5Sget_select_npoints (hid_t` *`space`*`)`

This function determines the number of elements in the current selection of a dataspace.

`herr_t H5Soffset_simple (hid_t` *`space`*`, const hssize_t *` *`offset`*`)`

Sets the offset of a simple dataspace *space*. The *offset* array must be the same number of elements as the number of dimensions for the dataspace. If the *offset* array is set to `NULL`, the offset for the dataspace is reset to 0.

4.5. Miscellaneous Dataspace Operations

`herr_t H5Slock (hid_t` *`space`*`)`

Locks the dataspace so that it cannot be modified or closed. When the library exits, the dataspace will be unlocked and closed.

`hid_t H5Screate_simple(int` *`rank`*`, const hsize_t *`*`current_size`*`, const hsize_t *`*`maximum_size`*`)`

This function is a "convenience" wrapper to create a simple dataspace and set it's extent in one call. It is equivalent to calling H5Screate and H5Sset_extent_simple() in two steps.

`int H5Sis_subspace(hid_t` *`space`*`)`

This function returns positive if *space* is located within another dataspace, zero if it is not, and negative on a failure.

`char *H5Ssubspace_name(hid_t` *`space`*`)`

This function returns the name of the named dataspace that *space* is located within. If *space* is not located within another dataspace, or an error occurs, NULL is returned. The application is responsible for freeing the string returned.

`herr_t H5Ssubspace_location(hid_t` *`space`*`, hsize_t *`*`loc`*`)`

If *space* is located within another dataspace, this function puts the location of the origin of *space* in the *loc* array. The *loc* array must be at least as large as the number of dimensions of *space*. If *space* is not located within another dataspace or an error occurs, a negative value is returned, otherwise a non-negative value is returned.

Last modified: 14 October 1999

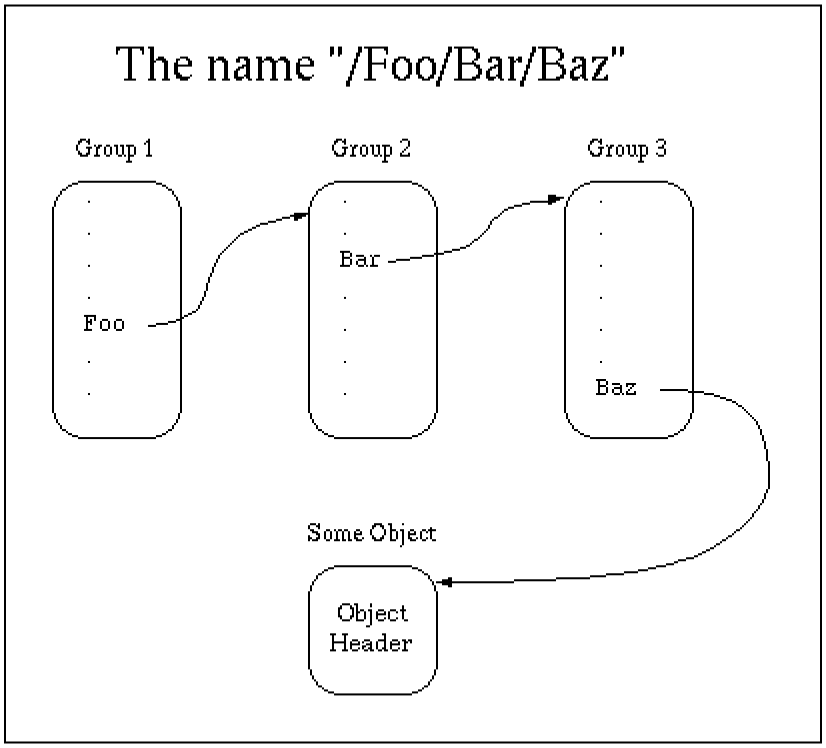
5. The Group Interface (H5G)

5.1. Introduction

An object in HDF5 consists of an object header at a fixed file address that contains messages describing various properties of the object such as its storage location, layout, compression, etc. and some of these messages point to other data such as the raw data of a dataset. The address of the object header is also known as an *OID* and HDF5 has facilities for translating names to OIDs.

Every HDF5 object has at least one name and a set of names can be stored together in a group. Each group implements a name space where the names are any length and unique with respect to other names in the group.

Since a group is a type of HDF5 object it has an object header and a name which exists as a member of some other group. In this way, groups can be linked together to form a directed graph. One particular group is called the *Root Group* and is the group to which the HDF5 file boot block points. Its name is "/" by convention. The *full name* of an object is created by joining component names with slashes much like Unix.



However, unlike Unix which arranges directories hierarchically, HDF5 arranges groups in a directed graph. Therefore, there is no ".." entry in a group since a group can have more than one parent. There is no "." entry either but the library understands it internally.

5.2. Names

HDF5 places few restrictions on names: component names may be any length except zero and may contain any character except slash ("/") and the null terminator. A full name may be composed of any number of component names separated by slashes, with any of the component names being the special name ".". A name which begins with a slash is an *absolute* name which is looked up beginning at the root group of the file while all other *relative* names are looked up beginning at the specified group. Multiple consecutive slashes in a full name are treated as single slashes and trailing slashes are not significant. A special case is the name "/" (or equivalent) which refers to the root group.

Functions which operate on names generally take a location identifier which is either a file ID or a group ID and perform the lookup with respect to that location. Some possibilities are:

Location Type	**Object Name**	**Description**
File ID	`/foo/bar`	The object `bar` in group `foo` in the root group.
Group ID	`/foo/bar`	The object `bar` in group `foo` in the root group of the file containing the specified group. In other words, the group ID's only purpose is to supply a file.
File ID	`/`	The root group of the specified file.
Group ID	`/`	The root group of the file containing the specified group.
File ID	`foo/bar`	The object `bar` in group `foo` in the specified group.
Group ID	`foo/bar`	The object `bar` in group `foo` in the specified group.
File ID	`.`	The root group of the file.
Group ID	`.`	The specified group.
Other ID	`.`	The specified object.

5.3. Creating, Opening, and Closing Groups

Groups are created with the `H5Gcreate()` function, and existing groups can be access with `H5Gopen()`. Both functions return an object ID which should be eventually released by calling `H5Gclose()`.

`hid_t H5Gcreate (hid_t` *`location_id`*`, const char *`*`name`*`, size_t` *`size_hint`*`)`

This function creates a new group with the specified name at the specified location which is either a file ID or a group ID. The name must not already be taken by some other object and all parent groups must already exist. The *size_hint* is a hint for the number of bytes to reserve to store the names which will be eventually added to the new group. Passing a value of zero for *size_hint* is usually adequate since the library is able to dynamically resize the name heap, but a correct hint may result in better performance. The return value is a handle for the open group and it should be closed by calling `H5Gclose()` when it's no longer needed. A negative value is returned for failure.

`hid_t H5Gopen (hid_t` *`location_id`*`, const char *`*`name`*`)`

This function opens an existing group with the specified name at the specified location which is either a file ID or a group ID and returns an object ID. The object ID should be released by calling `H5Gclose()` when it is no longer needed. A negative value is returned for failure.

`herr_t H5Gclose (hid_t` *`group_id`*`)`

This function releases resources used by an group which was opened by `H5Gcreate()` or `H5Gopen()`. After closing a group the *group_id* should not be used again. This function returns zero for success or a negative value for failure.

5.4. Objects with Multiple Names

An object (including a group) can have more than one name. Creating the object gives it the first name, and then functions described here can be used to give it additional names. The association between a name and the object is called a *link* and HDF5 supports two types of links: a *hard link* is a direct association between the name and the object where both exist in a single HDF5 address space, and a *soft link* is an indirect association.

Object Creation

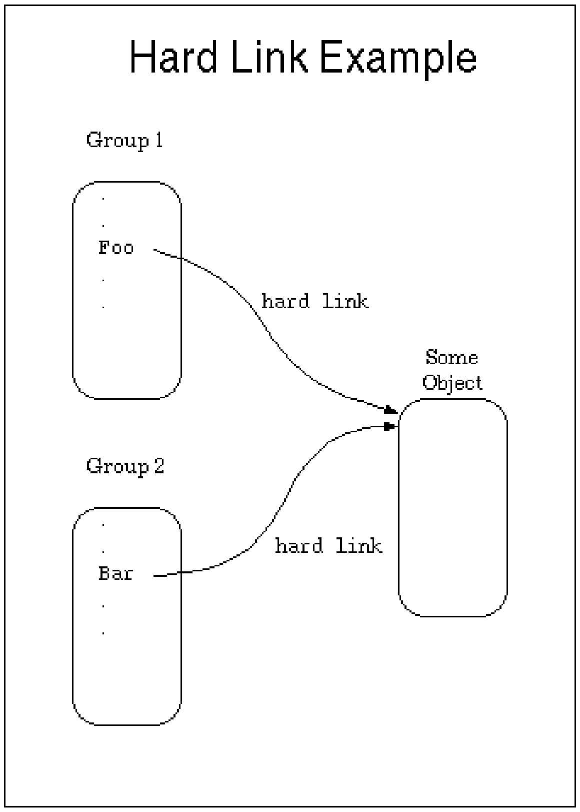
The creation of an object creates a hard link which is indistinguishable from other hard links that might be added later.

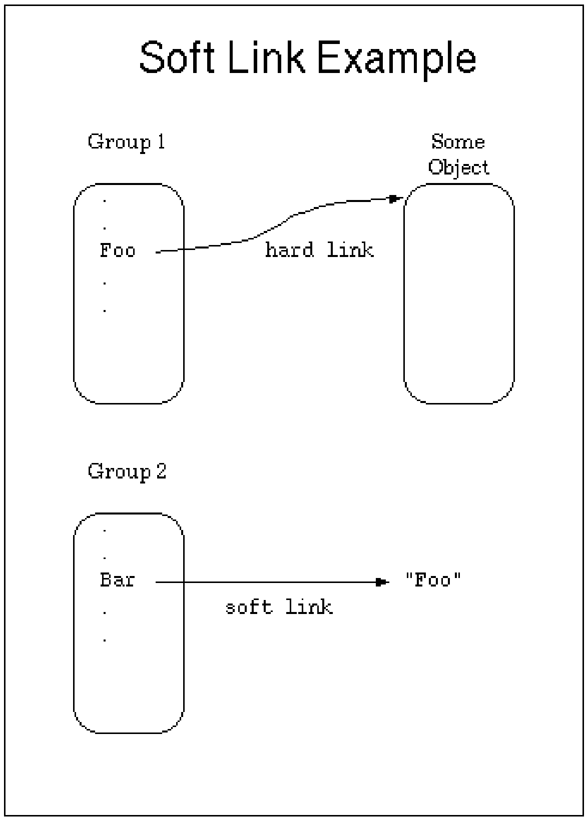
`herr_t H5Glink (hid_t` ***`file_id`***`, H5G_link_t` ***`link_type`***`, const char *`***`current_name`***`, const char *`***`new_name`***`)`

Creates a new name for an object that has some current name (possibly one of many names it currently has). If the *link_type* is `H5G_LINK_HARD` then a new hard link is created. Otherwise if *link_type* is `H5T_LINK_SOFT` a soft link is created which is an alias for the *current_name*. When creating a soft link the object need not exist. This function returns zero for success or negative for failure.

`herr_t H5Gunlink (hid_t` ***`file_id`***`, const char *`***`name`***`)`

This function removes an association between a name and an object. Object headers keep track of how many hard links refer to the object and when the hard link count reaches zero the object can be removed from the file (but objects which are open are not removed until all handles to the object are closed).





5.5. Comments

Objects can have a comment associated with them. The comment is set and queried with these two functions:

`herr_t H5Gset_comment (hid_t` *`loc_id`*`, const char *`*`name`*`, const char *`*`comment`*`)`

> The previous comment (if any) for the specified object is replace with a new comment. If the *comment* argument is the empty string or a null pointer then the comment message is removed from the object. Comments should be relatively short, null-terminated, ASCII strings.

`herr_t H5Gget_comment (hid_t` *`loc_id`*`, const char *`*`name`*`, size_t` *`bufsize`*`, char *`*`comment`*`)`

> The comment string for an object is returned through the *comment* buffer. At most *bufsize* characters including a null terminator are copied, and the result is not null terminated if the comment is longer than the supplied buffer. If an object doesn't have a comment then the empty string is returned.

Last modified: 14 October 1999

6. The Reference Interface (H5R) and the Indentifier Interface (H5I)

6.1. Introduction

This document discusses the kinds of references implemented (and planned) in HDF5 and the functions implemented (and planned) to support them.

6.2. References

This section contains an overview of the kinds of references implemented, or planned for implementation, in HDF5.

Object reference

Reference to an entire object in the current HDF5 file.
The only kind of reference currently implemented.

An object reference points to an entire object in the current HDF5 file by storing the relative file address (OID) of the object header for the object pointed to. The relative file address of an object header is constant for the life of the object. An object reference is of a fixed size in the file.

Dataset region reference

Reference to a specific dataset region.
Not yet implemented.

A dataset region reference points to a region of a dataset in the current HDF5 file by storing the OID of the dataset and the global heap offset of the region referenced. The region referenced is located by retrieving the coordinates of the areas in the region from the global heap. A dataset region reference is of a variable size in the file.

Internal dataset region reference

Reference to a region within the current dataset.
Not yet implemented.

An internal dataset region reference points to a region of the current dataset by storing the coordinates of the region. An internal dataset region reference is of a fixed size in the file.

Note: All references are treated as soft links for the purposes of reference counting. The library does not keep track of reference links and they may dangle if the object they refer to is deleted, moved, or not yet available.

6.3. Reference Types

This section lists valid HDF5 reference types for use in the H5R functions.

Reference Type	Value	Description
`H5R_BADTYPE`	`-1`	Invalid reference type
`H5R_OBJECT`	`0`	Object reference
`H5R_DATASET_REGION`	`1`	Dataset region reference
`H5R_INTERNAL`	`2`	Internal reference

6.4. Functions

Four functions, three in the H5R interface and one in the H5I interface, have been implemented to support references. The H5I function is also useful outside the context of references.

herr_t `H5Rcreate(`*void* `*reference,` *hid_t* `loc_id,` *const char* `*name,` *H5R_type_t* `type,` *hid_t* `space_id)`

> `H5Rcreate` creates an object which is a particular type of reference (specified with the `type` parameter) to some file object and/or location specified with the `space_id` parameter. For dataset region references, the selection specified in the dataspace is the portion of the dataset which will be referred to.

hid_t `H5Rdereference(`*hid_t* `dset,` *H5R_type_t* `rtype,` *void* `*reference)`

> `H5Rdereference` opens the object referenced and returns an identifier for that object. The parameter `reference` specifies a reference of type `rtype` that is stored in the dataset `dset`.

H5S_t `H5Rget_region(`*H5D_t* `dataset,` *H5R_type_t* `type,` *void* `*reference)`

> `H5Rget_region` creates a copy of dataspace of the dataset that is pointed to and defines a selection in the copy which is the location (or region) pointed to. The parameter `ref` specifies a reference of type `rtype` that is stored in the dataset `dset`.

H5I_type_t `H5Iget_type(`*hid_t* `id)`

> Returns the type of object referred to by the identifier `id`. Valid return values appear in the following list:

`H5I_BADID`	Invalid ID
`H5I_FILE`	File objects
`H5I_GROUP`	Group objects
`H5I_DATATYPE`	Datatype objects
`H5I_DATASPACE`	Dataspace objects
`H5I_DATASET`	Dataset objects
`H5I_ATTR`	Attribute objects

> This function was inspired by the need of users to figure out which type of object closing function (`H5Dclose`, `H5Gclose`, etc.) to call after a call to `H5Rdereference`, but it is also of general use.

6.5. Examples

Object Reference Writing Example
Create a dataset which has links to other datasets as part of its raw data and write the dataset to the file.

```
{
    hid_t file1;
    hid_t dataset1;
    hid_t datatype, dataspace;
    char buf[128];
    hobj_ref_t link;
    hobj_ref_t data[10][10];
    int rank;
    size_t dimsf[2];
    int i, j;

    /* Open the file */
    file1=H5Fopen("example.h5", H5F_ACC_RDWR, H5P_DEFAULT);

    /* Describe the size of the array and create the data space */
    rank=2;
    dimsf[0] = 10;
    dimsf[1] = 10;
    dataspace = H5Screate_simple(rank, dimsf, NULL);

    /* Define datatype */
    datatype = H5Tcopy(H5T_STD_REF_OBJ);

    /* Create a dataset */
    dataset1=H5Dcreate(file1,"Dataset One",datatype,dataspace,H5P_DEFAULT);

    /* Construct array of OIDs for other datasets in the file */
    /* somewhat hokey and artificial, but demonstrates the point */
    for(i=0; i<10; i++)
        for(j=0; j<10; i++)
          {
            sprintf(buf,"/Group/Linked Set %d-%d",i,j);
            if(H5Rcreate(&link,file1,buf,H5R_OBJECT,-1)0)
                data[i][j]=link;
          } /* end for */

    /* Write the data to the dataset using default transfer properties.  */
    H5Dwrite(dataset, H5T_STD_REF_OBJ, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    /* Close everything */
    H5Sclose(dataspace);
    H5Tclose(datatype);
    H5Dclose(dataset1);
    H5Fclose(file1);
}
```

Object Reference Reading Example
Open a dataset which has links to other datasets as part of its raw data and read in those links.

```
{
    hid_t file1;
    hid_t dataset1, tmp_dset;
    href_t data[10][10];
    int i, j;

    /* Open the file */
    file1=H5Fopen("example.h5", H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dataset1=H5Dopen(file1,"Dataset One",H5P_DEFAULT);

    /*
     * Read the data to the dataset using default transfer properties.
     * (we are assuming the dataset is the same and not querying the
     *  dimensions, etc.)
     */
    H5Dread(dataset, H5T_STD_REF_OBJ, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    /* Analyze array of OIDs of linked datasets in the file */
    /* somewhat hokey and artificial, but demonstrates the point */
    for(i=0; i<10; i++)
        for(j=0; j<10; i++)
          {
            if((tmp_dset=H5Rdereference(dataset, H5T_STD_REF_OBJ, data[i][j]))0)
              {

              } /* end if */
            H5Dclose(tmp_dset);
          } /* end for */


    /* Close everything */
    H5Dclose(dataset1);
    H5Fclose(file1);
}
```

Dataset Region Reference Writing Example

Create a dataset which has links to other dataset regions (single elements in this case) as part of its raw data and write the dataset to the file.

```
{
    hid_t file1;
    hid_t dataset1, dataset2;
    hid_t datatype, dataspace1, dataspace2;
    char buf[128];
    href_t link;
    href_t data[10][10];     /* HDF5 reference type */
    int rank;
    size_t dimsf[2];
    hssize_t start[3],count[3];
    int i, j;

    /* Open the file */
    file1=H5Fopen("example.h5", H5F_ACC_RDWR, H5P_DEFAULT);

    /* Describe the size of the array and create the data space */
    rank=2;
    dimsf[0] = 10;
    dimsf[1] = 10;
    dataspace1 = H5Screate_simple(rank, dimsf, NULL);

    /* Define Dataset Region Reference datatype */
    datatype = H5Tcopy(H5T_STD_REF_DATAREG);

    /* Create a dataset */
    dataset1=H5Dcreate(file1,"Dataset One",datatype,dataspace1,H5P_DEFAULT);

    /* Construct array of OIDs for other datasets in the file */
    /* (somewhat artificial, but demonstrates the point) */
    for(i=0; i<10; i++)
        for(j=0; j<10; i++)
          {
            sprintf(buf,"/Group/Linked Set %d-%d",i,j);

            /* Get the dataspace for the object to point to */
            dataset2=H5Dopen(file1,buf,H5P_DEFAULT);
            dataspace2=H5Dget_space(dataspace2);

            /* Select the region to point to */
            /* (could be different region for each pointer) */
            start[0]=5; start[1]=4; start[2]=3;
            count[0]=2; count[1]=4; count[2]=1;
            H5Sselect_hyperslab(dataspace2,H5S_SELECT_SET,start,NULL,count,NULL);

            if(H5Rcreate(&link,file1,buf,H5R_REF_DATAREG,dataspace2)0)
                /* Store the reference */
                data[i][j]=link;

            H5Sclose(dataspace2);
            H5Dclose(dataspace2);
          } /* end for */

    /* Write the data to the dataset using default transfer properties.  */
    H5Dwrite(dataset, H5T_STD_REF_DATAREG, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    /* Close everything */
    H5Sclose(dataspace);
    H5Tclose(datatype);
    H5Dclose(dataset1);
    H5Fclose(file1);
}
```

Dataset Region Reference Reading Example

Open a dataset which has links to other datasets regions (single elements in this case) as part of its raw data and read in those links.

```
{
    hid_t file1;
    hid_t dataset1, tmp_dset;
    hid_t dataspace;
    href_t data[10][10];     /* HDF5 reference type */
    int i, j;

    /* Open the file */
    file1=H5Fopen("example.h5", H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open the dataset */
    dataset1=H5Dopen(file1,"Dataset One",H5P_DEFAULT);

    /*
     * Read the data to the dataset using default transfer properties.
     * (we are assuming the dataset is the same and not querying the
     *  dimensions, etc.)
     */
    H5Dread(dataset, H5T_STD_REF_DATAREG, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    /* Analyze array of OIDs of linked datasets in the file */
    /* (somewhat artificial, but demonstrates the point) */
    for(i=0; i<10; i++)
        for(j=0; j<10; i++)
          {
            if((tmp_dset=H5Rdereference(dataset, H5D_STD_REF_DATAREG,data[i][j]))0)
              {
                  /* Get the dataspace with the pointed to region selected */
                  dataspace=H5Rget_space(data[i][j]);



                  H5Sclose(dataspace);
              } /* end if */
            H5Dclose(tmp_dset);
          } /* end for */


    /* Close everything */
    H5Dclose(dataset1);
    H5Fclose(file1);
}
```

Last modified: 14 October 1999

7. The Attribute Interface (H5A)

7.1. Introduction

The attribute API (H5A) is primarily designed to easily allow small datasets to be attached to primary datasets as metadata information. Additional goals for the H5A interface include keeping storage requirements for each attribute to a minimum and easily sharing attributes among datasets.

Because attributes are intended to be small objects, large datasets intended as additional information for a primary dataset should be stored as supplemental datasets in a group with the primary dataset. Attributes can then be attached to the group containing everything to indicate a particular type of dataset with supplemental datasets is located in the group. How small is "small" is not defined by the library and is up to the user's interpretation.

Attributes are not separate objects in the file, they are always contained in the object header of the object they are attached to. The I/O functions defined below are required to read or write attribute information, not the H5D I/O routines.

7.2. Creating, Opening, Closing and Deleting Attributes

Attributes are created with the `H5Acreate()` function, and existing attributes can be accessed with either the `H5Aopen_name()` or `H5Aopen_idx()` functions. All three functions return an object ID which should be eventually released by calling `H5Aclose()`.

`hid_t H5Acreate (hid_t` *`loc_id`*`, const char *`*`name`*`, hid_t` *`type_id`*`, hid_t` *`space_id`*`, hid_t` *`create_plist_id`*`)`

> This function creates an attribute which is attached to the object specified with *loc_id*. The name specified with *name* for each attribute for an object must be unique for that object. The *type_id* and *space_id* are created with the H5T and H5S interfaces respectively. Currently only simple dataspaces are allowed for attribute dataspaces. The *create_plist_id* property list is currently unused, but will be used in the future for optional properties of attributes. The attribute ID returned from this function must be released with H5Aclose or resource leaks will develop. Attempting to create an attribute with the same name as an already existing attribute will fail, leaving the pre-existing attribute in place. This function returns an attribute ID for success or negative for failure.

`hid_t H5Aopen_name (hid_t` *`loc_id`*`, const char *`*`name`*`)`

> This function opens an attribute which is attached to the object specified with *loc_id*. The name specified with *name* indicates the attribute to access. The attribute ID returned from this function must be released with H5Aclose or resource leaks will develop. This function returns an attribute ID for success or negative for failure.

`hid_t H5Aopen_idx (hid_t` *`loc_id`*`, unsigned` *`idx`*`)`

> This function opens an attribute which is attached to the object specified with *loc_id*. The attribute specified with *idx* indicates the *idx*th attribute to access, starting with '0'. The attribute ID returned from this function must be released with H5Aclose or resource leaks will develop. This function returns an attribute ID for success or negative for failure.

`herr_t H5Aclose (hid_t` *`attr_id`*`)`

> This function releases an attribute from use. Further use of the attribute ID will result in undefined behavior. This function returns non-negative on success, negative on failure.

`herr_t H5Adelete (hid_t loc_id, const char *name)`

This function removes the named attribute from a dataset or group. This function should not be used when attribute IDs are open on *loc_id* as it may cause the internal indexes of the attributes to change and future writes to the open attributes to produce incorrect results. Returns non-negative on success, negative on failure.

7.3. Attribute I/O Functions

Attributes may only be written as an entire object, no partial I/O is currently supported.

`herr_t H5Awrite (hid_t attr_id, hid_t mem_type_id, void *buf)`

This function writes an attribute, specified with *attr_id*, with *mem_type_id* specifying the datatype in memory. The entire attribute is written from *buf* to the file. This function returns non-negative on success, negative on failure.

`herr_t H5Aread (hid_t attr_id, hid_t mem_type_id, void *buf)`

This function read an attribute, specified with *attr_id*, with *mem_type_id* specifying the datatype in memory. The entire attribute is read into *buf* from the file. This function returns non-negative on success, negative on failure.

7.4. Attribute Inquiry Functions

```
int H5Aiterate (hid_t loc_id, unsigned *attr_number, H5A_operator operator, void
*operator_data)
```

This function iterates over the attributes of the dataset or group specified with *loc_id*. For each attribute of the object, the *operator_data* and some additional information (specified below) are passed to the *operator* function. The iteration begins with the **attr_number* object in the group and the next attribute to be processed by the operator is returned in **attr_number*.

The iterator returns a negative value if something is wrong, the return value of the last operator if it was non-zero, or zero if all attributes were processed.

The prototype for H5A_operator_t is:
```
typedef herr_t (*H5A_operator_t)(hid_t loc_id, const char *attr_name, void
*operator_data);
```

The operation receives the ID for the group or dataset being iterated over (*loc_id*), the name of the current attribute about the object (*attr_name*) and the pointer to the operator data passed in to H5Aiterate (*operator_data*). The return values from an operator are:

- Zero causes the iterator to continue, returning zero when all attributes have been processed.
- Positive causes the iterator to immediately return that positive value, indicating short-circuit success. The iterator can be restarted at the next attribute.
- Negative causes the iterator to immediately return that value, indicating failure. The iterator can be restarted at the next attribute.

`hid_t H5Aget_space (hid_t attr_id)`

This function retrieves a copy of the dataspace for an attribute. The dataspace ID returned from this function must be released with H5Sclose or resource leaks will develop. This function returns a dataspace ID for success or negative for failure.

`hid_t H5Aget_type (hid_t` *`attr_id`*`)`

This function retrieves a copy of the datatype for an attribute. The datatype ID returned from this function must be released with H5Tclose or resource leaks will develop. This function returns a datatype ID for success or negative for failure.

`ssize_t H5Aget_name (hid_t` *`attr_id`*`, size_t` *`buf_size`*`, char *`*`buf`*`)`

This function retrieves the name of an attribute for an attribute ID. Up to *buf_size* characters are stored in *buf* followed by a '\0' string terminator. If the name of the attribute is longer than *buf_size*-1, the string terminator is stored in the last position of the buffer to properly terminate the string. This function returns the length of the attribute's name (which may be longer than *buf_size*) on success or negative for failure.

`int H5Aget_num_attrs (hid_t` *`loc_id`*`)`

This function returns the number of attributes attached to a dataset or group, *loc_id*. This function returns non-negative for success or negative for failure.

Last modified: 14 October 1999

8. The Property List Interface (H5P)

8.1. Introduction

The property list (a.k.a., template) interface provides a mechanism for default named arguments for a C function interface. A property list is a collection of name/value pairs which can be passed to various other HDF5 functions to control features that are typically unimportant or whose default values are usually used.

For instance, file creation needs to know various things such as the size of the user-block at the beginning of the file, or the size of various file data structures. Wrapping this information in a property list simplifies the API by reducing the number of arguments to `H5Fcreate()`.

8.2. General Property List Operations

Property lists follow the same create/open/close paradigm as the rest of the library.

`hid_t H5Pcreate (H5P_class_t` *`class`*`)`

A new property list can be created as an instance of some property list class. The new property list is initialized with default values for the specified class. The classes are:

`H5P_FILE_CREATE`

Properties for file creation. See the H5F documentation for details about the file creation properties.

`H5P_FILE_ACCESS`

Properties for file access. See the H5F documentation for details about the file creation properties.

`H5P_DATASET_CREATE`

Properties for dataset creation. See the H5D documentation for details about dataset creation properties.

`H5P_DATASET_XFER`

Properties for raw data transfer. See the H5D documentation for details about raw data transfer properties.

`hid_t H5Pcopy (hid_t` *`plist`*`)`

A property list can be copied to create a new property list. The new property list has the same properties and values as the original property list.

`herr_t H5Pclose (hid_t` *`plist`*`)`

All property lists should be closed when the application is finished accessing them. This frees resources used by the property list.

`H5P_class_t H5Pget_class (hid_t` *`plist`*`)`

The class of which the property list is a member can be obtained by calling this function. The property list classes are defined above for `H5Pcreate()`.

Last modified: 14 October 1999

9. The Error Handling Interface (H5E)

9.1. Introduction

When an error occurs deep within the HDF5 library a record is pushed onto an error stack and that function returns a failure indication. Its caller detects the failure, pushes another record onto the stack, and returns a failure indication. This continues until the application-called API function returns a failure indication (a negative integer or null pointer). The next API function which is called (with a few exceptions) resets the stack.

9.2. Error Handling Operations

In normal circumstances, an error causes the stack to be printed on the standard error stream. The first item, number "#000" is produced by the API function itself and is usually sufficient to indicate to the application programmer what went wrong.

Example: An Error Message

If an application calls `H5Tclose` on a predefined datatype then the following message is printed on the standard error stream. This is a simple error that has only one component, the API function; other errors may have many components.

```
HDF5-DIAG: Error detected in thread 0.  Back trace follows.
  #000: H5T.c line 462 in H5Tclose(): predefined datatype
    major(01): Function argument
    minor(05): Bad value
```

The error stack can also be printed and manipulated by these functions, but if an application wishes make explicit calls to `H5Eprint()` then the automatic printing should be turned off to prevent error messages from being displayed twice (see `H5Eset_auto()` below).

`herr_t H5Eprint (FILE *`*`stream`*`)`

The error stack is printed on the specified stream. Even if the error stack is empty a one-line message will be printed: `HDF5-DIAG: Error detected in thread 0.`

`herr_t H5Eclear (void)`

The error stack can be explicitly cleared by calling this function. The stack is also cleared whenever an API function is called, with certain exceptions (for instance, `H5Eprint()`).

Sometimes an application will call a function for the sake of its return value, fully expecting the function to fail. Under these conditions, it would be misleading if an error message were automatically printed. Automatic printing of messages is controlled by the `H5Eset_auto()` function:

`herr_t H5Eset_auto (herr_t(*`*`func`*`)(void*), void *`*`client_data`*`)`

If *func* is not a null pointer, then the function to which it points will be called automatically when an API function is about to return an indication of failure. The function is called with a single argument, the *client_data* pointer. When the library is first initialized the auto printing function is set to `H5Eprint()` (cast appropriately) and *client_data* is the standard error stream pointer, `stderr`.

`herr_t H5Eget_auto (herr_t(**`*`func`*`)(void*), void **`*`client_data`*`)`

This function returns the current automatic error traversal settings through the *func* and *client_data* arguments. Either (or both) arguments may be null pointers in which case the corresponding information is not returned.

Example: Error Control

An application can temporarily turn off error messages while "probing" a function.

```
/* Save old error handler */
herr_t (*old_func)(void*);
void *old_client_data;
H5Eget_auto(&old_func, &old_client_data);

/* Turn off error handling */
H5Eset_auto(NULL, NULL);

/* Probe. Likely to fail, but that's okay */
status = H5Fopen (......);

/* Restore previous error handler */
H5Eset_auto(old_func, old_client_data);
```

Or automatic printing can be disabled altogether and error messages can be explicitly printed.

```
/* Turn off error handling permanently */
H5Eset_auto (NULL, NULL);

/* If failure, print error message */
if (H5Fopen (....)<0) {
    H5Eprint (stderr);
    exit (1);
}
```

The application is allowed to define an automatic error traversal function other than the default `H5Eprint()`. For instance, one could define a function that prints a simple, one-line error message to the standard error stream and then exits.

Example: Simple Messages

The application defines a function to print a simple error message to the standard error stream.

```
herr_t
my_hdf5_error_handler (void *unused)
{
   fprintf (stderr, "An HDF5 error was detected. Bye.\n");
   exit (1);
}
```

The function is installed as the error handler by saying

```
H5Eset_auto (my_hdf5_error_handler, NULL);
```

The `H5Eprint()` function is actually just a wrapper around the more complex `H5Ewalk()` function which traverses an error stack and calls a user-defined function for each member of the stack.

`herr_t H5Ewalk (H5E_direction_t` *`direction`*`, H5E_walk_t` *`func`*`, void *`*`client_data`*`)`

The error stack is traversed and *func* is called for each member of the stack. Its arguments are an integer sequence number beginning at zero (regardless of *direction*), a pointer to an error description record, and the *client_data* pointer. If *direction* is `H5E_WALK_UPWARD` then traversal begins at the inner-most function that detected the error and concludes with the API function. The opposite order is `H5E_WALK_DOWNWARD`.

`typedef herr_t (*H5E_walk_t)(int` *`n`*`, H5E_error_t *`*`eptr`*`, void *`*`client_data`*`)`

An error stack traversal callback function takes three arguments: *n* is a sequence number beginning at zero for each traversal, *eptr* is a pointer to an error stack member, and *client_data* is the same pointer passed to `H5Ewalk()`.

```
typedef struct {
    H5E_major_t maj_num;
    H5E_minor_t min_num;
    const char  *func_name;
    const char  *file_name;
    unsigned    line;
    const char  *desc;
} H5E_error_t;
```

The *maj_num* and *min_num* are major and minor error numbers, *func_name* is the name of the function where the error was detected, *file_name* and *line* locate the error within the HDF5 library source code, and *desc* points to a description of the error.

`const char *H5Eget_major (H5E_major_t` *`num`*`)`

`const char *H5Eget_minor (H5E_minor_t` *`num`*`)`

These functions take a major or minor error number and return a constant string which describes the error. If *num* is out of range than a string like "Invalid major error number" is returned.

Example: H5Ewalk_cb

This is the implementation of the default error stack traversal callback.

```
herr_t
H5Ewalk_cb(int n, H5E_error_t *err_desc, void *client_data)
{
    FILE          *stream = (FILE *)client_data;
    const char             *maj_str = NULL;
    const char             *min_str = NULL;
    const int              indent = 2;

    /* Check arguments */
    assert (err_desc);
    if (!client_data) client_data = stderr;

    /* Get descriptions for the major and minor error numbers */
    maj_str = H5Eget_major (err_desc->maj_num);
    min_str = H5Eget_minor (err_desc->min_num);

    /* Print error message */
    fprintf (stream, "%*s#%03d: %s line %u in %s(): %s\n",
              indent, "", n, err_desc->file_name, err_desc->line,
              err_desc->func_name, err_desc->desc);
    fprintf (stream, "%*smajor(%02d): %s\n",
              indent*2, "", err_desc->maj_num, maj_str);
    fprintf (stream, "%*sminor(%02d): %s\n",
              indent*2, "", err_desc->min_num, min_str);

    return 0;
}
```

Last modified: 14 October 1999

10. Filters in HDF5

Note: Transient pipelines described in this document have not been implemented.

10.1. Introduction

HDF5 allows chunked data to pass through user-defined filters on the way to or from disk. The filters operate on chunks of an `H5D_CHUNKED` dataset can be arranged in a pipeline so output of one filter becomes the input of the next filter.

Each filter has a two-byte identification number (type `H5Z_filter_t`) allocated by NCSA and can also be passed application-defined integer resources to control its behavior. Each filter also has an optional ASCII comment string.

Values for `H5Z_filter_t`

Value	Description
`0-255`	These values are reserved for filters predefined and registered by the HDF5 library and of use to the general public. They are described in a separate section below.
`256-511`	Filter numbers in this range are used for testing only and can be used temporarily by any organization. No attempt is made to resolve numbering conflicts since all definitions are by nature temporary.
`512-65535`	Reserved for future assignment. Please contact the HDF5 development team (`hdf5dev@ncsa.uiuc.edu`) to reserve a value or range of values for use by your filters.

10.2. Defining and Querying the Filter Pipeline

Two types of filters can be applied to raw data I/O: permanent filters and transient filters. The permanent filter pipeline is defned when the dataset is created while the transient pipeline is defined for each I/O operation. During an `H5Dwrite()` the transient filters are applied first in the order defined and then the permanent filters are applied in the order defined. For an `H5Dread()` the opposite order is used: permanent filters in reverse order, then transient filters in reverse order. An `H5Dread()` must result in the same amount of data for a chunk as the original `H5Dwrite()`.

The permanent filter pipeline is defined by calling `H5Pset_filter()` for a dataset creation property list while the transient filter pipeline is defined by calling that function for a dataset transfer property list.

`herr_t H5Pset_filter (hid_t` *`plist`*`, H5Z_filter_t` *`filter`*`, unsigned int` *`flags`*`, size_t` *`cd_nelmts`*`, const unsigned int` *`cd_values`*`[])`

> This function adds the specified *filter* and corresponding properties to the end of the transient or permanent output filter pipeline (depending on whether *plist* is a dataset creation or dataset transfer property list). The *flags* argument specifies certain general properties of the filter and is documented below. The *cd_values* is an array of *cd_nelmts* integers which are auxiliary data for the filter. The integer values will be stored in the dataset object header as part of the filter information.

`int H5Pget_nfilters (hid_t` *`plist`*`)`

> This function returns the number of filters defined in the permanent or transient filter pipeline depending on whether *plist* is a dataset creation or dataset transfer property list. In each pipeline the filters are numbered from 0 through *N*-1 where *N* is the value returned by this function. During output to the file the filters of a pipeline are applied in increasing order (the inverse is true for input). Zero is returned if there are no filters in the pipeline and a negative value is returned for errors.

`H5Z_filter_t H5Pget_filter (hid_t` *`plist`*`, int` *`filter_number`*`, unsigned int *`*`flags`*`, size_t *`*`cd_nelmts`*`, unsigned int *`*`cd_values`*`, size_t namelen, char name[])`

> This is the query counterpart of `H5Pset_filter()` and returns information about a particular filter number in a permanent or transient pipeline depending on whether *plist* is a dataset creation or dataset transfer property list. On input, *cd_nelmts* indicates the number of entries in the *cd_values* array allocated by the caller while on exit it contains the number of values defined by the filter. The *filter_number* should be a value between zero and *N*-1 as described for `H5Pget_nfilters()` and the function will return failure (a negative value) if the filter number is out of range. If *name* is a pointer to an array of at least *namelen* bytes then the filter name will be copied into that array. The name will be null terminated if the *namelen* is large enough. The filter name returned will be the name appearing in the file or else the name registered for the filter or else an empty string.

The flags argument to the functions above is a bit vector of the following fields:

Values for the *flags* argument

Value	Description
`H5Z_FLAG_OPTIONAL`	If this bit is set then the filter is optional. If the filter fails (see below) during an `H5Dwrite()` operation then the filter is just excluded from the pipeline for the chunk for which it failed; the filter will not participate in the pipeline during an `H5Dread()` of the chunk. This is commonly used for compression filters: if the compression result would be larger than the input then the compression filter returns failure and the uncompressed data is stored in the file. If this bit is clear and a filter fails then the `H5Dwrite()` or `H5Dread()` also fails.

10.3. Defining Filters

Each filter is bidirectional, handling both input and output to the file, and a flag is passed to the filter to indicate the direction. In either case the filter reads a chunk of data from a buffer, usually performs some sort of transformation on the data, places the result in the same or new buffer, and returns the buffer pointer and size to the caller. If something goes wrong the filter should return zero to indicate a failure.

During output, a filter that fails or isn’t defined and is marked as optional is silently excluded from the pipeline and will not be used when reading that chunk of data. A required filter that fails or isn’t defined causes the entire output operation to fail. During input, any filter that has not been excluded from the pipeline during output and fails or is not defined will cause the entire input operation to fail.

Filters are defined in two phases. The first phase is to define a function to act as the filter and link the function into the application. The second phase is to register the function, associating the function with an `H5Z_filter_t` identification number and a comment.

`typedef size_t (*H5Z_func_t)(unsigned int` *`flags`*`, size_t` *`cd_nelmts`*`, const unsigned int` *`cd_values`*`[], size_t` *`nbytes`*`, size_t *`*`buf_size`*`, void **`*`buf`*`)`

> The *flags*, *cd_nelmts*, and *cd_values* are the same as for the `H5Pset_filter()` function with the additional flag `H5Z_FLAG_REVERSE` which is set when the filter is called as part of the input pipeline. The input buffer is pointed to by **buf* and has a total size of **buf_size* bytes but only *nbytes* are valid data. The filter should perform the transformation in place if possible and return the number of valid bytes or zero for failure. If the transformation cannot be done in place then the filter should allocate a new buffer with `malloc()` and assign it to **buf*, assigning the allocated size of that buffer to **buf_size*. The old buffer should be freed by calling `free()`.

`herr_t H5Zregister (H5Z_filter_t` *`filter_id`*`, const char *`*`comment`*`, H5Z_func_t` *`filter`*`)`

> The *filter* function is associated with a filter number and a short ASCII comment which will be stored in the hdf5 file if the filter is used as part of a permanent pipeline during dataset creation.

10.4. Predefined Filters

If `zlib` version 1.1.2 or later was found during configuration then the library will define a filter whose `H5Z_filter_t` number is `H5Z_FILTER_DEFLATE`. Since this compression method has the potential for generating compressed data which is larger than the original, the `H5Z_FLAG_OPTIONAL` flag should be turned on so such cases can be handled gracefully by storing the original data instead of the compressed data. The *cd_nvalues* should be one with *cd_value[0]* being a compression agression level between zero and nine, inclusive (zero is the fastest compression while nine results in the best compression ratio).

A convenience function for adding the `H5Z_FILTER_DEFLATE` filter to a pipeline is:

`herr_t H5Pset_deflate (hid_t` *`plist`*`, unsigned` *`aggression`*`)`

> The deflate compression method is added to the end of the permanent or transient filter pipeline depending on whether *plist* is a dataset creation or dataset transfer property list. The *aggression* is a number between zero and nine (inclusive) to indicate the tradeoff between speed and compression ratio (zero is fastest, nine is best ratio).

Even if the `zlib` isn't detected during configuration the application can define `H5Z_FILTER_DEFLATE` as a permanent filter. If the filter is marked as optional (as with `H5Pset_deflate()`) then it will always fail and be automatically removed from the pipeline. Applications that read data will fail only if the data is actually compressed; they won't fail if `H5Z_FILTER_DEFLATE` was part of the permanent output pipeline but was automatically excluded because it didn't exist when the data was written.

10.5. Example

This example shows how to define and register a simple filter that adds a checksum capability to the data stream.

The function that acts as the filter always returns zero (failure) if the `md5()` function was not detected at configuration time (left as an excercise for the reader). Otherwise the function is broken down to an input and output half. The output half calculates a checksum, increases the size of the output buffer if necessary, and appends the checksum to the end of the buffer. The input half calculates the checksum on the first part of the buffer and compares it to the checksum already stored at the end of the buffer. If the two differ then zero (failure) is returned, otherwise the buffer size is reduced to exclude the checksum.

```
size_t
md5_filter(unsigned int flags, size_t cd_nelmts,
           const unsigned int cd_values[], size_t nbytes,
           size_t *buf_size, void **buf)
{
#ifdef HAVE_MD5
    unsigned char       cksum[16];

    if (flags & H5Z_REVERSE) {
        /* Input */
        assert(nbytes=16);
        md5(nbytes-16, *buf, cksum);

        /* Compare */
        if (memcmp(cksum, (char*)(*buf)+nbytes-16, 16)) {
            return 0; /*fail*/
        }

        /* Strip off checksum */
        return nbytes-16;

    } else {
        /* Output */
        md5(nbytes, *buf, cksum);

        /* Increase buffer size if necessary */
        if (nbytes+16*buf_size) {
            *buf_size = nbytes + 16;
            *buf = realloc(*buf, *buf_size);
        }

        /* Append checksum */
        memcpy((char*)(*buf)+nbytes, cksum, 16);
        return nbytes+16;
    }
#else
    return 0; /*fail*/
#endif
}
```

Once the filter function is defined it must be registered so the HDF5 library knows about it. Since we’re testing this filter we choose one of the `H5Z_filter_t` numbers from the reserved range. We’ll randomly choose 305.

```
#define FILTER_MD5 305
herr_t status = H5Zregister(FILTER_MD5, "md5 checksum", md5_filter);
```

Now we can use the filter in a pipeline. We could have added the filter to the pipeline before defining or registering the filter as long as the filter was defined and registered by time we tried to use it (if the filter is marked as optional then we could have used it without defining it and the library would have automatically removed it from the pipeline for each chunk written before the filter was defined and registered).

```
hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
hsize_t chunk_size[3] = {10,10,10};
H5Pset_chunk(dcpl, 3, chunk_size);
H5Pset_filter(dcpl, FILTER_MD5, 0, 0, NULL);
hid_t dset = H5Dcreate(file, "dset", H5T_NATIVE_DOUBLE, space, dcpl);
```

10.6. Filter Diagnostics

If the library is compiled with debugging turned on for the H5Z layer (usually as a result of `configure --enable-debug=z`) then filter statistics are printed when the application exits normally or the library is closed. The statistics are written to the standard error stream and include two lines for each filter that was used: one for input and one for output. The following fields are displayed:

Field Name	Description
Method	This is the name of the method as defined with `H5Zregister()` with the charaters "< or ">" prepended to indicate input or output.
Total	The total number of bytes processed by the filter including errors. This is the maximum of the *nbytes* argument or the return value.
Errors	This field shows the number of bytes of the Total column which can be attributed to errors.
User, System, Elapsed	These are the amount of user time, system time, and elapsed time in seconds spent in the filter function. Elapsed time is sensitive to system load. These times may be zero on operating systems that don't support the required operations.
Bandwidth	This is the filter bandwidth which is the total number of bytes processed divided by elapsed time. Since elapsed time is subject to system load the bandwidth numbers cannot always be trusted. Furthermore, the bandwidth includes bytes attributed to errors which may significanly taint the value if the function is able to detect errors without much expense.

Example: Filter Statistics

```
H5Z: filter statistics accumulated over life of library:
   Method     Total  Errors  User  System  Elapsed Bandwidth
   ------     -----  ------  ----  ------  ------- ---------
   deflate  160000   40000  0.62    0.74     1.33 117.5 kBs
   <deflate  120000       0  0.11    0.00     0.12 1.000 MBs
```

Last modified: 14 October 1999

11. HDF5 Palette Specification

This section is a work in progress. Please send comments to the HDF5 development team (`hdf5dev@ncsa.uiuc.edu`). Anything and everything on this page may be changed.

Questions regarding this specification:

- Currently, the range index is referred to as an attribute of the palette. I'm wondering if it makes more sense for this to be an attribute of the dataset... If a palette is to be shared by multiple data sets and each dataset has it's own range index mapping, then maybe one would want the index range to be an attribute of the palette.
- Should the range index be separate from the palette, or it be incorporated into a 4 column array?
- Is this method of specifying attributes satisfactory (string name identifier with corresponding value), or should they be specified as a struct that is read in and written out? The number of attributes is currently at 5.

Changes made since last revision

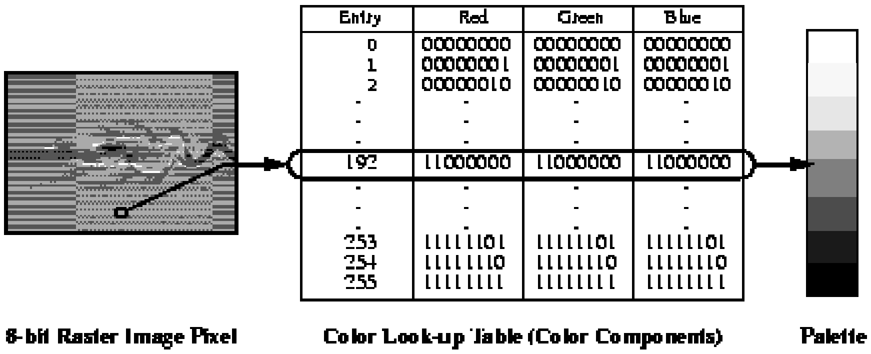
- What was previously called the FLOATRANGE index array has been changed to the RANGEINDEX array, and can be of any type, *preferably* matching that of the data set type.
- A range value min/max attribute for the color numeric may be specified. i.e. the red component value of an RGB will be between 0 and 1.
- A dataset may specify an array of palettes that it may be used with now. previously it was just one.
- CMYK and YCbCr color models have been added.

11.1. HDF 5.0 Palette Overview:

HDF 5.0 adds the following new features to what existed in earlier HDF versions:

- palettes of varying length.
- definable arbitrary index range
- definable color model type (RGB, YUV, HSV, CMY, etc.)
- definable color numeric type

A palette is the means by which color is applied to an image and is also referred to as a color lookup table. It is a table in which every row contains the numerical representation of a particular color. In the example of an 8-bit standard RGB color model palette, this numerical representation of a color is presented as a triplet specifying the intensity of red, green, and blue components that make up each color.



Entry	Red	Green	Blue
0	00000000	00000000	00000000
1	00000001	00000001	00000001
2	00000010	00000010	00000010
-	-	-	-
-	-	-	-
-	-	-	-
192	11000000	11000000	11000000
-	-	-	-
-	-	-	-
-	-	-	-
253	11111101	11111101	11111101
254	11111110	11111110	11111110
255	11111111	11111111	11111111

In this example, the color component numeric type is an 8-bit unsigned integer. While this is most common and recommended for general use, other component color numeric datatypes, such as a 16-bit unsigned integer , may be used. This type is specified as the type attribute of the palette dataset. (see H5Tget_type(), H5Tset_type())

The minimum and maximum values of the component color numeric are specified as attribute of the palette dataset. See below (attributes PAL_MINNUMERIC, PAL_MAXNUMERIC). If these attributes do not exist, it is assumed that the range of values will fill the space of the color numeric type. i.e. with an 8-bit unsigned integer, the valid range would be 0 to 255 for each color component.

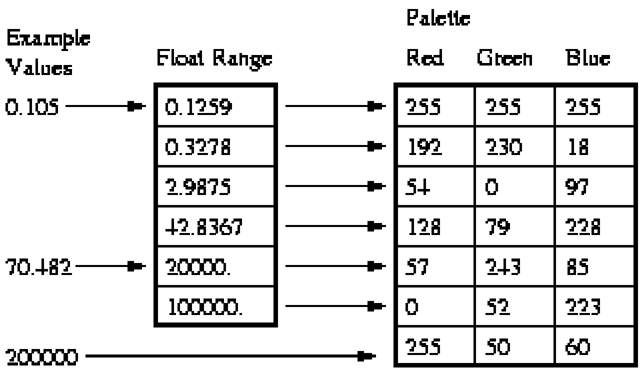
The HDF 5.0 palette specification additionally allows for color models beyond RGB. YUV, HSV, CMY, CMYK, YCbCr color models are supported, and may be specified as a color model attribute of the palette dataset. *(see "Palette Attributes" for details).*

In HDF 4 and earlier, palettes were limited to 256 colors. The HDF 5.0 palette specification allows for palettes of varying length. The length is specified as the number of rows of the palette dataset.

In a standard palette, the color entries are indexed directly. HDF 5.0 supports the notion of a range index table. Such a table defines an ascending ordered of ranges that map dataset values to the palette. If a range index table exists for the palette, the PAL_TYPE attribute will be set to "RANGEINDEX", and the PAL_RANGEINDEX attribute will contain an object reference to a range index table array. If not, the PAL_TYPE attribute either does not exist, or will be set to "STANDARD".

The range index table array consists of a one dimensional array with the same length as the palette dataset - 1. Ideally, the range index would be of the same type as the dataset it refers to, however this is not a requirement.

Example 2: A range index array of type floating point



Example Values	Float Range	Red	Green	Blue
0.105	0.1259	255	255	255
	0.3278	192	230	18
	2.9875	54	0	97
	42.8367	128	79	228
70.482	20000.	57	243	85
	100000.	0	52	223
200000		255	50	60

The range index array attribute defines the "*to*" of the range. Notice that the range index array attribute is one less entry in size than the palette. The first entry of 0.1259, specifies that all values below and up to 0.1259 inclusive, will map to the first palette entry. The second entry signifies that all values greater than 0.1259 up to 0.3278 inclusive, will map to the second palette entry, etc. All value greater than the last range index array attribute (100000) map to the last entry in the palette.

11.2. Palette Attributes

A palette exists in an HDF file as an independent data set with accompanying attributes.

These attributes are defined as follows:

Attribute name="**CLASS**"

> This attribute is of type H5T_STR_NULLTERM.
>
> For all palettes, the value of this attribute is "PALETTE". This attribute identifies this palette data set as a palette that conforms to the specifications on this page.

Attribute name="**PAL_COLORMODEL**"

> This attribute is of type H5T_STR_NULLTERM.
>
> Possible values for this are "RGB", "YUV", "CMY", "HSV".
>
> This defines the color model that the entries in the palette data set represent.

"RGB"

> Each color index contains a triplet where the the first value defines the red component, second defines the green component, and the third the blue component.

"CMY"

> Each color index contains a triplet where the the first value defines the cyan component, second defines the magenta component, and the third the yellow component.

"CMYK"

> Each color index contains a quadruplet where the the first value defines the cyan component, second defines the magenta component, the third the yellow component, and the forth the black component.

"YCbCr"

> Class Y encoding model. Each color index contains a triplet where the the first value defines the luminance, second defines the Cb Chromonance, and the third the Cr Chromonance.

"YUV"

> Composite encoding color model. Each color index contains a triplet where the the first value defines the luminance component, second defines the chromonance component, and the third the value component.

"HSV"

> Each color index contains a triplet where the the first value defines the hue component, second defines the saturation component, and the third the value component. The hue component defines the hue spectrum with a low value representing magenta/red progressing to a high value which would represent blue/magenta, passing through yellow, green, cyan. A low value for the saturation component means less color saturation than a high value. A low value for *value* will be darker than a high value.

Attribute name="**PAL_TYPE**"

This attribute is of type H5T_STR_NULLTERM
The current supported values for this attribute are : "STANDARD8" or "RANGEINDEX"

A PAL_TYPE of "STANDARD8" defines a palette dataset such that the first entry defines index 0, the second entry defines index 1, etc. up until the length of the palette - 1. This assumes an image dataset with direct indexes into the palette.

If the PAL_TYPE is set to "RANGEINDEX", there will be an additional attribute with a name of "**PAL_RANGEINDEX**", The **PAL_RANGEINDEX** attribute contains an HDF object reference pointer which specifies a range index array in the file to be used for color lookups for the palette. (See example 2 for more details)

Attribute name="**PAL_MINNUMERIC**"

Attribute name="**PAL_MAXNUMERIC**"

These two attributes are of the same type as the palette elements or color numerics.
They specify the minimum and maximum values of the color numeric components. For example, if the palette was an RGB of type Float, the color numeric range for Red, Green, and Blue could be set to be between 0.0 and 1.0. The intensity of the color guns would then be scaled accordingly to be between this minimum and maximum attribute.

11.3. Specifying a Palette for a Dataset

A dataset within an HDF5 file may specify an array of palettes to be viewed with. The dataset will have an attribute field called "**PALETTE**" which contains an array of object reference pointers which refer to palettes in the file. The first palette in this array will be the default palette that the data may be viewed with.

Last modified: 14 October 1999

12. Data Caching

12.1. Meta Data Caching

The HDF5 library caches two types of data: meta data and raw data. The meta data cache holds file objects like the file header, symbol table nodes, global heap collections, object headers and their messages, etc. in a partially decoded state. The cache has a fixed number of entries which is set with the file access property list (defaults to 10k) and each entry can hold a single meta data object. Collisions between objects are handled by preempting the older object in favor of the new one.

12.2. Raw Data Chunk Caching

Raw data chunks are cached because I/O requests at the application level typically don't map well to chunks at the storage level. The chunk cache has a maximum size in bytes set with the file access property list (defaults to 1MB) and when the limit is reached chunks are preempted based on the following set of heuristics.

- Chunks which have not been accessed for a long time relative to other chunks are penalized.
- Chunks which have been accessed frequently in the recent past are favored.
- Chunks which are completely read and not written, completely written but not read, or completely read and completely written are penalized according to *w0*, an application-defined weight between 0 and 1 inclusive. A weight of zero does not penalize such chunks while a weight of 1 penalizes those chunks more than all other chunks. The default is 0.75.
- Chunks which are larger than the maximum cache size do not participate in the cache.

One should choose large values for *w0* if I/O requests typically do not overlap but smaller values for *w0* if the requests do overlap. For instance, reading an entire 2d array by reading from non-overlapping "windows" in a row-major order would benefit from a high *w0* value while reading a diagonal accross the dataset where each request overlaps the previous request would benefit from a small *w0*.

12.3. Data Caching Operations

The cache parameters for both caches are part of a file access property list and are set and queried with this pair of functions:

`herr_t H5Pset_cache(hid_t` *`plist`*`, unsigned int` *`mdc_nelmts`*`, size_t` *`rdcc_nbytes`*`, double` *`w0`*`)`

`herr_t H5Pget_cache(hid_t` *`plist`*`, unsigned int *`*`mdc_nelmts`*`, size_t *`*`rdcc_nbytes`*`, double` *`w0`*`)`

Sets or queries the meta data cache and raw data chunk cache parameters. The *plist* is a file access property list. The number of elements (objects) in the meta data cache is *mdc_nelmts*. The total size of the raw data chunk cache and the preemption policy is *rdcc_nbytes* and *w0*. For `H5Pget_cache()` any (or all) of the pointer arguments may be null pointers.

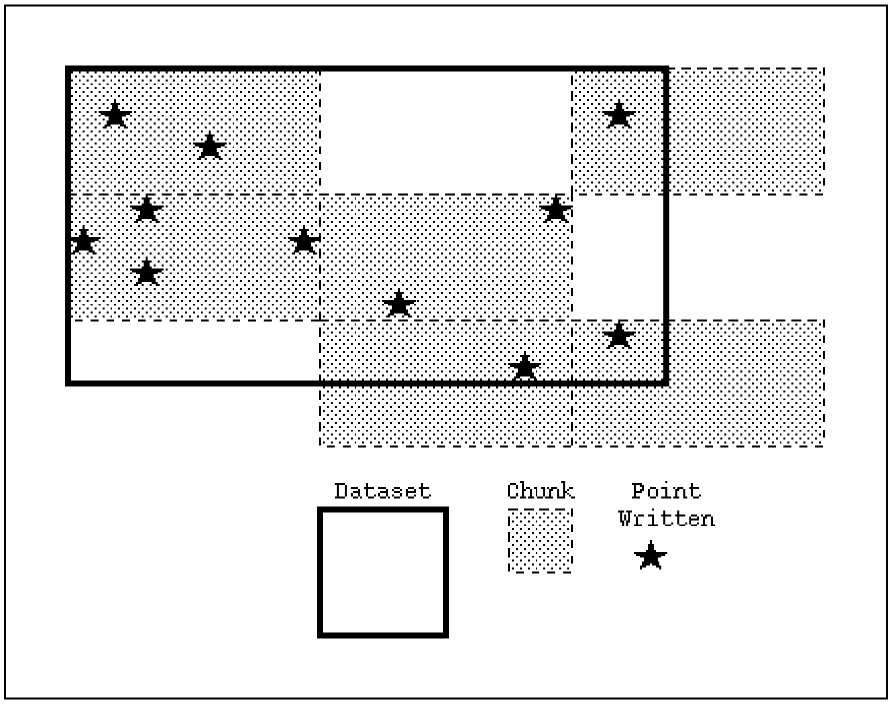
Last modified: 14 October 1999

13. Dataset Chunking Issues

13.1. Introduction

Chunking refers to a storage layout where a dataset is partitioned into fixed-size multi-dimensional chunks. The chunks cover the dataset but the dataset need not be an integral number of chunks. If no data is ever written to a chunk then that chunk isn't allocated on disk. Figure 1 shows a 25x48 element dataset covered by nine 10x20 chunks and 11 data points written to the dataset. No data was written to the region of the dataset covered by three of the chunks so those chunks were never allocated in the file -- the other chunks are allocated at independent locations in the file and written in their entirety.

Figure 1



The HDF5 library treats chunks as atomic objects -- disk I/O is always in terms of complete chunks (footnote 1). This allows data filters to be defined by the application to perform tasks such as compression, encryption, checksumming, *etc*. on entire chunks. As shown in Figure 2 (next page), if `H5Dwrite()` touches only a few bytes of the chunk, the entire chunk is read from the file, the data passes upward through the filter pipeline, the few bytes are modified, the data passes downward through the filter pipeline, and the entire chunk is written back to the file.

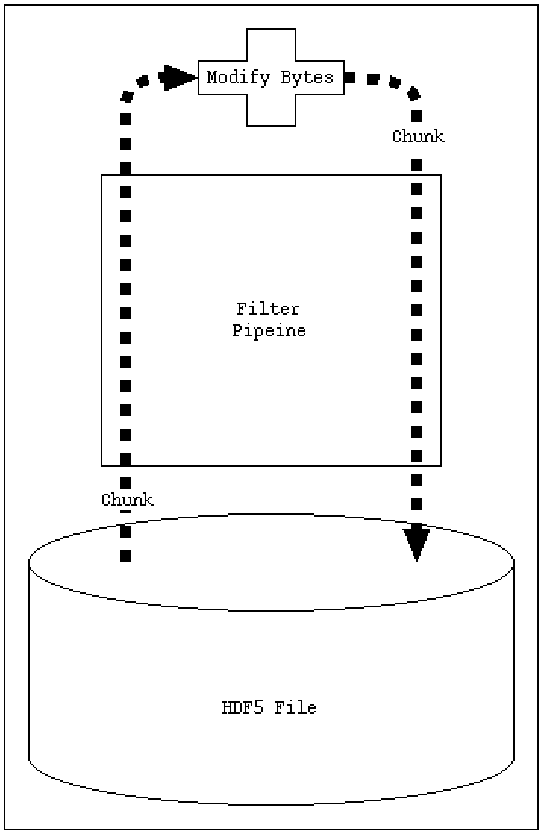


Figure 2

13.2. The Raw Data Chunk Cache

It’s obvious from Figure 2 that calling `H5Dwrite()` many times from the application would result in poor performance even if the data being written all falls within a single chunk. A raw data chunk cache layer was added between the top of the filter stack and the bottom of the byte modification layer (footnote 2). By default, the chunk cache will store 521 chunks or 1MB of data (whichever is less) but these values can be modified with `H5Pset_cache()`.

The preemption policy for the cache favors certain chunks and tries not to preempt them.

- Chunks that have been accessed frequently in the near past are favored.
- A chunk which has just entered the cache is favored.
- A chunk which has been completely read or completely written but not partially read or written is penalized according to some application specified weighting between zero and one.
- A chunk which is larger than the maximum cache size is not eligible for caching.

13.3. Cache Efficiency

Now for some real numbers... A 2000x2000 element dataset is created and covered by a 20x20 array of chunks (each chunk is 100x100 elements). The raw data cache is adjusted to hold at most 25 chunks by setting the maximum number of bytes to 25 times the chunk size in bytes. Then the application creates a square, two-dimensional memory buffer and uses it as a window into the dataset, first reading and then rewriting in row-major order by moving the window across the dataset (the read and write tests both start with a cold cache).

The measure of efficiency in Figure 3 is the number of bytes requested by the application divided by the number of bytes transferred from the file. There are at least a couple ways to get an estimate of the cache performance: one way is to turn on cache debugging (see *Debugging* in this *User's Guide*) and look at the number of cache misses. A more accurate and specific way is to register a data filter whose sole purpose is to count the number of bytes that pass through it (that's the method used below).

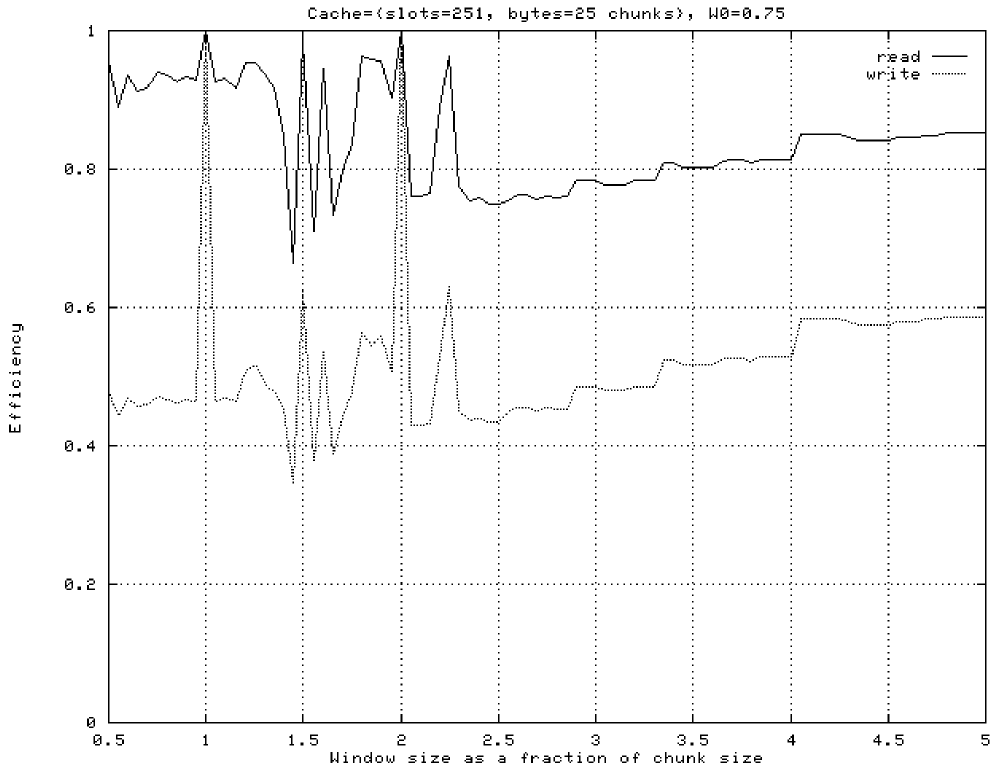


Figure 3

The read efficiency is less than one for two reasons: collisions in the cache are handled by preempting one of the colliding chunks, and the preemption algorithm occasionally preempts a chunk which hasn't been referenced for a long time but is about to be referenced in the near future.

The write test results in lower efficiency for most window sizes because HDF5 is unaware that the application is about to overwrite the entire dataset and must read in most chunks before modifying parts of them.

There is a simple way to improve efficiency for this example. It turns out that any chunk that has been completely read or written is a good candidate for preemption. If we increase the penalty for such chunks from the default 0.75 to the maximum 1.00 then efficiency improves.

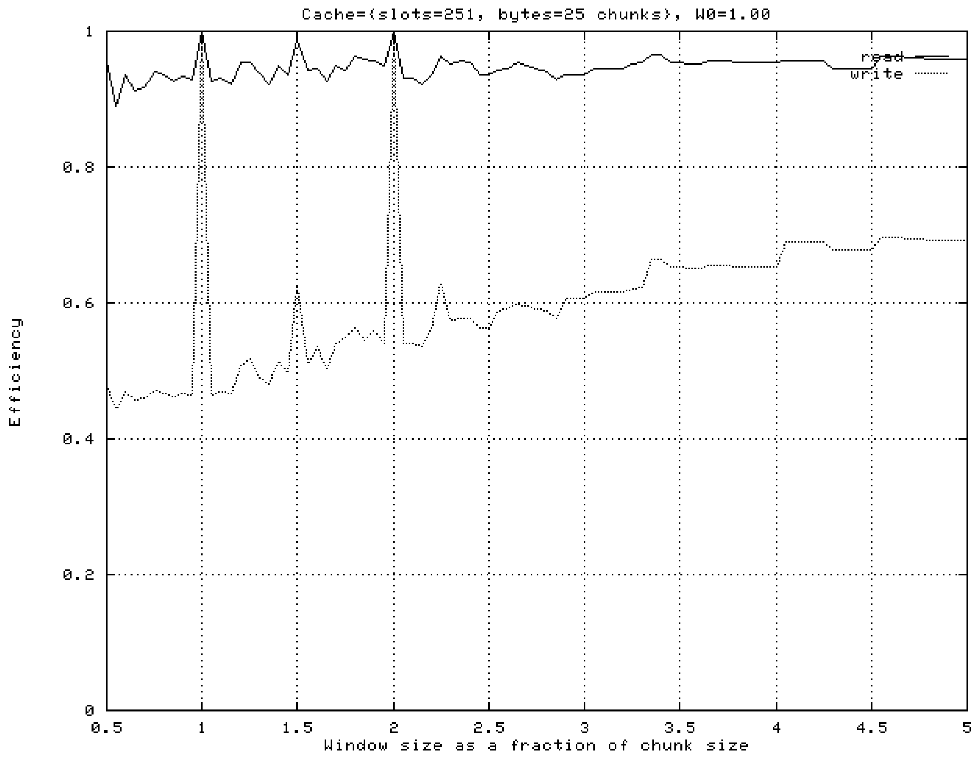


Figure 4

The read efficiency is still less than one because of collisions in the cache. The number of collisions can often be reduced by increasing the number of slots in the cache. Figure 5 shows what happens when the maximum number of slots is increased by an order of magnitude from the default (this change has no major effect on memory used by the test since the byte limit was not increased for the cache).

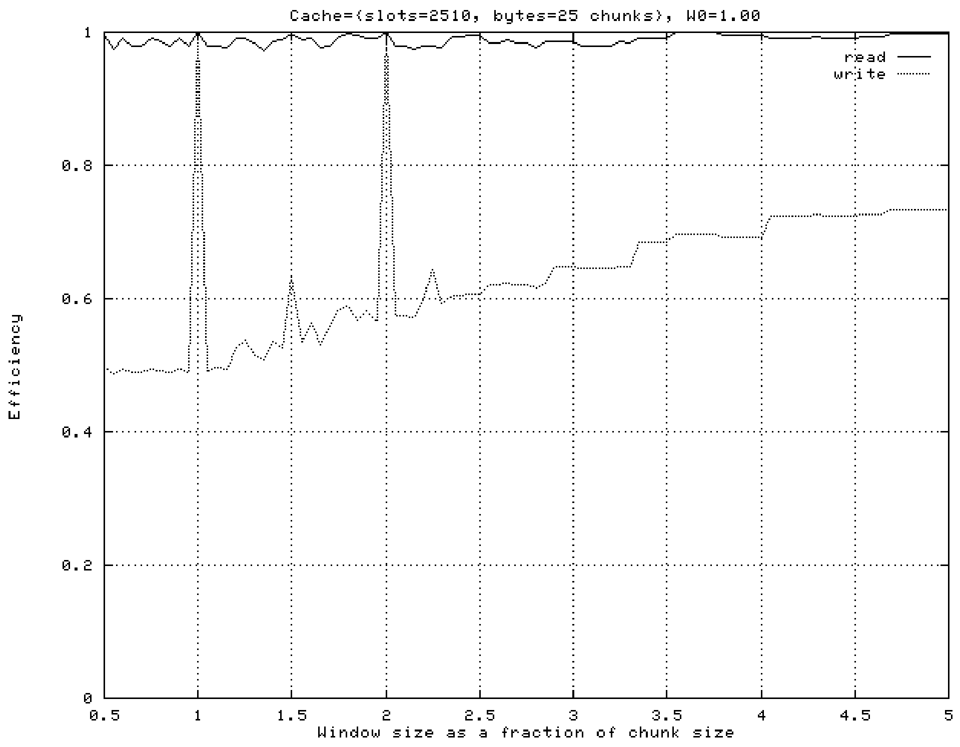


Figure 5

Although the application eventually overwrites every chunk completely the library has know way of knowing this before hand since most calls to `H5Dwrite()` modify only a portion of any given chunk. Therefore, the first modification of a chunk will cause the chunk to be read from disk into the chunk buffer through the filter pipeline. Eventually HDF5 might contain a data set transfer property that can turn off this read operation resulting in write efficiency which is equal to read efficiency.

13.4. Fragmentation

Even if the application transfers the entire dataset contents with a single call to `H5Dread()` or `H5Dwrite()` it's possible the request will be broken into smaller, more manageable pieces by the library. This is almost certainly true if the data transfer includes a type conversion.

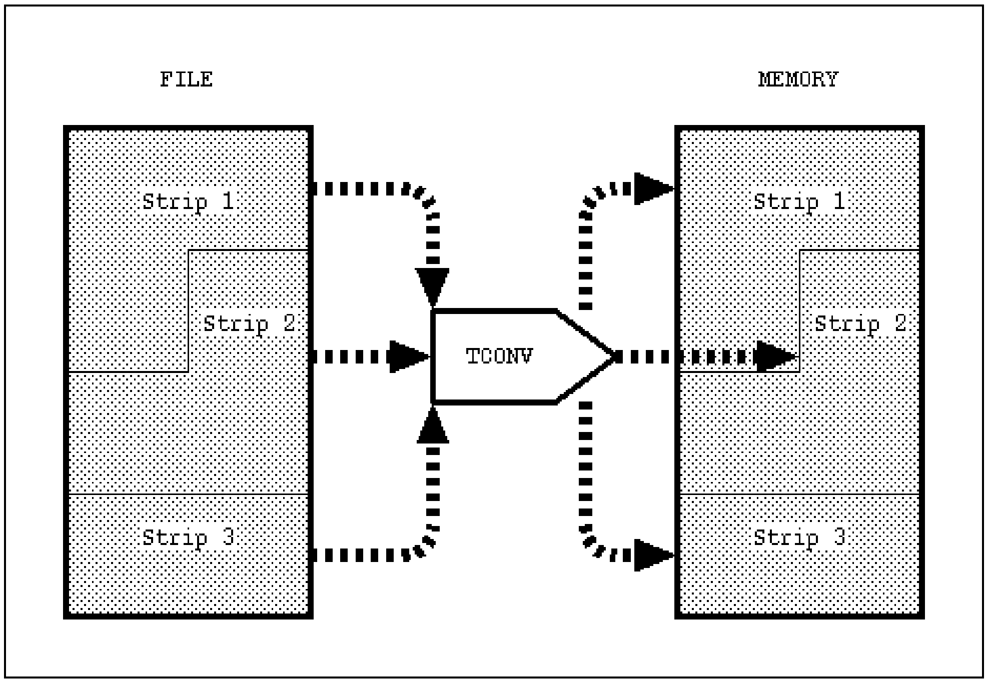


Figure 6

By default the strip size is 1MB but it can be changed by calling `H5Pset_buffer()`.

13.5. File Storage Overhead

The chunks of the dataset are allocated at independent locations throughout the HDF5 file and a B-tree maps chunk N-dimensional addresses to file addresses. The more chunks that are allocated for a dataset the larger the B-tree. Large B-trees have two disadvantages:

- The file storage overhead is higher and more disk I/O is required to traverse the tree from root to leaves.
- The increased number of B-tree nodes will result in higher contention for the meta data cache.

There are three ways to reduce the number of B-tree nodes. The obvious way is to reduce the number of chunks by choosing a larger chunk size (doubling the chunk size will cut the number of B-tree nodes in half). Another method is to adjust the split ratios for the B-tree by calling `H5Pset_split_ratios()`, but this method typically results in only a slight improvement over the default settings. Finally, the out-degree of each node can be increased by calling `H5Pset_istore_k()` (increasing the out degree actually increases file overhead while decreasing the number of nodes).

Footnote 1: Parallel versions of the library can access individual bytes of a chunk when the underlying file uses MPI-IO.
Footnote 2: The raw data chunk cache was added before the second alpha release.

Last modified: 14 October 1999

14. Mounting Files

14.1. Purpose

This document contrasts two methods for mounting an hdf5 file on another hdf5 file: the case where the relationship between files is a tree and the case where it's a graph. The tree case simplifies current working group functions and allows symbolic links to point into ancestor files whereas the graph case is more consistent with the organization of groups within a particular file.

14.2. Definitions

If file `child` is mounted on file `parent` at group `/mnt` in `parent` then the contents of the root group of `child` will appear in the group `/mnt` of `parent`. The group `/mnt` is called the *mount point* of the child in the parent.

14.3. Common Features

These features are common to both mounting schemes.

- The previous contents of `/mnt` in `parent` is temporarily hidden. If objects in that group had names from other groups then the objects will still be visible by those other names.

- The mount point is actually an OID (not a name) so if there are other names besides `/mnt` for that group then the root group of the child will be visible in all those names.

- At most one file can be mounted per mount point but a parent can have any number of mounted children.

- Name lookups will entail a search through the mount table at each stage of the lookup. The search will be O(log *N*) where *N* is the number of children mounted on that file.

- Files open for read-only can be mounted on other files that are open for read-only. Mounting a file in no way changes the contents of the file.

- Mounting a child may hide mount points that exist below the child's mount point, but it does not otherwise affect mounted files.

- Hard links cannot cross file boundaries. An object cannot be moved or renamed with `H5Gmove()` in such a way that the new location would be in a different file than the original location.

- The child can be accessed in a manner different from the parent. For instance, a read-write child in a read-only parent, a parallel child in a serial parent, *etc*.

- If some object in the child is open and the child is unmounted and/or closed, the object will remain open and accessible until explicitly closed. As in the mountless case, the underlying UNIX file will be held open until all member objects are closed.

- Current working groups that point into a child will remain open and usable even after the child has been unmounted and/or closed.

- Datasets that share a committed datatype must reside in the same file as the datatype.

14.4. Contrasting Features

Tree	Graph
The set of mount-related files makes a tree.	The set of mount-related files makes a directed graph.
A file can be mounted at only one mount point.	A file can be mounted at any number of mount points.
Symbolic links in the child that have a link value which is an absolute name can be interpreted with respect to the root group of either the child or the root of the mount tree, a property which is determined when the child is mounted.	Symbolic links in the child that have a link value which is an absolute name are interpreted with respect to the root group of the child.
Closing a child causes it to be unmounted from the parent.	Closing a child has no effect on its relationship with the parent. One can continue to access the child contents through the parent.
Closing the parent recursively unmounts and closes all mounted children.	Closing the parent unmounts all children but does not close them or unmount their children.
The current working group functions `H5Gset()`, `H5Gpush()`, and `H5Gpop()` operate on the root of the mount tree.	The current working group functions operate on the file specified by their first argument.
Absolute name lookups (like for `H5Dopen()`) are always performed with respect to the root of the mount tree.	Absolute name lookups are performed with respect to the file specified by the first argument.
Relative name lookups (like for `H5Dopen()`) are always performed with respect to the specified group or the current working group of the root of the mount tree.	Relative name lookups are always performed with respect to the specified group or the current working group of the file specified by the first argument.
Mounting a child temporarily hides the current working group stack for that child	Mounting a child has no effect on its current working group stack.
Calling `H5Fflush()` will flush all files of the mount tree regardless of which file is specified as the argument.	Calling `H5Fflush()` will flush only the specified file.

14.5. Functions

`herr_t H5Fmount(hid_t loc, const char *name, hid_t child, hid_t plist)`

The file *child* is mounted at the specified location in the parent. The *loc* and *name* specify the mount point, a group in the parent. The *plist* argument is an optional mount property list. The call will fail if some file is already mounted on the specified group.

Tree	Graph
The call will fail if the child is already mounted elsewhere.	A child can be mounted at numerous mount points.
The call will fail if the child is an ancestor of the parent.	The mount graph is allowed to have cycles.
Subsequently closing the child will cause it to be unmounted from the parent.	Closing the child has no effect on its mount relationship with the parent.

`herr_t H5Funmount(hid_t loc, const char *name)`

Any file mounted at the group specified by *loc* and *name* is unmounted. The child is not closed. This function fails if no child is mounted at the specified point.

`hid_t H5Pcreate(H5P_MOUNT)`

Creates and returns a new mount property list initialized with default values.

`herr_t H5Pset_symlink_locality(hid_t plist, H5G_symlink_t locality)`

`herr_t H5Pget_symlink_locality(hid_t plist, H5G_symlink_t *locality)`

These functions exist only for the tree scheme. They set or query the property that determines whether symbolic links with absolute name value in the child are looked up with respect to the child or to the mount root. The possible values are `H5G_SYMLINK_LOCAL` or `H5G_SYMLINK_GLOBAL` (the default).

`hid_t H5Freopen(hid_t file)`

A file handle is reopened, creating an additional file handle. The new file handle refers to the same file but has an empty current working group stack.

Tree	Graph
The new handle is not mounted but the old handle continues to be mounted.	The new handle is mounted at the same location(s) as the original handle.

14.6. Example

A file `eos.h5` contains data which is constant for all problems. The output of a particular physics application is dumped into `data1.h5` and `data2.h5` and the physics expects various constants from `eos.h5` in the `eos` group of the two data files. Instead of copying the contents of `eos.h5` into every physics output file we simply mount `eos.h5` as a read-only child of `data1.h5` and `data2.h5`.

Tree

```
/* Create data1.h5 */
data1 = H5Fcreate("data1.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
H5Gclose(H5Gcreate(data1, "/eos", 0));
H5Gset_comment(data1, "/eos", "EOS mount point");

/* Create data2.h5 */
data2 = H5Fcreate("data2.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
H5Gclose(H5Gcreate(data2, "/eos", 0));
H5Gset_comment(data2, "/eos", "EOS mount point");

/* Open eos.h5 and mount it in both files */
eos1 = H5Fopen("eos.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
H5Fmount(data1, "/eos", eos1, H5P_DEFAULT);
eos2 = H5Freopen(eos1);
H5Fmount(data2, "/eos", eos2, H5P_DEFAULT);

    ... physics output ...

H5Fclose(data1);
H5Fclose(data2);
```

Graph

```
/* Create data1.h5 */
data1 = H5Fcreate("data1.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
H5Gclose(H5Gcreate(data1, "/eos", 0));
H5Gset_comment(data1, "/eos", "EOS mount point");

/* Create data2.h5 */
data2 = H5Fcreate("data2.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
H5Gclose(H5Gcreate(data2, "/eos", 0));
H5Gset_comment(data2, "/eos", "EOS mount point");

/* Open eos.h5 and mount it in both files */
eos = H5Fopen("eos.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
H5Fmount(data1, "/eos", eos, H5P_DEFAULT);
H5Fmount(data2, "/eos", eos, H5P_DEFAULT);
H5Fclose(eos);

    ... physics output ...

H5Fclose(data1);
H5Fclose(data2);
```

Last modified: 14 October 1999

15. Performance Analysis and Issues

15.1. Introduction

This section includes brief discussions of performance issues in HDF5 and performance analysis tools for HDF5 or pointers to such discussions.

15.2. Dataset Chunking

Appropriate dataset chunking can make a siginificant difference in HDF5 performance. This topic is discussed in *Dataset Chunking Issues* elsewhere in this *User's Guide*.

15.3. Use of the Pablo Instrumentation of HDF5

Pablo HDF5 Trace software provides a means of measuring the performance of programs using HDF5.

The Pablo software consists of an instrumented copy of the HDF5 library, the Pablo Trace and Trace Extensions libraries, and some utilities for processing the output. The instrumented version of the HDF5 library has hooks inserted into the HDF5 code which call routines in the Pablo Trace library just after entry to each instrumented HDF5 routine and just prior to exit from the routine. The Pablo Trace Extension library has programs that track the I/O activity between the entry and exit of the HDF5 routine during execution.

A few lines of code must be inserted in the user's main program to enable tracing and to specify which HDF5 procedures are to be traced. The program is linked with the special HDF5 and Pablo libraries to produce an executable. Running this executable on a single processor produces an output file called the trace file which contains records, called Pablo Self-Defining Data Format (SDDF) records, which can later be analyzed using the HDF5 Analysis Utilities. The HDF5 Analysis Utilites can be used to interpret the SDDF records in the trace files to produce a report describing the HDF5 IO activity that occurred during execution.

For further instructions, see the file `READ_ME` in the `$(toplevel)/hdf5/pablo/` subdirectory of the HDF5 source code distribution.

For further information about Pablo and the Self-Defining Data Format, visit the Pablo website at the following URL:

```
http://www-pablo.cs.uiuc.edu/
```

Last modified: 14 October 1999

16. Debugging HDF5 Applications

16.1. Introduction

The HDF5 library contains a number of debugging features to make programmers' lives easier including the ability to print detailed error messages, check invariant conditions, display timings and other statistics, and trace API function calls and return values.

Error Messages

Error messages are normally displayed automatically on the standard error stream and include a stack trace of the library including file names, line numbers, and function names. The application has complete control over how error messages are displayed and can disable the display on a permanent or temporary basis. Refer to the documentation for the H5E error handling package.

Invariant Conditions

Unless `NDEBUG` is defined during compiling, the library will include code to verify that invariant conditions have the expected values. When a problem is detected the library will display the file and line number within the library and the invariant condition that failed. A core dump may be generated for post mortem debugging. The code to perform these checks can be included on a per-package bases.

Timings and Statistics

The library can be configured to accumulate certain statistics about things like cache performance, datatype conversion, data space conversion, and data filters. The code is included on a per-package basis and enabled at runtime by an environment variable.

API Tracing

All API calls made by an application can be displayed and include formal argument names and actual values and the function return value. This code is also conditionally included at compile time and enabled at runtime.

The statistics and tracing can be displayed on any output stream (including streams opened by the shell) with output from different packages even going to different streams.

16.2. Error Messages

By default any API function that fails will print an error stack to the standard error stream.

```
HDF5-DIAG: Error detected in thread 0.  Back trace follows.
  #000: H5F.c line 1245 in H5Fopen(): unable to open file
    major(04): File interface
    minor(10): Unable to open file
  #001: H5F.c line 846 in H5F_open(): file does not exist
    major(04): File interface
    minor(10): Unable to open file
```

The error handling package (H5E) is described section 9, *Error Handling*, in this *User’s Guide*.

16.3. Invariant Conditions

To include checks for invariant conditions the library should be configured with `--disable-production`, the default for versions before 1.2. The library designers have made every attempt to handle error conditions gracefully but an invariant condition assertion may fail in certain cases. The output from a failure usually looks something like this:

```
Assertion failed: H5.c:123: i<NELMTS(H5_debug_g)
IOT Trap, core dumped.
```

16.4. Timings and Statistics

Code to accumulate statistics is included at compile time by using the `--enable-debug` configure switch. The switch can be followed by an equal sign and a comma-separated list of package names or else a default list is used.

Name	Default	Description
a	No	Attributes
ac	Yes	Meta data cache
b	Yes	B-Trees
d	Yes	Datasets
e	Yes	Error handling
f	Yes	Files
g	Yes	Groups
hg	Yes	Global heap

hl	No	Local heaps
i	Yes	Interface abstraction
mf	No	File memory management
mm	Yes	Library memory managment
o	No	Object headers and messages
p	Yes	Property lists
s	Yes	Data spaces
t	Yes	Datatypes
v	Yes	Vectors
z	Yes	Raw data filters

In addition to including the code at compile time the application must enable each package at runtime. This is done by listing the package names in the `HDF5_DEBUG` environment variable. That variable may also contain file descriptor numbers (the default is '2') which control the output for all following packages up to the next file number. The word `all` refers to all packages. Any word my be preceded by a minus sign to turn debugging off for the package.

Sample debug specifications

`all`	This causes debugging output from all packages to be sent to the standard error stream.
`all -t –s`	Debugging output for all packages except datatypes and data spaces will appear on the standard error stream.
`-all ac 255 t,s`	This disables all debugging even if the default was to debug something, then output from the meta data cache is send to the standard error stream and output from data types and spaces is sent to file descriptor 255 which should be redirected by the shell.

The components of the `HDF5_DEBUG` value may be separated by any non-lowercase letter.

16.5. API Tracing

The HDF5 library can trace API calls by printing the function name, the argument names and their values, and the return value. Some people like to see lots of output during program execution instead of using a good symbolic debugger, and this feature is intended for their consumption. For example, the output from `h5ls foo` after turning on tracing, includes:

```
H5Tcopy(type=184549388) = 184549419 (type);
H5Tcopy(type=184549392) = 184549424 (type);
H5Tlock(type=184549424) = SUCCEED;
H5Tcopy(type=184549393) = 184549425 (type);
H5Tlock(type=184549425) = SUCCEED;
H5Fopen(filename="foo", flags=0, access=H5P_DEFAULT) = FAIL;
HDF5-DIAG: Error detected in thread 0.  Back trace follows.
  #000: H5F.c line 1245 in H5Fopen(): unable to open file
    major(04): File interface
    minor(10): Unable to open file
  #001: H5F.c line 846 in H5F_open(): file does not exist
    major(04): File interface
    minor(10): Unable to open file
```

The code that performs the tracing must be included in the library by specifying the `--enable-trace` configuration switch (the default for versions before 1.2). Then the word `trace` must appear in the value of the `HDF5_DEBUG` variable. The output will appear on the last file descriptor before the word `trace` or two (standard error) by default.

To display the trace on the standard error stream:

```
$ env HDF5_DEBUG=trace a.out
```

To send the trace to a file:

```
$ env HDF5_DEBUG="55 trace" a.out 55trace-output
```

Performance

If the library was not configured for tracing then there is no unnecessary overhead since all tracing code is excluded. However, if tracing is enabled but not used there is a small penalty. First, code size is larger because of extra statically-declared character strings used to store argument types and names and extra auto variable pointer in each function. Also, execution is slower because each function sets and tests a local variable and each API function calls the `H5_trace()` function.

If tracing is enabled and turned on then the penalties from the previous paragraph apply plus the time required to format each line of tracing information. There is also an extra call to H5_trace() for each API function to print the return value.

Safety

The tracing mechanism is invoked for each API function before arguments are checked for validity. If bad arguments are passed to an API function it could result in a segmentation fault. However, the tracing output is line-buffered so all previous output will appear.

Completeness

There are two API functions that don't participate in tracing. They are `H5Eprint()` and `H5Eprint_cb()` because their participation would mess up output during automatic error reporting.

On the other hand, a number of API functions are called during library initialization and they print tracing information.

Implementation

For those interested in the implementation here is a description. Each API function should have a call to one of the `H5TRACE()` macros immediately after the `FUNC_ENTER()` macro. The first argument is the return type encoded as a string. The second argument is the types of all the function arguments encoded as a string. The remaining arguments are the function arguments. This macro was designed to be as terse and unobtrousive as possible.

In order to keep the `H5TRACE()` calls synchronized with the source code we've written a perl script which gets called automatically just before Makefile dependencies are calculated for the file. However, this only works when one is using GNU make. To reinstrument the tracing explicitly, invoke the `trace` program from the hdf5 bin directory with the names of the source files that need to be updated. If any file needs to be modified then a backup is created by appending a tilde to the file name.

Explicit Instrumentation

```
$ ../bin/trace *.c
H5E.c: in function ‘H5Ewalk_cb’:
H5E.c:336: warning: trace info was not inserted
```

Note: The warning message is the result of a comment of the form `/*NO TRACE*/` somewhere in the function body. Tracing information will not be updated or inserted if such a comment exists.

Error messages have the same format as a compiler so that they can be parsed from program development environments like Emacs. Any function which generates an error will not be modified.

Last modified: 14 October 1999

17. HDF5 Library Environment Variables and Configuration Parameters

17.1. Environment Variables

The HDF5 library uses UNIX environment variables to control or adjust certain library features at runtime. The variables and their defined effects are as follows:

`HDF5_DEBUG`

Defines a list of debugging switches documented in the *Debugging* section of the *HDF5 User's Guide*.

`HDF5_NOCLEANUP`

When set to a non-empty value, the programs in the test directory do not remove temporary HDF5 data files. The default is for each test to remove the files before exit.

`HDF5_MPI_OPT_TYPES` (for parallel beta version only)

When set to `1`, PHDF5 will use the MPI optimized code to perform parallel read/write accesses to datasets. Currently, this optimization fails when accessing extendable datasets. The default is not to use the optimized code.

`HDF5_MPI_1_METAWRITE` (for parallel beta version only)

When set to `1`, PHDF5 will write the metadata via process 0 of each opened parallel HDF5 file. This should improve I/O throughput. The default is not to use this optimization.

17.2. Configuration Parameters

The HDF5 configuration script accepts a list of parameters to control configuration features when creating the Makefiles for the library. The command

configure –help

will display the current list of parameters and their effects.

Last modified: 14 October 1999

18. DDL in BNF for HDF5

18.1. Introduction

This document contains the data description language (DDL) for an HDF5 file. The description is in Backus-Naur Form.

18.2. Explanation of Symbols

This section contains a brief explanation of the symbols used in the DDL.

`::=`	defined as
`<tname>`	a token with the name tname
`<a> | <b>`	one of `<a>` or `<b>`
`<a>opt`	zero or one occurrence of `<a>`
`<a>*`	zero or more occurrence of `<a>`
`<a>+`	one or more occurrence of `<a>`
`TBD`	To Be Decided

18.3. The DDL

```
<file> ::= HDF5 <file_name> { <file_boot_block>opt <root_group> }

<file_name> ::= <identifier>

<file_boot_block> ::= BOOT_BLOCK { <boot_block_content> }

<boot_block_content> ::= TBD

<root_group> ::= GROUP "/" { <unamed_datatype>* <group_attribute>* <group_member>* }

<unamed_datatype> ::= DATATYPE <unamed_type_name> { <compound_type> }

<unamed_type_name> ::= the assigned name for unamed type is in the form of
                      #oid1:oid2, where oid1 and oid2 are the object ids of the type

<compound_type> ::= <member_type_def>+

<member_type_def> ::= <scalar_type_def> | <array_type_def>

<scalar_type_def> ::= <atomic_type> <field_name> ;

<atomic_type> ::= <integer> | <float> | <time> | <string> | <bitfield> | <opaque> |
                 <reference> | <enum>

<integer> ::=  H5T_STD_I8BE | H5T_STD_I8LE | H5T_STD_I16BE | H5T_STD_I16LE |
H5T_STD_I32BE |
              H5T_STD_I32LE | H5T_STD_I64BE | H5T_STD_I64LE |  H5T_STD_U8BE |
              H5T_STD_U8LE | H5T_STD_U16BE | H5T_STD_U16LE | H5T_STD_U32BE |
              H5T_STD_U32LE | H5T_STD_U64BE | H5T_STD_U64LE | H5T_NATIVE_CHAR |
              H5T_NATIVE_UCHAR | H5T_NATIVE_SHORT | H5T_NATIVE_USHORT |
              H5T_NATIVE_INT | H5T_NATIVE_UINT | H5T_NATIVE_LONG | H5T_NATIVE_ULONG |
              H5T_NATIVE_LLONG | H5T_NATIVE_ULLONG

<float> ::= H5T_IEEE_F32BE | H5T_IEEE_F32LE | H5T_IEEE_F64BE |  H5T_IEEE_F64LE |
            H5T_NATIVE_FLOAT |  H5T_NATIVE_DOUBLE | H5T_NATIVE_LDOUBLE
```

```
<time> ::= TBD

<string> ::= { STRSIZE <strsize> ;
               STRPAD <strpad> ;
               CSET <cset> ;
               CTYPE <ctype> ; }

<strsize> ::= an integer

<strpad> ::= H5T_STR_NULLTERM | H5T_STR_NULLPAD | H5T_STR_SPACEPAD

<cset> ::= H5T_CSET_ASCII

<ctype> ::= H5T_C_S1 | H5T_FORTRAN_S1

<bitfield> ::= TBD

<opaque> ::= TBD

<reference> ::= H5T_REFERENCE

<field_name> ::= <identifier>

<array_type_def> ::= <atomic_type> <field_name> <dim_sizes> ;

<dim_sizes> ::= [dimsize1][dimsize2]..., where dimsize1, dimsize2 are integers

<group_attribute> ::= <attribute>

<attribute> ::= ATTRIBUTE <attr_name> { <datatype>
                                        <dataspace>
                                        <data>opt  }
// <datatype> and <dataspace> must appear before <data>.

<attr_name> ::= <identifier>

<datatype> ::= DATATYPE { <atomic_type> }  |
               DATATYPE { <compound_type> } |
               DATATYPE { <named_type> }

<enum> ::= H5T_ENUM { <integer>; <enum_def>+  }

<enum_def> ::= <enum_symbol> <enum_val>;

<enum_symbol> ::= <identifier>

<enum_val> ::= an integer;

<named_type> ::= <path_name>

<path_name> ::= <identifier>

<dataspace> ::= DATASPACE { SCALAR } |
                DATASPACE { SIMPLE <current_dims> / <max_dims> } |
                DATASPACE { COMPLEX <ds_definition>+ }
                DATASPACE { <dataspace_name> } |

<current_dims> ::= (i1, i2, ... ), where ik is an integer, k = 1,2,...

<max_dims> ::= (i1, i2, ... ) where ik is an integer or H5S_UNLIMITED

<ds_definition> ::= TBD
```

```
<dataspace_name> ::= <identifier>

<data> ::= DATA { <scalar_space_data> | <simple_space_data> | <complex_space_data> }

<scalar_space_data> ::= <atomic_scalar_data> | <compound_scalar_data>

<atomic_scalar_data> :: = <integer_data> | <float_data> | <time_data> | <string_data> |
                          <bitfield_data> | <opaque_data> | <enum_data> |
<reference_data>

<integer_data> ::= an integer

<float_data> ::= a floating point number

<time_data> ::= TBD

<string_data> ::= a string
// A string is enclosed in double quotes.
// If a string is displayed on more than one line, string concatenate operator ’//’is
used.

<bitfield_data> ::= TBD

<opaque_data> ::= TBD

<enum_data> ::= <enum_symbol>
//maybe will be <enum_symbol> in the future

<reference_data> ::= <object_ref_data> | <data_region_data> | NULL

<object_ref_data> ::= <object_type> <object_id>

<object_type> ::= DATASET | GROUP | DATATYPE

<object_id> ::= an integer:an integer

<data_region_data> ::= H5T_STD_REF_DSETREG <object_id> {<data_region_data_info>,
                     <data_region_data_info>, ...}

<data_region_data_info> ::= <region_info> | <point_info>

<region_info> ::= (<lower_bound>:<upper_bound>, <lower_bound>:<upper_bound>, ...)

<lower_bound> ::= an integer

<upper_bound> ::= an integer

<point_info> ::= (an integer, an integer, ...)

<compound_scalar_data> ::= { [ <member_data> ], [ <member_data> ], ... }

<member_data> ::= <atomic_scalar_data> | <atomic_simple_data>

<atomic_simple_data> :: = <atomic_element>, <atomic_element>, ...

<atomic_element> ::= <atomic_scalar_data>

<simple_space_data> :: = <atomic_simple_data> | <compound_simple_data>

<compound_simple_data> ::= <compoud_element>, <compound_element>, ...

<compound_element> ::= <compound_scalar_data>
```

```
<complex_space_data> ::= TBD

<group_member> ::= <named_datatype> | <named_dataspace> | <group> | <dataset> |
                   <softlink>

<named_datatype> ::= DATATYPE <type_name> { <compound_type> }

<type_name> ::= <identifier>

<named_dataspace> ::= TBD

<group> ::= GROUP <group_name> { <hardlink> } |
            GROUP <group_name> { <group_attribute>* <group_member>* }

<group_name> ::= <identifier>

<hardlink> ::= HARDLINK <path_name>

<dataset> ::= DATASET <dataset_name> { <hardlink> } |
              DATASET <dataset_name> { <datatype>
                                       <dataspace>
                                       <storagelayout>opt
                                       <compression>opt
                                       <dataset_attribute>*
                                       <data>opt  }
// Tokens within {} can be in any order  as long as <data> and <dataset_attribute>
// are after <datatype> and <dataspace>.

<dataset_name> ::= <identifier>

<storagelayout> :: = STORAGELAYOUT <contiguous_layout>  |
                     STORAGELAYOUT <chunked_layout>  |
                     STORAGELAYOUT <compact_layout>  |
                     STORAGELAYOUT <external_layout>

<contiguous_layout> ::= {CONTIGUOUS}    // default

<chunked_layout> ::=  {CHUNKED <dims> }

<dims> ::= (i1, i2, ... ), ik is an integer, k = 1,2,...

<compact_layout> ::= TBD

<external_layout> ::= {EXTERNAL <external_file>+ }

<external_file> ::= (<file_name> <offset> <size>)

<offset> ::= an integer

<size> ::= an integer

<compression> :: = COMPRESSION { TBD }

<dataset_attribute> ::= <attribute>

<softlink> ::= SOFTLINK <softlink_name> { LINKTARGET <target> }

<softlink_name> ::= <identifier>

<target> ::= <identifier>

<identifier> ::= string
// character ’/’ should be used with care.
```

18.4. An Example of an HDF5 File in DDL

```
HDF5 "example.h5" {
GROUP "/" {
   ATTRIBUTE "attr1" {
      DATATYPE {
         { STRSIZE 17;
           STRPAD H5T_STR_NULLTERM;
           CSET H5T_CSET_ASCII;
           CTYPE H5T_C_S1;
         }
      }
      DATASPACE { SCALAR }
      DATA {
         "string attribute"
      }
   }
   DATASET "dset1" {
      DATATYPE { H5T_STD_I32BE }
      DATASPACE { SIMPLE ( 10, 10 ) / ( 10, 10 ) }
      DATA {
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         0, 1, 2, 3, 4, 5, 6, 7, 8, 9
      }
   }
   DATASET "dset2" {
      DATATYPE {
         H5T_STD_I32BE "a";
         H5T_IEEE_F32BE "b";
         H5T_IEEE_F64BE "c";
      }
      DATASPACE { SIMPLE ( 5 ) / ( 5 ) }
      DATA {
         {
            [ 1 ],
            [ 0.1 ],
            [ 0.01 ]
         },
         {
            [ 2 ],
            [ 0.2 ],
            [ 0.02 ]
         },
         {
            [ 3 ],
            [ 0.3 ],
            [ 0.03 ]
         },
         {
            [ 4 ],
            [ 0.4 ],
            [ 0.04 ]
         },
         {
            [ 5 ],
            [ 0.5 ],
            [ 0.05 ]
         }
      }
   }
   GROUP "group1" {
      DATASET "dset3" {
         DATATYPE {
            "/type1"
         }
         DATASPACE { SIMPLE ( 5 ) / ( 5 ) }
         DATA {
            {
               [ 0, 1, 2, 3 ],
               [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
                 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
                 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
                 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
                 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
            },
            {
               [ 0, 1, 2, 3 ],
               [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
                 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
                 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
                 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
                 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
            },
            {
               [ 0, 1, 2, 3 ],
               [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
                 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
                 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
                 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
                 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
            },
            {
               [ 0, 1, 2, 3 ],
               [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
                 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
                 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
                 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
                 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
            },
            {
               [ 0, 1, 2, 3 ],
               [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
                 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
                 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
                 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
                 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
            }
         }
      }
   }
   GROUP "group2" {
      HARDLINK "/group1"
   }
   SOFTLINK "slink1" {
      LINKTARGET "somevalue"
   }
   DATATYPE "type1" {
      H5T_STD_I32BE "a"[4];
      H5T_IEEE_F32BE "b"[5][6];
   }
}
}
```

Last modified: 14 October 1999

19. The Ragged Array Interface (H5RA)

The H5RA Interface is strictly experimental at this time; the interface may change dramatically or support for ragged arrays may be unavailable in future in releases. As a result, future releases may be unable to retrieve data stored with this interface.

Use these functions at your own risk!
Do not create any archives using this interface!

19.1. Introduction

Ragged arrays should be considered alpha quality. They were added to HDF5 to satisfy the needs of the ASCI/DMF vector bundle project; the interface and storage methods are likely to change in the future in ways that are not backward compatible.

A two-dimensional ragged array has been added to the library and built on top of other existing functionality. A ragged array is a one-dimensional array of *rows* where the length of any row is independent of the lengths of the other rows. The number of rows and the length of each row can be changed at any time (the current version does not support truncating an array by removing rows). All elements of the ragged array have the same datatype and, as with datasets, the data is type-converted between memory buffers and files.

The current implementation works best when most of the rows are approximately the same length since a two dimensional dataset can be created to hold a nominal number of elements from each row with the additional elements stored in a separate dataset which implements a heap.

A ragged array is a composite object implemented as a group with three datasets. The name of the group is the name of the ragged array. The *raw* dataset is a two-dimensional array that contains the first *N* elements of each row where *N* is determined by the application when the array is created. If most rows have fewer than *N* elements then internal fragmentation may be quite bad.

The *over* dataset is a one-dimensional array that contains elements from each row that don't fit in the *raw* dataset.

The *meta* dataset maintains information about each row such as the number of elements in the row, the location of the overflow elements in the *over* dataset (if any), and the amount of space reserved in *over* for the row. The *meta* dataset has one entry per row and is where most of the storage overhead is concentrated when rows are relatively short.

2. Opening and Closing

`hid_t H5RAcreate (hid_t` *`location`*`, const char *`*`name`*`, hid_t` *`type`*`, hid_t` *`plist`*`)`

This function creates a new ragged array by creating the group with the specified name and populating it with the component datasets (which should not be accessed independently). The dataset creation property list *plist* defines the width of the *raw* dataset; a nominal row is considered to be the width of a chunk. The *type* argument defines the datatype which will be stored in the file. A negative value is returned if the array cannot be created.

`hid_t H5RAopen (hid_t` *`location`*`, const char *`*`name`*`)`

This function opens a ragged array by opening the specified group and the component datasets (which should not be accessed indepently). A negative value is returned if the array cannot be opened.

`herr_t H5RAclose (hid_t` *`array`*`)`

All ragged arrays should be closed by calling this function. The group and component datasets will be closed automatically by the library.

19.3. Reading and Writing

In order to be as efficient as possible the ragged array layer operates on sets of contiguous rows and it is to the application’s advantage to perform I/O on as many rows at a time as possible. These functions take a starting row number and the number of rows on which to operate.

`herr_t H5RAwrite (hid_t` *`array_id`*`, hssize_t` *`start_row`*`, hsize_t` *`nrows`*`, hid_t` *`type`*`, hsize_t` *`size`*`[], void *`*`buf`*`[])`

A set of ragged array rows beginning at *start_row* and continuing for *nrows* is written to the file, converting the memory datatype *type* to the file data type which was defined when the array was created. The number of elements to write from each row is specified in the *size* array and the data for each row is pointed to from the *buf* array. The *size* and *buf* are indexed so their first element corresponds to the first row on which to operate.

`herr_t H5RAread (hid_t` *`array_id`*`, hssize_t` *`start_row`*`, hsize_t` *`nrows`*`, hid_t` *`type`*`, hsize_t` *`size`*`[], void *`*`buf`*`[])`

A set of ragged array rows beginning at *start_row* and continuing for *nrows* is read from the file, converting from the file datatype which was defined when the array was created to the memory datatype *type*. The number of elements to read from each row is specified in the *size* array and the buffers in which to place the results are pointed to by the *buf* array. On return, the *size* array will contain the actual size of the row which may be different than the requested size. When the request size is smaller than the actual size the row will be truncated; otherwise the remainder of the output buffer will be zero filled. If a pointer in the *buf* array is null then the library will ignore the corresponding *size* value and allocate a buffer large enough to hold the entire row. This function returns negative for failures with *buf* containing the original input values.

Last modified: 14 October 1999

HDF5 Glossary

Release 1.2 , October 1999

Relationships among Terms

- atomic datatype
- attribute
- chunked layout
- chunking
- compound datatype
- contiguous layout
- dataset
- dataspace
- datatype
  - atomic
  - compound
  - enumeration
  - named
  - opaque
  - variable-length
- enumeration datatype
- file
  - group
  - path
  - root group
  - super block
- file access mode
- group
  - member
  - root group
- hard link
- hyperslab
- identifier
- link
  - hard
  - soft
- member
- name
- named datatype
- opaque datatype
- path
- property list
  - data transfer
  - dataset access
  - dataset creation
  - file access
  - file creation
- root group
- selection
  - hyperslab
- serialization
- soft link
- storage layout
  - chunked
  - chunking
  - contiguous
- super block
- variable-length datatype

atomic datatype

A datatype which cannot be decomposed into smaller units at the API level.

attribute

A small dataset that can be used to describe the nature and/or the intended usage of the object it is attached to.

chunked layout

The storage layout of a chunked dataset.

chunking

A storage layout where a dataset is partitioned into fixed-size multi-dimensional chunks. Chunking tends to improve performance and facilitates dataset extensibility.

compound datatype

A collection of one or more atomic types or small arrays of such types. Similar to a struct in C or a common block in Fortran.

contiguous layout

The storage layout of a dataset that is not chunked, so that the entire data portion of the dataset is stored in a single contiguous block.

data transfer property list

The data transfer property list is used to control various aspects of the I/O, such as caching hints or collective I/O information.

dataset

A multi-dimensional array of data elements, together with supporting metadata.

dataset access property list

A property list containing information on how a dataset is to be accessed.

dataset creation property list

A property list containing information on how raw data is organized on disk and how the raw data is compressed.

dataspace

An object that describes the dimensionality of the data array. A dataspace is either a regular N-dimensional array of data points, called a simple dataspace, or a more general collection of data points organized in another manner, called a complex dataspace.

datatype

An object that describes the storage format of the individual data points of a data set. There are two categories of datatypes: atomic and compound datatypes. An atomic type is a type which cannot be decomposed into smaller units at the API level. A compound datatype is a collection of one or more atomic types or small arrays of such types.

enumeration datatype

A one-to-one mapping between a set of symbols and a set of integer values, and an order is imposed on the symbols by their integer values. The symbols are passed between the application and library as character strings and all the values for a particular enumeration datatype are of the same integer type, which is not necessarily a native type.

file

A container for storing grouped collections of multi-dimensional arrays containing scientific data.

file access mode

Determines whether an existing file will be overwritten, opened for read-only access, or opened for read/write access. All newly created files are opened for both reading and writing.

file access property list

File access property lists are used to control different methods of performing I/O on files:

file creation property list

The property list used to control file metadata.

group

A structure containing zero or more HDF5 objects, together with supporting metadata. The two primary HDF5 objects are datasets and groups.

hard link

A direct association between a name and the object where both exist in a single HDF5 address space.

hyperslab

A portion of a dataset. A hyperslab selection can be a logically contiguous collection of points in a dataspace or a regular pattern of points or blocks in a dataspace.

identifier

A unique entity provided by the HDF5 library and used to access an HDF5 object, such as a file, goup, dataset, datatype, etc.

link

An association between a name and the object in an HDF5 file group.

member

A group or dataset that is in another dataset, *dataset A*, is a member of *dataset A*.

name

A slash-separated list of components that uniquely identifies an element of an HDF5 file. A name begins that begins with a slash is an absolute name which is accessed beginning with the root group of the file; all other names are relative names and the associated objects are accessed beginning with the current or specified group.

named datatype

A datatype that is named and stored in a file. Naming is permanent; a datatype cannot be changed after being named.

opaque datatype

A mechanism for describing data which cannot be otherwise described by HDF5. The only properties associated with opaque types are a size in bytes and an ASCII tag.

path

The slash-separated list of components that forms the name uniquely identifying an element of an HDF5 file.

property list

A collection of name/value pairs that can be passed to other HDF5 functions to control features that are typically unimportant or whose default values are usually used.

root group

The group that is the entry point to the group graph in an HDF5 file. Every HDF5 file has exactly one root group.

selection

A subset of a dataset or a dataspace, up to the entire dataset or dataspace.

serialization

The flattening of an *N*-dimensional data object into a 1-dimensional object so that, for example, the data object can be transmitted over the network as a 1-dimensional bitstream.

soft link

An indirect association between a name and an object in an HDF5 file group.

storage layout

The manner in which a dataset is stored, either contiguous or chunked, in the HDF5 file.

super block

A block of data containing the information required to portably access HDF5 files on multiple platforms, followed by information about the groups and datasets in the file. The super block contains information about the size of offsets, lengths of objects, the number of entries in group tables, and additional version information for the file.

variable-length datatype

A sequence of an existing datatype (atomic, variable-length (VL), or compound) which are not fixed in length from one dataset location to another.

Last modified: 18 October 1999

HDF5 Reference Manual

Release 1.2
October 1999

Hierarchical Data Format (HDF) Group
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Last modified: 13 October 1999

HDF5: API Specification Reference Manual

H5: General Library Functions

These functions serve general-purpose needs of the HDF5 library and it users.

- H5open
- H5close
- H5get_libversion
- H5check_version
- H5dont_atexit

Name: H5open

Signature:

herr_t `H5open`(*void*)

Purpose:

Initializes the HDF5 library.

Description:

`H5open` initialize the library. This function is normally called automatically, but if you find that an HDF5 library function is failing inexplicably, try calling this function first.

Parameters:

None.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5close

Signature:

herr_t `H5close`(*void*)

Purpose:

Flushes all data to disk, closes file identifiers, and cleans up memory.

Description:

`H5close` flushes all data to disk, closes all file identifiers, and cleans up all memory used by the library. This function is generall called when the application calls `exit`, but may be called earlier in event of an emergency shutdown or out of desire to free all resources used by the HDF5 library.

Parameters:

None.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5dont_atexit

Signature:

herr_t `H5dont_atexit`(*void*)

Purpose:

Instructs library not to install `atexit` cleanup routine.

Description:

`H5dont_atexit` indicates to the library that an `atexit()` cleanup routine should not be installed. The major purpose for this is in situations where the library is dynamically linked into an application and is un-linked from the application before `exit()` gets called. In those situations, a routine installed with `atexit()` would jump to a routine which was no longer in memory, causing errors.

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

Parameters:

None.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5get_libversion

Signature:

herr_t `H5get_libversion`(*unsigned* *`majnum`, *unsigned* *`minnum`, *unsigned* *`relnum`)

Purpose:

Returns the HDF library release number.

Description:

`H5get_libversion` retrieves the major, minor, and release numbers of the version of the HDF library which is linked to the application.

Parameters:

unsigned *`majnum`

OUT: The major version of the library.

unsigned *`minnum`

OUT: The minor version of the library.

unsigned *`relnum`

OUT: The release number of the library.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5check_version

Signature:

herr_t `H5check_version`(*unsigned* `majnum`, *unsigned* `minnum`, *unsigned* `relnum`)

Purpose:

Description:

`H5check_version` verifies that the arguments match the version numbers compiled into the library. This function is intended to be called by the user to verify that the version of the header files compiled into the application match the version of the HDF5 library being used.

Due to the risks of data corruption or segmentation faults, `H5check_version` causes the application to abort if the version numbers do not match.

If the version numbers of the library do not match the version numbers in the header files being checked, the library calls the standard C function `abort()`.

Parameters:

unsigned `majnum`

IN: The major version of the library.

unsigned `minnum`

IN: The minor version of the library.

unsigned `relnum`

IN: The release number of the library.

Returns:

Returns a non-negative value if successful. Upon failure, this function causes the application to abort.

Last modified: 30 October 1998

H5A: Attribute Interface

Attribute API Functions

These functions create and manipulate attributes and information about attributes.

- H5Acreate
- H5Awrite
- H5Aread
- H5Aclose
- H5Aget_name
- H5Aopen_name
- H5Aopen_idx
- H5Aget_space
- H5Aget_type
- H5Aget_num_attrs
- H5Aiterate
- H5Adelete

The Attribute interface, H5A, is primarily designed to easily allow small datasets to be attached to primary datasets as metadata information. Additional goals for the H5A interface include keeping storage requirement for each attribute to a minimum and easily sharing attributes among datasets.

Because attributes are intended to be small objects, large datasets intended as additional information for a primary dataset should be stored as supplemental datasets in a group with the primary dataset. Attributes can then be attached to the group containing everything to indicate a particular type of dataset with supplemental datasets is located in the group. How small is "small" is not defined by the library and is up to the user's interpretation.

See the "Attributes" section of the *HDF5 User's Guide* for further information.

Name: H5Acreate

Signature:

hid_t `H5Acreate`(*hid_t* `loc_id`, *const char* `*name`, *hid_t* `type_id`, *hid_t* `space_id`, *hid_t* `create_plist`)

Purpose:

Creates a dataset as an attribute of another group, dataset, or named datatype.

Description:

`H5Acreate` creates an attribute which is attached to the object specified with `loc_id`. `loc_id` is an identifier of a group, dataset, or named datatype. The name specified with `name` for each attribute for an object must be unique for that object. The datatype and dataspace identifiers of the attribute, `type_id` and `space_id`, respectively, are created with the H5T and H5S interfaces, respectively. Currently only simple dataspaces are allowed for attribute dataspaces. The `create_plist_id` property list is currently unused, but will be used int the future for optional properties of attributes. The attribute identifier returned from this function must be released with `H5Aclose` or resource leaks will develop. Attempting to create an attribute with the same name as an already existing attribute will fail, leaving the pre-existing attribute in place.

Parameters:

hid_t `loc_id`

IN: Object (dataset, group, or named datatype) to be attached to.

const char `*name`

IN: Name of attribute to create.

hid_t `type_id`

IN: Identifier of datatype for attribute.

hid_t `space_id`

IN: Identifier of dataspace for attribute.

hid_t `create_plist`

IN: Identifier of creation property list (currently not used).

Returns:

Returns an attribute identifier if successful; otherwise returns a negative value.

Name: H5Aopen_name

Signature:

hid_t `H5Aopen_name`(*hid_t* `loc_id`, *const char* `*name`)

Purpose:

Opens an attribute specified by name.

Description:

`H5Aopen_name` opens an attribute specified by its name, `name`, which is attached to the object specified with `loc_id`. The location object may be either a group, dataset, or named datatype, which may have any sort of attribute. The attribute identifier returned from this function must be released with `H5Aclose` or resource leaks will develop.

Parameters:

hid_t `loc_id`

IN: Identifier of a group, dataset, or named datatype atttribute to be attached to.

const char `*name`

IN: Attribute name.

Returns:

Returns attribute identifier if successful; otherwise returns a negative value.

Name: H5Aopen_idx

Signature:

hid_t `H5Aopen_idx`(*hid_t* `loc_id`, *unsigned int* `idx`)

Purpose:

Opens the attribute specified by its index.

Description:

`H5Aopen_idx` opens an attribute which is attached to the object specified with `loc_id`. The location object may be either a group, dataset, or named datatype, all of which may have any sort of attribute. The attribute specified by the index, `idx`, indicates the attribute to access. The value of `idx` is a 0-based, non-negative integer. The attribute identifier returned from this function must be released with `H5Aclose` or resource leaks will develop.

Parameters:

hid_t `loc_id`

IN: Identifier of the group, dataset, or named datatype attribute to be attached to.

unsigned int `idx`

IN: Index of the attribute to open.

Returns:

Returns attribute identifier if successful; otherwise returns a negative value.

Name: H5Awrite

Signature:

herr_t `H5Awrite`(*hid_t* `attr_id`, *hid_t* `mem_type_id`, *void* `*buf`)

Purpose:

Writes data to an attribute.

Description:

`H5Awrite` writes an attribute, specified with `attr_id`. The attribute's memory datatype is specified with `mem_type_id`. The entire attribute is written from `buf` to the file.

Datatype conversion takes place at the time of a read or write and is automatic. See the "Data Conversion" section of *The Data Type Interface (H5T)* in the *HDF5 User's Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `attr_id`

IN: Identifier of an attribute to write.

hid_t `mem_type_id`

IN: Identifier of the attribute datatype (in memory).

void `*buf`

IN: Data to be written.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Aread

Signature:

herr_t `H5Aread`(*hid_t* `attr_id`, *hid_t* `mem_type_id`, *void* `*buf`)

Purpose:

Reads an attribute.

Description:

`H5Aread` reads an attribute, specified with `attr_id`. The attribute's memory datatype is specified with `mem_type_id`. The entire attribute is read into `buf` from the file.

Datatype conversion takes place at the time of a read or write and is automatic. See the “Data Conversion” section of *The Data Type Interface (H5T)* in the *HDF5 User's Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `attr_id`

IN: Identifier of an attribute to read.

hid_t `mem_type_id`

IN: Identifier of the attribute datatype (in memory).

void `*buf`

OUT: Buffer for data to be read.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Aget_space

Signature:

hid_t `H5Aget_space`(*hid_t* `attr_id`)

Purpose:

Gets a copy of the dataspace for an attribute.

Description:

`H5Aget_space` retrieves a copy of the dataspace for an attribute. The dataspace identifier returned from this function must be released with `H5Sclose` or resource leaks will develop.

Parameters:

hid_t `attr_id`

IN: Identifier of an attribute.

Returns:

Returns attribute dataspace identifier if successful; otherwise returns a negative value.

Name: H5Aget_type

Signature:

hid_t `H5Aget_type`(*hid_t* `attr_id`)

Purpose:

Gets an attribute datatype.

Description:

`H5Aget_type` retrieves a copy of the datatype for an attribute.

The datatype is reopened if it is a named type before returning it to the application. The datatypes returned by this function are always read-only. If an error occurs when atomizing the return datatype, then the datatype is closed.

The datatype identifier returned from this function must be released with `H5Tclose` or resource leaks will develop.

Parameters:

hid_t `attr_id`

IN: Identifier of an attribute.

Returns:

Returns a datatype identifier if successful; otherwise returns a negative value.

Name: H5Aget_name

Signature:

ssize_t `H5Aget_name`(*hid_t* `attr_id`, *size_t* `buf_size`, *char* `*buf`)

Purpose:

Gets an attribute name.

Description:

`H5Aget_name` retrieves the name of an attribute specified by the identifier, `attr_id`. Up to `buf_size` characters are stored in `buf` followed by a `\0` string terminator. If the name of the attribute is longer than `buf_size` -1, the string terminator is stored in the last position of the buffer to properly terminate the string.

Parameters:

hid_t `attr_id`

IN: Identifier of the attribute.

size_t `buf_size`

IN: The size of the buffer to store the name in.

char `*buf`

IN: Buffer to store name in.

Returns:

Returns the length of the attribute's name, which may be longer than `buf_size`, if successful. Otherwise returns a negative value.

Name: H5Aget_num_attrs

Signature:

int `H5Aget_num_attrs`(*hid_t* `loc_id`)

Purpose:

Determines the number of attributes attached to an object.

Description:

`H5Aget_num_attrs` returns the number of attributes attached to the object specified by its identifier, `loc_id`. The object can be a group, dataset, or named datatype.

Parameters:

hid_t `loc_id`

IN: Identifier of a group, dataset, or named datatype.

Returns:

Returns the number of attributes if successful; otherwise returns a negative value.

Name: H5Aiterate

Signature:

int `H5Aiterate`(*hid_t* `loc_id`, *unsigned* * `idx`, *H5A_operator_t* `op`, *void* *`op_data`)

Purpose:

Calls a user's function for each attribute on an object.

Description:

`H5Aiterate` iterates over the attributes of the object specified by its identifier, `loc_id`. The object can be a group, dataset, or named datatype. For each attribute of the object, the `op_data` and some additional information specified below are passed to the operator function `op`. The iteration begins with the attribute specified by its index, `idx`; the index for the next attribute to be processed by the operator, `op`, is returned in `idx`. If `idx` is the null pointer, then all attributes are processed.

The prototype for `H5A_operator_t` is:
```
typedef herr_t (*H5A_operator_t)(hid_t loc_id, const char *attr_name, void
*operator_data);
```

The operation receives the identifier for the group, dataset or named datatype being iterated over, `loc_id`, the name of the current attribute about the object, `attr_name`, and the pointer to the operator data passed in to `H5Aiterate`, `op_data`. The return values from an operator are:

- Zero causes the iterator to continue, returning zero when all attributes have been processed.
- Positive causes the iterator to immediately return that positive value, indicating short-circuit success. The iterator can be restarted at the next attribute.
- Negative causes the iterator to immediately return that value, indicating failure. The iterator can be restarted at the next attribute.

Parameters:

hid_t `loc_id`

IN: Identifier of a group, dataset or named datatype.

unsigned * `idx`

IN/OUT: Starting (IN) and ending (OUT) attribute index.

H5A_operator_t `op`

IN: User's function to pass each attribute to

void `*op_data`

IN/OUT: User's data to pass through to iterator operator function

Returns:

If successful, returns the return value of the last operator if it was non-zero, or zero if all attributes were processed. Otherwise returns a negative value.

Name: H5Adelete

Signature:

herr_t `H5Adelete`(*hid_t* `loc_id`, *const char* `*name`)

Purpose:

Deletes an attribute from a location.

Description:

`H5Adelete` removes the attribute specified by its name, `name`, from a dataset, group, or named datatype. This function should not be used when attribute identifiers are open on `loc_id` as it may cause the internal indexes of the attributes to change and future writes to the open attributes to produce incorrect results.

Parameters:

hid_t `loc_id`

IN: Identifier of the dataset, group, or named datatype to have the attribute deleted from.

const char `*name`

IN: Name of the attribute to delete.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Aclose

Signature:

herr_t `H5Aclose`(*hid_t* `attr_id`)

Purpose:

Closes the specified attribute.

Description:

`H5Aclose` terminates access to the attribute specified by its identifier, `attr_id`. Further use of the attribute identifier will result in undefined behavior.

Parameters:

hid_t `attr_id`

IN: Attribute to release access to.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Last modified: 20 October 1999

H5D: Datasets Interface

Dataset Object API Functions

These functions create and manipulate dataset objects, and set and retrieve their constant or persistent properties.

- H5Dcreate
- H5Dopen
- H5Dclose
- H5Dget_space
- H5Dget_type
- H5Dget_create_plist
- H5Dget_storage_size
- H5Dget_vlen_buf_size
- H5Dvlen_reclaim
- H5Dread
- H5Dwrite
- H5Diterate
- H5Dextend

Name: H5Dcreate

Signature:

hid_t `H5Dcreate`(*hid_t* `loc_id`, *const char* *`name`, *hid_t*`type_id`, *hid_t*`space_id`, *hid_t*`create_plist_id`)

Purpose:

Creates a dataset at the specified location.

Description:

`H5Dcreate` creates a data set with a name, `name`, in the file or in the group specified by the identifier `loc_id`. The dataset has the datatype and dataspace identified by `type_id` and `space_id`, respectively. The specified datatype and dataspace are the datatype and dataspace of the dataset as it will exist in the file, which may be different than in application memory. Dataset creation properties are specified by the argument `create_plist_id`.

`create_plist_id` is a `H5P_DATASET_CREATE` property list created with `H5Pcreate()` and initialized with the various functions described above. `H5Dcreate()` returns a dataset identifier for success or negative for failure. The identifier should eventually be closed by calling `H5Dclose()` to release resources it uses.

Parameters:

hid_t `loc_id`

Identifier of the file or group to create the dataset within.

const char * `name`

The name of the dataset to create.

hid_t `type_id`

Identifier of the datatype to use when creating the dataset.

hid_t `space_id`

Identifier of the dataspace to use when creating the dataset.

hid_t `create_plist_id`

Identifier of the set creation property list.

Returns:

Returns a dataset identifier if successful; otherwise returns a negative value.

Name: H5Dopen

Signature:

hid_t `H5Dopen`(*hid_t* `loc_id`, *const char* *`name`)

Purpose:

Opens an existing dataset.

Description:

`H5Dopen` opens an existing dataset for access in the file or group specified in `loc_id`. `name` is a dataset name and is used to identify the dataset in the file.

Parameters:

hid_t `loc_id`

Identifier of the dataset to open or the file or group to access the dataset within.

const char * `name`

The name of the dataset to access.

Returns:

Returns a dataset identifier if successful; otherwise returns a negative value.

Name: H5Dget_space

Signature:

hid_t `H5Dget_space`(*hid_t* `dataset_id`)

Purpose:

Returns an identifier for a copy of the dataspace for a dataset.

Description:

`H5Dget_space` returns an identifier for a copy of the dataspace for a dataset. The dataspace identifier should be released with the `H5Sclose()` function.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to query.

Returns:

Returns a dataspace identifier if successful; otherwise returns a negative value.

Name: H5Dget_type

Signature:

hid_t `H5Dget_type`(*hid_t* `dataset_id`)

Purpose:

Returns an identifier for a copy of the datatype for a dataset.

Description:

`H5Dget_type` returns an identifier for a copy of the datatype for a dataset. The datatype should be released with the `H5Tclose()` function.

If a dataset has a named datatype, then an identifier to the opened datatype is returned. Otherwise, the returned datatype is read-only. If atomization of the datatype fails, then the datatype is closed.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to query.

Returns:

Returns a datatype identifier if successful; otherwise returns a negative value.

Name: H5Dget_create_plist

Signature:

hid_t `H5Dget_create_plist`(*hid_t* `dataset_id`)

Purpose:

Returns an identifier for a copy of the dataset creation property list for a dataset.

Description:

`H5Dget_create_plist` returns an identifier for a copy of the dataset creation property list for a dataset. The creation property list identifier should be released with the `H5Pclose()` function.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to query.

Returns:

Returns a dataset creation property list identifier if successful; otherwise returns a negative value.

Name: H5Dget_storage_size

Signature:

hsize_t `H5Dget_storage_size`(*hid_t* `dataset_id`)

Purpose:

Returns the amount of storage required for a dataset.

Description:

`H5Dget_storage_size` returns the amount of storage that is required for the specified dataset, `dataset_id`. For chunked datasets, this is the number of allocated chunks times the chunk size. The return value may be zero if no data has been stored.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to query.

Returns:

Returns the amount of storage space allocated for the dataset, not counting meta data; otherwise returns a negative value.

Name: H5Dget_vlen_buf_size ***(Not yet implemented.)***

Signature:

herr_t `H5Dget_vlen_buf_size`(*hid_t* `dataset_id`, *hid_t* `type_id`, *hid_t* `space_id`, *hsize_t* `*size`)

Purpose:

Determines the number of bytes required to store VL data.

Description:

`H5Dget_vlen_buf_size` determines the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory.

`*size` is returned with the number of bytes are required to store the VL data in memory.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to query.

hid_t `type_id`

Identifier of the datatype.

hid_t `space_id`

Identifier of the dataspace.

hsize_t `*size`

The size in bytes of the buffer required to store the VL data.

Returns:

Returns non-negative value if successful; otherwise returns a negative value.

Name: H5Dvlen_reclaim

Signature:

herr_t `H5Dvlen_reclaim`(*hid_t* `type_id` *hid_t* `space_id`, *hid_t* `plist_id`, *void* `*buf`)

Purpose:

Reclaims VL datatype memory buffers.

Description:

`H5Dvlen_reclaim` reclaims memory buffers created to store VL datatypes.

The `type_id` must be the datatype stored in the buffer. The `space_id` describes the selection for the memory buffer to free the VL datatypes within. The `plist_id` is the dataset transfer property list which was used for the I/O transfer to create the buffer. And `buf` is the pointer to the buffer to be reclaimed.

The VL structures (`hvl_t`) in the user's buffer are modified to zero out the VL information after the memory has been reclaimed.

If nested VL datatypes were used to create the buffer, this routine frees them *from the bottom up*, releasing all the memory without creating memory leaks.

Parameters:

hid_t `type_id`

Identifier of the datatype.

hid_t `space_id`

Identifier of the dataspace.

hid_t `plist_id`

Identifier of the property list used to create the buffer.

void `*buf`

Pointer to the buffer to be reclaimed.

Returns:

Returns non-negative value if successful; otherwise returns a negative value.

Name: H5Dread

Signature:

herr_t `H5Dread`(*hid_t* `dataset_id`, *hid_t* `mem_type_id`, *hid_t* `mem_space_id`, *hid_t* `file_space_id`, *hid_t* `xfer_plist_id`, *void* * `buf`)

Purpose:

Reads raw data from the specified dataset into `buf`, converting from file datatype and dataspace to memory datatype and dataspace.

Description:

`H5Dread` reads a (partial) dataset, specified by its identifier `dataset_id`, from the file into the application memory buffer `buf`. Data transfer properties are defined by the argument `xfer_plist_id`. The memory datatype of the (partial) dataset is identified by the identifier `mem_type_id`. The part of the dataset to read is defined by `mem_space_id` and `file_space_id`.

`file_space_id` can be the constant `H5S_ALL`, which indicates that the entire file data space is to be referenced.

`mem_space_id` can be the constant `H5S_ALL`, in which case the memory data space is the same as the file data space defined when the dataset was created.

The number of elements in the memory data space must match the number of elements in the file data space.

`xfer_plist_id` can be the constant `H5P_DEFAULT`, in which case the default data transfer properties are used.

Datatype conversion takes place at the time of a read or write and is automatic. See the “Data Conversion” section of *The Data Type Interface (H5T)* in the *HDF5 User’s Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `dataset_id`

Identifier of the dataset read from.

hid_t `mem_type_id`

Identifier of the memory datatype.

hid_t `mem_space_id`

Identifier of the memory dataspace.

hid_t `file_space_id`

Identifier of the dataset's dataspace in the file.

hid_t `xfer_plist_id`

Identifier of a transfer property list for this I/O operation.

void * `buf`

Buffer to store data read from the file.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Dwrite

Signature:

herr_t `H5Dwrite`(*hid_t* `dataset_id`, *hid_t* `mem_type_id`, *hid_t* `mem_space_id`, *hid_t* `file_space_id`, *hid_t* `xfer_plist_id`, *const void* * `buf`)

Purpose:

Writes raw data from an application buffer `buf` to the specified dataset, converting from memory datatype and dataspace to file datatype and dataspace.

Description:

`H5Dwrite` writes a (partial) dataset, specified by its identifier `dataset_id`, from the application memory buffer `buf` into the file. Data transfer properties are defined by the argument `xfer_plist_id`. The memory datatype of the (partial) dataset is identified by the identifier `mem_type_id`. The part of the dataset to write is defined by `mem_space_id` and `file_space_id`.

`file_space_id` can be the constant `H5S_ALL`. which indicates that the entire file data space is to be referenced.

`mem_space_id` can be the constant `H5S_ALL`, in which case the memory data space is the same as the file data space defined when the dataset was created.

The number of elements in the memory data space must match the number of elements in the file data space.

`xfer_plist_id` can be the constant `H5P_DEFAULT`. in which case the default data transfer properties are used.

Writing to an external dataset will fail if the HDF5 file is not open for writing.

Datatype conversion takes place at the time of a read or write and is automatic. See the "Data Conversion" section of *The Data Type Interface (H5T)* in the *HDF5 User's Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `dataset_id`

Identifier of the dataset read from.

hid_t `mem_type_id`

Identifier of the memory datatype.

hid_t `mem_space_id`

Identifier of the memory dataspace.

hid_t `file_space_id`

Identifier of the dataset's dataspace in the file.

hid_t `xfer_plist_id`

Identifier of a transfer property list for this I/O operation.

const void * `buf`

Buffer with data to be written to the file.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Dextend

Signature:

herr_t `H5Dextend`(*hid_t* `dataset_id`, *const hsize_t* * `size`)

Purpose:

Extends a dataset with unlimited dimension.

Description:

`H5Dextend` verifies that the dataset is at least of size `size`. The dimensionality of `size` is the same as that of the dataspace of the dataset being changed. This function cannot be applied to a dataset with fixed dimensions.

Parameters:

hid_t `dataset_id`

Identifier of the dataset.

const hsize_t * `size`

Array containing the new magnitude of each dimension.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Dclose

Signature:

hid_t `H5Dclose`(*hid_t* `dataset_id`)

Purpose:

Closes the specified dataset.

Description:

`H5Dclose` ends access to a dataset specified by `dataset_id` and releases resources used by it. Further use of the dataset identifier is illegal in calls to the dataset API.

Parameters:

hid_t `dataset_id`

Identifier of the dataset to finish access to.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Diterate

Signature:

herr_t `H5Diterate`(*void* `*buf`, *hid_t* `type_id`, *hid_t* `space_id`, *H5D_operator_t* `operator`, *void* `*operator_data`)

Purpose:

Iterates over all selected elements in a dataspace.

Description:

`H5Diterate` iterates over all the elements selected in a memory buffer. The callback function is called once for each element selected in the dataspace.

The selection in the dataspace is modified so that any elements already iterated over are removed from the selection if the iteration is interrupted (by the `H5D_operator_t` function returning non-zero) before the iteration is complete; the iteration may then be re-started by the user where it left off.

Parameters:

void `*buf`

IN/OUT: Pointer to the buffer in memory containing the elements to iterate over.

hid_t `type_id`

IN: Datatype identifier for the elements stored in `buf`.

hid_t `space_id`

IN: Dataspace identifier for `buf`. Also contains the selection to iterate over.

H5D_operator_t `operator`

IN: Function pointer to the routine to be called for each element in `buf` iterated over.

void `*operator_data`

IN/OUT: Pointer to any user-defined data associated with the operation.

Returns:

Returns the return value of the last operator if it was non-zero, or zero if all elements have been processed. Otherwise returns a negative value.

Last modified: 20 October 1999

H5E: Error Interface

Error API Functions

These functions provide error handling capabilities in the HDF5 environment.

- H5Eset_auto
- H5Eget_auto
- H5Eclear
- H5Eprint
- H5Ewalk
- H5Ewalk_cb
- H5Eget_major
- H5Eget_minor

The Error interface provides error handling in the form of a stack. The `FUNC_ENTER()` macro clears the error stack whenever an interface function is entered. When an error is detected, an entry is pushed onto the stack. As the functions unwind, additional entries are pushed onto the stack. The API function will return some indication that an error occurred and the application can print the error stack.

Certain API functions in the H5E package, such as `H5Eprint()`, do not clear the error stack. Otherwise, any function which does not have an underscore immediately after the package name will clear the error stack. For instance, `H5Fopen()` clears the error stack while `H5F_open()` does not.

An error stack has a fixed maximum size. If this size is exceeded then the stack will be truncated and only the inner-most functions will have entries on the stack. This is expected to be a rare condition.

Each thread has its own error stack, but since multi-threading has not been added to the library yet, this package maintains a single error stack. The error stack is statically allocated to reduce the complexity of handling errors within the H5E package.

Name: H5Eset_auto

Signature:

herr_t `H5Eset_auto`(*H5E_auto_t* `func`, *void* *`client_data`)

Purpose:

Turns automatic error printing on or off.

Description:

`H5Eset_auto` turns on or off automatic printing of errors. When turned on (non-null `func` pointer), any API function which returns an error indication will first call `func`, passing it `client_data` as an argument.

When the library is first initialized the auto printing function is set to `H5Eprint()` (cast appropriately) and `client_data` is the standard error stream pointer, `stderr`.

Automatic stack traversal is always in the `H5E_WALK_DOWNWARD` direction.

Parameters:

H5E_auto_t `func`

Function to be called upon an error condition.

void *`client_data`

Data passed to the error function.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Eget_auto

Signature:

herr_t `H5Eget_auto`(*H5E_auto_t* * `func`, *void* **`client_data`)

Purpose:

Returns the current settings for the automatic error stack traversal function and its data.

Description:

`H5Eget_auto` returns the current settings for the automatic error stack traversal function, `func`, and its data, `client_data`. Either (or both) arguments may be null in which case the value is not returned.

Parameters:

H5E_auto_t * `func`

Current setting for the function to be called upon an error condition.

void **`client_data`

Current setting for the data passed to the error function.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Eclear

Signature:

herr_t `H5Eclear(void)`

Purpose:

Clears the error stack for the current thread.

Description:

`H5Eclear` clears the error stack for the current thread.

The stack is also cleared whenever an API function is called, with certain exceptions (for instance, `H5Eprint()`).

`H5Eclear` can fail if there are problems initializing the library.

Parameters:

None

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Eprint

Signature:

herr_t `H5Eprint`(*FILE* * `stream`)

Purpose:

Prints the error stack in a default manner.

Description:

`H5Eprint` prints the error stack on the specified stream, `stream`. Even if the error stack is empty, a one-line message will be printed:

```
HDF5-DIAG: Error detected in thread 0.
```

`H5Eprint` is a convenience function for `H5Ewalk()` with a function that prints error messages. Users are encouraged to write there own more specific error handlers.

Parameters:

FILE * `stream`

File pointer, or stderr if NULL.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Ewalk

Signature:

herr_t `H5Ewalk`(*H5E_direction_t* `direction`, *H5E_walk_t* `func`, *void* * `client_data`)

Purpose:

Walks the error stack for the current thread, calling a specified function.

Description:

`H5Ewalk` walks the error stack for the current thread and calls the specified function for each error along the way.

`direction` determines whether the stack is walked from the inside out or the outside in. A value of `H5E_WALK_UPWARD` means begin with the most specific error and end at the API; a value of `H5E_WALK_DOWNWARD` means to start at the API and end at the inner-most function where the error was first detected.

`func` will be called for each error in the error stack. Its arguments will include an index number (beginning at zero regardless of stack traversal direction), an error stack entry, and the `client_data` pointer passed to `H5E_print`.

`H5Ewalk` can fail if there are problems initializing the library.

Parameters:

H5E_direction_t `direction`

Direction in which the error stack is to be walked.

H5E_walk_t `func`

Function to be called for each error encountered.

void * `client_data`

Data to be passed with `func`.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Ewalk_cb

Signature:

herr_t `H5Ewalk_cb`(*int* `n`, *H5E_error_t* *`err_desc`, *void* *`client_data`)

Purpose:

Default error stack traversal callback function that prints error messages to the specified output stream.

Description:

`H5Ewalk_cb` is a default error stack traversal callback function that prints error messages to the specified output stream. It is not meant to be called directly but rather as an argument to the `H5Ewalk()` function. This function is called also by `H5Eprint()`. Application writers are encouraged to use this function as a model for their own error stack walking functions.

`n` is a counter for how many times this function has been called for this particular traversal of the stack. It always begins at zero for the first error on the stack (either the top or bottom error, or even both, depending on the traversal direction and the size of the stack).

`err_desc` is an error description. It contains all the information about a particular error.

`client_data` is the same pointer that was passed as the `client_data` argument of `H5Ewalk()`. It is expected to be a file pointer (or stderr if null).

Parameters:

int `n`

Number of times this function has been called for this traversal of the stack.

H5E_error_t `*err_desc`

Error description.

void `*client_data`

A file pointer, or stderr if null.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Eget_major

Signature:

const char * `H5Eget_major`(*H5E_major_t* `n`)

Purpose:

Returns a character string describing an error specified by a major error number.

Description:

Given a major error number, `H5Eget_major` returns a constant character string that describes the error.

Parameters:

H5E_major_t `n`

Major error number.

Returns:

Returns a character string describing the error if successful. Otherwise returns "Invalid major error number."

Name: H5Eget_minor

Signature:

const char * `H5Eget_minor`(*H5E_minor_t* `n`)

Purpose:

Returns a character string describing an error specified by a minor error number.

Description:

Given a minor error number, `H5Eget_minor` returns a constant character string that describes the error.

Parameters:

H5E_minor_t n

Minor error number.

Returns:

Returns a character string describing the error if successful. Otherwise returns "Invalid minor error number."

Last modified: 30 October 1998

H5F: File Interface

File API Functions

These functions are designed to provide file-level access to HDF5 files. Further manipulation of objects inside a file is performed through one of APIs documented below.

- H5Fcreate
- H5Fopen
- H5Freopen
- H5Fclose
- H5Fflush
- H5Fis_hdf5
- H5Fmount
- H5Funmount
- H5Fget_create_plist
- H5Fget_access_plist

Name: H5Fopen

Signature:

hid_t `H5Fopen`(*const char* *`name`, *unsigned* `flags`, *hid_t* `access_id`)

Purpose:

Opens an existing file.

Description:

`H5Fopen` opens an existing file and is the primary function for accessing existing HDF5 files.

The parameter `access_id` is a file access property list identifier or `H5P_DEFAULT` for the default I/O access parameters.

The `flags` argument determines whether writing to an existing file will be allowed or not. The file is opened with read and write permission if `flags` is set to `H5F_ACC_RDWR`. All flags may be combined with the bit-wise OR operator (‘|’) to change the behavior of the file open call. The more complex behaviors of a file’s access are controlled through the file-access property list.

Files which are opened more than once return a unique identifier for each `H5Fopen()` call and can be accessed through all file identifiers.

The return value is a file identifier for the open file and it should be closed by calling `H5Fclose()` when it is no longer needed.

Parameters:

const char *`name`

Name of the file to access.

unsigned `flags`

File access flags. Allowable values include:

H5F_ACC_RDWR

Allow read and write access to file.

H5F_ACC_RDONLY

Allow read-only access to file.

H5F_ACC_DEBUG

Print debug information. (Used only by HDF5 library developers. Do not use this flag in applications.)

`H5F_ACC_RDWR` and `H5F_ACC_RDONLY` are mutually exclusive; use exactly one.

hid_t `access_id`

Identifier for the file access properties list. If parallel file access is desired, this is a collective call according to the communicator stored in the `access_id`. Use `H5P_DEFAULT` for default file access properties.

Returns:

Returns a file identifier if successful; otherwise returns a negative value.

Name: H5Fcreate

Signature:

hid_t `H5Fcreate`(*const char* *`name`, *unsigned* `flags`, *hid_t* `create_id`, *hid_t* `access_id`)

Purpose:

Creates HDF5 files.

Description:

`H5Fcreate` is the primary function for creating HDF5 files .

The `flags` parameter determines whether an existing file will be overwritten. All newly created files are opened for both reading and writing. All flags may be combined with the bit-wise OR operator (‘|’) to change the behavior of the `H5Fcreate` call.

The more complex behaviors of file creation and access are controlled through the file-creation and file-access property lists. The value of `H5P_DEFAULT` for a property list value indicates that the library should use the default values for the appropriate property list. Also see `H5Fpublic.h` for the list of supported flags.

Parameters:

const char *`name`

Name of the file to access.

uintn `flags`

File access flags. Allowable values include:

H5F_ACC_TRUNC

Truncate file, if it already exists, erasing all data previously stored in the file.

H5F_ACC_EXCL

Fail if file already exists.

H5F_ACC_DEBUG

Print debug information. (Used only by HDF5 library developers. Do not use this flag in applications.)

`H5F_ACC_TRUNC` and `H5F_ACC_EXCL` are mutually exclusive; use exactly one.

hid_t `create_id`

File creation property list identifier, used when modifying default file meta-data. Use `H5P_DEFAULT` for default file creation properties.

hid_t `access_id`

File access property list identifier. If parallel file access is desired, this is a collective call according to the communicator stored in the `access_id`. Use `H5P_DEFAULT` for default file access properties.

Returns:

Returns a file identifier if successful; otherwise returns a negative value.

Name: H5Fflush

Signature:

herr_t `H5Fflush`(*hid_t* `object_id`, *H5F_scope_t* `scope`)

Purpose:

Flushes all buffers associated with a file to disk.

Description:

`H5Fflush` causes all buffers associated with a file to be immediately flushed to disk without removing the data from the cache.

`object_id` can be any object associated with the file, including the file itself, a dataset, a group, an attribute, or a named data type.

`scope` specifies whether the scope of the flushing action is global or local. Valid values are

`H5F_SCOPE_GLOBAL`	Flushes the entire virtual file.
`H5F_SCOPE_LOCAL`	Flushes only the specified file.

Parameters:

hid_t `object_id`

Identifier of object used to identify the file.

H5F_scope_t `scope`

Specifies the scope of the flushing action.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Fis_hdf5

Signature:

htri_t `H5Fis_hdf5`(*const char* *`name`)

Purpose:

Determines whether a file is in the HDF5 format.

Description:

`H5Fis_hdf5` determines whether a file is in the HDF5 format.

Parameters:

const char *`name`

File name to check format.

Returns:

When successful, returns a positive value, for `TRUE`, or `0` (zero), for `FALSE`. Otherwise returns a negative value.

Name: H5Fget_create_plist

Signature:

hid_t `H5Fget_create_plist`(*hid_t* `file_id`)

Purpose:

Returns a file creation property list identifier.

Description:

`H5Fget_create_plist` returns a file creation property list identifier identifying the creation properties used to create this file. This function is useful for duplicating properties when creating another file.

See "File Creation Properties" in *H5P: Property List Interface* in this reference manual and "File Creation Properties" in *Files* in the *HDF5 User's Guide* for additional information and related functions.

Parameters:

hid_t `file_id`

Identifier of the file to get creation property list of

Returns:

Returns a file creation property list identifier if successful; otherwise returns a negative value.

Name: H5Fget_access_plist

Signature:

hid_t `H5Fget_access_plist`(*hid_t* `file_id`)

Purpose:

Returns a file access property list identifier.

Description:

`H5Fget_access_plist` returns the file access property list identifier of the specified file.

See "File Access Properties" in the "H5P: Property List Interface" section of this reference manual and "File Access Property Lists" in the "Files" section of the *HDF5 User's Guide* for additional information and related functions.

Parameters:

hid_t `file_id`

Identifier of file to get access property list of

Returns:

Returns a file access property list identifier if successful; otherwise returns a negative value.

Name: H5Fclose

Signature:

herr_t `H5Fclose`(*hid_t* `file_id`)

Purpose:

Terminates access to an HDF5 file.

Description:

`H5Fclose` terminates access to an HDF5 file. If this is the last file identifier open for a file and if access identifiers are still in use, this function will fail.

Parameters:

hid_t `file_id`

Identifier of a file to terminate access to.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Fmount

Signature:

herr_t `H5Fmount`(*hid_t* `loc_id`, *const char* *`name`, *hid_t* `child_id`, *hid_t* `plist_id`)

Purpose:

Mounts a file.

Description:

`H5Fmount` mounts the file specified by `child_id` onto the group specified by `loc_id` and `name` using the mount properties `plist_id`.

Note that `loc_id` is either a file or group identifier and `name` is relative to `loc_id`.

Parameters:

hid_t `loc_id`

The identifier for of file or group in which `name` is defined.

const char *`name`

The name of the group onto which the file specified by `child_id` is to be mounted.

hid_t `child_id`

The identifier of the file to be mounted.

hid_t `plist_id`

The identifier of the property list to be used.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Funmount

Signature:

herr_t `H5Funmount`(*hid_t* `loc_id`, *const char* `*name`)

Purpose:

Unmounts a file.

Description:

Given a mount point, `H5Funmount` dissassociates the mount point's file from the file mounted there. This function does not close either file.

The mount point can be either the group in the parent or the root group of the mounted file (both groups have the same name). If the mount point was opened before the mount then it is the group in the parent; if it was opened after the mount then it is the root group of the child.

Note that `loc_id` is either a file or group identifier and `name` is relative to `loc_id`.

Parameters:

hid_t `loc_id`

The file or group identifier for the location at which the specified file is to be unmounted.

const char `*name`

The name of the mount point.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Freopen

Signature:

hid__t `H5Freopen`(*hid_t* `file_id`)

Purpose:

Reopens an HDF5 file.

Description: `H5Freopen` reopens an HDF5 file. The new file identifier which is returned points to the same file as the specified file idetifier, `file_id`. Both identifiers share caches and other information. The only difference between the identifiers is that the new identifier is not mounted anywhere and no files are mounted on it.

Parameters:

hid_t `file_id`

Identifier of a file to terminate access to.

Returns:

Returns a new file identifier if successful; otherwise returns a negative value.

Last modified: 20 October 1999

H5G: Group Interface

Group Object API Functions

The Group interface functions create and manipulate groups of objects in an HDF5 file.

- H5Gcreate
- H5Gopen
- H5Gclose
- H5Glink
- H5Gunlink
- H5Giterate
- H5Gmove
- H5Gget_objinfo
- H5Gget_linkval
- H5Gset_comment
- H5Gget_comment

A group associates names with objects and provides a mechanism for mapping a name to an object. Since all objects appear in at least one group (with the possible exception of the root object) and since objects can have names in more than one group, the set of all objects in an HDF5 file is a directed graph. The internal nodes (nodes with out-degree greater than zero) must be groups while the leaf nodes (nodes with out-degree zero) are either empty groups or objects of some other type. Exactly one object in every non-empty file is the root object. The root object always has a positive in-degree because it is pointed to by the file boot block.

An object name consists of one or more components separated from one another by slashes. An absolute name begins with a slash and the object is located by looking for the first component in the root object, then looking for the second component in the first object, etc., until the entire name is traversed. A relative name does not begin with a slash and the traversal begins at the location specified by the create or access function.

Name: H5Gcreate

Signature:

hid_t `H5Gcreate`(*hid_t* `loc_id`, *const char* *`name`, *size_t* `size_hint`)

Purpose:

Creates a new empty group and gives it a name.

Description:

`H5Gcreate` creates a new group with the specified name at the specified location, `loc_id`. The location is identified by a file or group identifier. The name, `name`, must not already be taken by some other object and all parent groups must already exist.

`size_hint` is a hint for the number of bytes to reserve to store the names which will be eventually added to the new group. Passing a value of zero for `size_hint` is usually adequate since the library is able to dynamically resize the name heap, but a correct hint may result in better performance. If a non-positive value is supplied for `size_hint`, then a default size is chosen.

The return value is a group identifier for the open group. This group identifier should be closed by calling `H5Gclose()` when it is no longer needed.

Parameters:

hid_t `loc_id`

The file or group identifier.

const char `*name`

The absolute or relative name of the new group.

size_t `size_hint`

An optional parameter indicating the number of bytes to reserve for the names that will appear in the group. A conservative estimate could result in multiple system-level I/O requests to read the group name heap; a liberal estimate could result in a single large I/O request even when the group has just a few names. HDF5 stores each name with a null terminator.

Returns:

Returns a valid group identifier for the open group if successful; otherwise returns a negative value.

Name: H5Gopen

Signature:

hid_t `H5Gopen`(*hid_t* `loc_id`, *const char* `*name`)

Purpose:

Opens an existing group for modification and returns a group identifier for that group.

Description:

`H5Gopen` opens an existing group with the specified name at the specified location, `loc_id`.

The location is identified by a file or group identifier

`H5Gopen` returns a group identifier for the group that was opened. This group identifier should be released by calling `H5Gclose()` when it is no longer needed.

Parameters:

hid_t `loc_id`

File or group identifier within which group is to be open.

const char `* name`

Name of group to open.

Returns:

Returns a valid group identifier if successful; otherwise returns a negative value.

Name: H5Gclose

Signature:

herr_t `H5Gclose`(*hid_t* `group_id`)

Purpose:

Closes the specified group.

Description:

`H5Gclose` releases resources used by a group which was opened by `H5Gcreate()` or `H5Gopen()`. After closing a group, the `group_id` cannot be used again.

Failure to release a group with this call will result in resource leaks.

Parameters:

hid_t `group_id`

Group identifier to release.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Glink

Signature:

herr_t `H5Glink`(*hid_t* `loc_id`, *H5G_link_t* `link_type`, *const char* *`current_name`, *const char* *`new_name`)

Purpose:

Creates a link of the specified type from `new_name` to `current_name`.

Description:

`H5Glink` creates a new name for an object that has some current name, possibly one of many names it currently has.

If `link_type` is `H5G_LINK_HARD`, then `current_name` must specify the name of an existing object and both names are interpreted relative to `loc_id`, which is either a file identifier or a group identifier.

If `link_type` is `H5G_LINK_SOFT`, then `current_name` can be anything and is interpreted at lookup time relative to the group which contains the final component of `new_name`. For instance, if `current_name` is `./foo`, `new_name` is `./x/y/bar`, and a request is made for `./x/y/bar`, then the actual object looked up is `./x/y/./foo`.

Parameters:

hid_t `loc_id`

File or group identifier.

H5G_link_t `link_type`

Link type. Possible values are `H5G_LINK_HARD` and `H5G_LINK_SOFT`.

const char * `current_name`

Name of the existing object if link is a hard link. Can be anything for the soft link.

const char * `new_name`

New name for the object.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Gunlink

Signature:

herr_t `H5Gunlink`(*hid_t* `loc_id`, *const char* *`name`)

Purpose:

Removes the specified `name` from the group graph and decrements the link count for the object to which `name` points

Description:

`H5Gunlink` removes an association between a name and an object. Object headers keep track of how many hard links refer to the object; when the hard link count reaches zero, the object can be removed from the file. Objects which are open are not removed until all identifiers to the object are closed.

If the link count reaches zero, all file-space associated with the object will be reclaimed. If the object is open, the reclamation of the file space is delayed until all handles to the object are closed.

Parameters:

hid_t `loc_id`

Identifier of the file containing the object.

const char * `name`

Name of the object to unlink.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Giterate

Signature:

int `H5Giterate`(*hid_t* `loc_id`, *const char* `*name`, *int* `*idx`, *H5G_operator_t* `operator`, *void* `*operator_data`)

Purpose:

Iterates an operation over the entries of a group.

Description:

`H5Giterate` iterates over the members of `name` in the file or group specified with `loc_id`. For each object in the group, the `operator_data` and some additional information, specified below, are passed to the `operator` function. The iteration begins with the `idx` object in the group and the next element to be processed by the operator is returned in `idx`. If `idx` is NULL, then the iterator starts at the first group member; since no stopping point is returned in this case, the iterator cannot be restarted if one of the calls to its operator returns non-zero.

The prototype for `H5G_operator_t` is:

`typedef` *herr_t* *(`H5G_operator_t`)(*hid_t* `group_id`, *const char* *`member_name`, *void* `*operator_data/*in,out*/`);

The operation receives the group identifier for the group being iterated over, `group_id`, the name of the current object within the group, `member_name`, and the pointer to the operator data passed in to `H5Giterate`, `operator_data`.

The return values from an operator are:

- Zero causes the iterator to continue, returning zero when all group members have been processed.
- Positive causes the iterator to immediately return that positive value, indicating short-circuit success. The iterator can be restarted at the next group member.
- Negative causes the iterator to immediately return that value, indicating failure. The iterator can be restarted at the next group member.

Parameters:

hid_t `loc_id`

IN: File or group identifier.

const char `*name`

IN: Group over which the iteration is performed.

int `*idx`

IN/OUT: Location at which to begin the iteration.

H5G_iterate_t `operator`

IN: Operation to be performed on an object at each step of the iteration.

void `*operator_data`

IN/OUT: Data associated with the operation.

Returns:

Returns the return value of the last operator if it was non-zero, or zero if all group members were processed. Otherwise returns a negative value.

Name: H5Gmove

Signature:

herr_t `H5Gmove`(*hid_t* `loc_id`, *const char* `*src`, *const char* `*dst`)

Purpose:

Renames an object within an HDF5 file.

Description:

`H5Gmove` renames an object within an HDF5 file. The original name, `src`, is unlinked from the group graph and the new name, `dst`, is inserted as an atomic operation. Both names are interpreted relative to `loc_id`, which is either a file or a group identifier.

Parameters:

hid_t `loc_id`

File or group identifier.

const char `*src`

Object's original name.

const char `*dst`

Object's new name.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Gget_objinfo

Signature:

herr_t `H5Gget_objinfo`(*hid_t* `loc_id`, *const char* `*name`, *hbool_t* `follow_link`, *H5G_stat_t* `*statbuf`)

Purpose:

Returns information about an object.

Description:

`H5Gget_objinfo` returns information about the specified object through the `statbuf` argument. `loc_id` (a file or group identifier) and `name` together determine the object. If the object is a symbolic link and `follow_link` is zero (`0`), then the information returned is that for the link itself; otherwise the link is followed and information is returned about the object to which the link points. If `follow_link` is non-zero but the final symbolic link is dangling (does not point to anything), then an error is returned. The `statbuf` fields are undefined for an error. The existence of an object can be tested by calling this function with a null `statbuf`.

`H5Gget_objinfo()` fills in the following data structure:

```
typedef struct H5G_stat_t {
    unsigned long fileno[2];
    unsigned long objno[2];
    unsigned nlink;
    int type;
    time_t mtime;
    size_t linklen;
} H5G_stat_t
```

The `fileno` and `objno` fields contain four values which uniquely itentify an object among those HDF5 files which are open: if all four values are the same between two objects, then the two objects are the same (provided both files are still open). The `nlink` field is the number of hard links to the object or zero when information is being returned about a symbolic link (symbolic links do not have hard links but all other objects always have at least one). The `type` field contains the type of the object, one of `H5G_GROUP`, `H5G_DATASET`, `H5G_LINK`, or `H5G_TYPE`. The `mtime` field contains the modification time. If information is being returned about a symbolic link then `linklen` will be the length of the link value (the name of the pointed-to object with the null terminator); otherwise `linklen` will be zero. Other fields may be added to this structure in the future.

Note:

Some systems will be able to record the time accurately but unable to retrieve the correct time; such systems (e.g., Irix64) will report an `mtime` value of 0 (zero).

Parameters:

hid_t `loc_id`

IN: File or group identifier.

const char `*name`

IN: Name of the object for which status is being sought.

hbool_t `follow_link`

IN: Link flag.

H5G_stat_t `*statbuf`

OUT: Buffer in which to return information about the object.

Returns:

Returns a non-negative value if successful, with the fields of `statbuf` (if non-null) initialized. Otherwise returns a negative value.

Name: H5Gget_linkval

Signature:

herr_t `H5Gget_linkval`(*hid_t* `loc_id`, *const char* *`name`, *size_t* `size`, *char* *`value`)

Purpose:

Returns the name of the object that the symbolic link points to.

Description:

`H5Gget_linkval` returns `size` characters of the name of the object that the symbolic link `name` points to.

The parameter `loc_id` is a file or group identifier.

The parameter `name` must be a symbolic link pointing to the desired object and must be defined relative to `loc_id`.

If `size` is smaller than the size of the returned object name, then the name stored in the buffer `value` will not be null terminated.

This function fails if `name` is not a symbolic link. The presence of a symbolic link can be tested by passing zero for `size` and NULL for `value`.

This function should be used only after `H5Gget_objinfo()` has been called to verify that `name` is a symbolic link.

Parameters:

hid_t `loc_id`

IN: Identifier of the file or group.

const char *`name`

IN: Symbolic link to the object whose name is to be returned.

size_t `size`

IN: Maximum number of characters of `value` to be returned.

char *`value`

OUT: A buffer to hold the name of the object being sought.

Returns:

Returns a non-negative value, with the link value in `value`, if successful. Otherwise returns a negative value.

Name: H5Gset_comment

Signature:

herr_t `H5Gset_comment`(*hid_t* `loc_id`, *const char* *`name`, *const char* *`comment`)

Purpose:

Sets comment for specified object.

Description:

`H5Gset_comment` sets the comment for the the object `name` to `comment`. Any previously existing comment is overwritten.

If `comment` is the empty string or a null pointer, the comment message is removed from the object.

Comments should be relatively short, null-terminated, ASCII strings.

Comments can be attached to any object that has an object header, e.g., data sets, groups, named data types, and data spaces, but not symbolic links.

Parameters:

hid_t `loc_id`

IN: Identifier of the file or group.

const char *`name`

IN: Name of the object whose comment is to be set or reset.

const char *`comment`

IN: The new comment.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Gget_comment

Signature:

herr_t `H5Gget_comment`(*hid_t* `loc_id`, *const char* *`name`, *size_t* `bufsize`, *char* *`comment`)

Purpose:

Retrieves comment for specified object.

Description:

`H5Gget_comment` retrieves the comment for the the object `name`. The comment is returned in the buffer `comment`.

At most `bufsize` characters, including a null terminator, are copied. The result is not null terminated if the comment is longer than the supplied buffer.

If an object does not have a comment, the empty string is returned.

Parameters:

hid_t `loc_id`

IN: Identifier of the file or group.

const char `*name`

IN: Name of the object whose comment is to be set or reset.

size_t `bufsize`

IN: Anticipated size of the buffer required to hold `comment`.

char `*comment`

OUT: The comment.

Returns:

Returns the number of characters in the comment, counting the null terminator, if successful; the value returned may be larger than `bufsize`. Otherwise returns a negative value.

Last modified: 20 October 1999

H5I: Identifier Interface

Identifier API Functions

This function provides a tool for working with object identifiers.

- H5Iget_type

Name: H5Iget_type

Signature:

H5I_type_t `H5Iget_type`(*hid_t* `obj_id`)

Purpose:

Retrieves the type of an object.

Description:

`H5Iget_type` retrieves the type of the object identified by `obj_id`.

Valid types returned by the function are

`H5I_FILE`	File
`H5I_GROUP`	Group
`H5I_DATATYPE`	Datatype
`H5I_DATASPACE`	Dataspace
`H5I_DATASET`	Dataset
`H5I_ATTR`	Attribute

If no valid type can be determined or the identifier submitted is invalid, the function returns

`H5I_BADID`	Invalid identifier

This function is of particular value in determining the type of object closing function (`H5Dclose`, `H5Gclose`, etc.) to call after a call to `H5Rdereference`.

Parameters:

hid_t `obj_id`

IN: Object identifier whose type is to be determined.

Returns:

Returns the object type if successful; otherwise `H5I_BADID`.

Last modified: 30 October 1998

H5P: Property List Interface

Property List API Functions

These functions manipulate property list objects to allow objects which require many different parameters to be easily manipulated.

General Property List Operations

- H5Pcreate
- H5Pget_class
- H5Pcopy
- H5Pclose

File Creation Properties

- H5Pget_version
- H5Pset_userblock
- H5Pget_userblock
- H5Pset_sizes
- H5Pget_sizes
- H5Pset_sym_k
- H5Pget_sym_k
- H5Pset_istore_k
- H5Pget_istore_k

Variable-length Datatype Properties

- H5Pset_vlen_mem_manager
- H5Pget_vlen_mem_manager

File Access Properties

- H5Pget_driver
- H5Pset_stdio
- H5Pget_stdio
- H5Pset_sec2
- H5Pget_sec2
- H5Pset_alignment
- H5Pget_alignment
- H5Pset_core
- H5Pget_core
- H5Pset_mpi ||
- H5Pget_mpi ||
- H5Pset_family
- H5Pget_family
- H5Pset_cache
- H5Pget_cache
- H5Pset_split
- H5Pget_split
- H5Pset_gc_references
- H5Pget_gc_references

|| *Available only in the parallel HDF5 library.*

Dataset Creation Properties

- H5Pset_layout
- H5Pget_layout
- H5Pset_chunk
- H5Pget_chunk
- H5Pset_deflate
- H5Pset_fill_value
- H5Pget_fill_value
- H5Pset_filter
- H5Pget_nfilters
- H5Pget_filter
- H5Pset_external
- H5Pget_external_count
- H5Pget_external

Dataset Memory and Transfer Properties

- H5Pset_buffer
- H5Pget_buffer
- H5Pset_preserve
- H5Pget_preserve
- H5Pset_hyper_cache
- H5Pget_hyper_cache
- H5Pset_btree_ratios
- H5Pget_btree_ratios
- H5Pset_xfer ||
- H5Pget_xfer ||

Name: H5Pcreate

Signature:

hid_t `H5Pcreate`(*H5P_class_t* `type`)

Purpose:

Creates a new property as an instance of a property list class.

Description:

`H5Pcreate` creates a new property as an instance of some property list class. The new property list is initialized with default values for the specified class. The classes are:

`H5P_FILE_CREATE`

Properties for file creation. See *Files* in the *HDF User's Guide* for details about the file creation properties.

`H5P_FILE_ACCESS`

Properties for file access. See *Files* in the *HDF User's Guide* for details about the file creation properties.

`H5P_DATASET_CREATE`

Properties for dataset creation. See *Datasets* in the *HDF User's Guide* for details about dataset creation properties.

`H5P_DATASET_XFER`

Properties for raw data transfer. See *Datasets* in the *HDF User's Guide* for details about raw data transfer properties.

`H5P_MOUNT`

Properties for file mounting. With this parameter, `H5Pcreate` creates and returns a new mount property list initialized with default values.

Parameters:

H5P_class_t `type`

IN: The type of property list to create.

Returns:

Returns a property list identifier (`plist`) if successful; otherwise Fail (-1).

Name: H5Pclose

Signature:

herr_t `H5Pclose`(*hid_t* `plist`)

Purpose:

Terminates access to a property list.

Description:

`H5Pclose` terminates access to a property list. All property lists should be closed when the application is finished accessing them. This frees resources used by the property list.

Parameters:

hid_t `plist`

IN: Identifier of the property list to terminate access to.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_class

Signature:

H5P_class_t `H5Pget_class`(*hid_t* `plist`)

Purpose:

Returns the property list class for a property list.

Description:

`H5Pget_class` returns the property list class for the property list identified by the `plist` parameter. Valid property list classes are defined in the description of `H5Pcreate()`.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

Returns:

Returns a property list class if successful. Otherwise returns H5P_NO_CLASS (-1).

Name: H5Pcopy

Signature:

hid_t `H5Pcopy`(*hid_t* `plist`)

Purpose:

Copies an existing property list to create a new property list.

Description:

`H5Pcopy` copies an existing property list to create a new property list. The new property list has the same properties and values as the original property list.

Parameters:

hid_t `plist`

IN: Identifier of property list to duplicate.

Returns:

Returns a property list identifier if successful; otherwise returns a negative value.

Name: H5Pget_version

Signature:

herr_t `H5Pget_version`(*hid_t* `plist`, *int* * `boot`, *int* * `freelist`, *int* * `stab`, *int* * `shhdr`)

Purpose:

Retrieves the version information of various objects for a file creation property list.

Description:

`H5Pget_version` retrieves the version information of various objects for a file creation property list. Any pointer parameters which are passed as NULL are not queried.

Parameters:

hid_t `plist`

IN: Identifier of the file creation property list.

int * `boot`

OUT: Pointer to location to return boot block version number.

int * `freelist`

OUT: Pointer to location to return global freelist version number.

int * `stab`

OUT: Pointer to location to return symbol table version number.

int * `shhdr`

OUT: Pointer to location to return shared object header version number.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_userblock

Signature:

herr_t `H5Pset_userblock`(*hid_t* `plist`, *hsize_t* `size`)

Purpose:

Sets user block size.

Description:

`H5Pset_userblock` sets the user block size of a file creation property list. The default user block size is 0; it may be set to any power of 2 equal to 512 or greater (512, 1024, 2048, etc.).

Parameters:

hid_t `plist`

IN: Identifier of property list to modify.

hsize_t `size`

IN: Size of the user-block in bytes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_userblock

Signature:

herr_t `H5Pget_userblock`(*hid_t* `plist`, *hsize_t* * `size`)

Purpose:

Retrieves the size of a user block.

Description:

`H5Pget_userblock` retrieves the size of a user block in a file creation property list.

Parameters:

hid_t `plist`

IN: Identifier for property list to query.

hsize_t * `size`

OUT: Pointer to location to return user-block size.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_sizes

Signature:

herr_t `H5Pset_sizes`(*hid_t* `plist`, *size_t* `sizeof_addr`, *size_t* `sizeof_size`)

Purpose:

Sets the byte size of the offsets and lengths used to address objects in an HDF5 file.

Description:

`H5Pset_sizes` sets the byte size of the offsets and lengths used to address objects in an HDF5 file. This function is only valid for file creation property lists. Passing in a value of 0 for one of the sizeof parameters retains the current value. The default value for both values is 4 bytes. Valid values currently are 2, 4, 8 and 16.

Parameters:

hid_t `plist`

IN: Identifier of property list to modify.

size_t `sizeof_addr`

IN: Size of an object offset in bytes.

size_t `sizeof_size`

IN: Size of an object length in bytes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_sizes

Signature:

herr_t `H5Pget_sizes`(*hid_t* `plist`, *size_t* * `sizeof_addr`, *size_t* * `sizeof_size`)

Purpose:

Retrieves the size of the offsets and lengths used in an HDF5 file.

Description:

`H5Pget_sizes` retrieves the size of the offsets and lengths used in an HDF5 file. This function is only valid for file creation property lists.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

size_t * `size`

OUT: Pointer to location to return offset size in bytes.

size_t * `size`

OUT: Pointer to location to return length size in bytes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_mpi

Signature:

herr_t `H5Pset_mpi`(*hid_t* `plist`, *MPI_Comm* `comm`, *MPI_Info* `info`)

Purpose:

Retrieves the access mode for parallel I/O and the user supplied communicator and info object.

Description:

`H5Pset_mpi` stores the access mode for MPIO call and the user supplied communicator and info in the access property list, which can then be used to open file. This function is available only in the parallel HDF5 library and is not a collective function.

Parameters:

hid_t `plist`

IN: Identifier of property list to modify

MPI_Comm `comm`

IN: MPI communicator to be used for file open as defined in MPI_FILE_OPEN of MPI-2. This function does not make a duplicated `comm`. Any modification to `comm` after this function call returns may have undetermined effect to the access property list. Users should call this function again to setup the property list.

MPI_Info `info`

IN: MPI info object to be used for file open as defined in MPI_FILE_OPEN of MPI-2. This function does not make a duplicated `info`. Any modification to `info` after this function call returns may have undetermined effect to the access property list. Users should call this function again to setup the property list.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_mpi

Signature:

herr_t `H5Pget_mpi`(*hid_t* `plist`, *MPI_Comm* `*comm`, *MPI_Info* `*info`)

Purpose:

Retrieves the communicator and info object.

Description:

`H5Pget_mpi` retrieves the communicator and info object that have been set by H5Pset_mpi. This function is available only in the parallel HDF5 library and is not a collective function.

Parameters:

hid_t `plist`

IN: Identifier of a file access property list that has been set successfully by H5Pset_mpi.

MPI_Comm * `comm`

OUT: Pointer to location to return the communicator.

MPI_Info * `info`

OUT: Pointer to location to return the info object.

Returns:

Returns a non-negative value if the file access property list is set to the MPI. Otherwise returns a negative value.

Name: H5Pset_xfer

Signature:

herr_t `H5Pset_xfer`(*hid_t* `plist`, *H5D_transfer_t* `data_xfer_mode`)

Purpose:

Sets the transfer mode of the dataset transfer property list.

Description:

`H5Pset_xfer` sets the transfer mode of the dataset transfer property list. The list can then be used to control the I/O transfer mode during dataset accesses. This function is available only in the parallel HDF5 library and is not a collective function.

Valid data transfer modes are:

H5D_XFER_INDEPENDENT

Use independent I/O access. (Currently the default mode.)

H5D_XFER_COLLECTIVE

Use MPI collective I/O access.

H5D_XFER_DFLT

User default I/O access.

Parameters:

hid_t `plist`

IN: Identifier of a dataset transfer property list

H5D_transfer_t `data_xfer_mode`

IN: Data transfer mode.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_xfer

Signature:

herr_t `H5Pget_xfer`(*hid_t* `plist`, *H5D_transfer_t* * `data_xfer_mode`)

Purpose:

Retrieves the transfer mode from the dataset transfer property list.

Description:

`H5Pget_xfer` retrieves the transfer mode from the dataset transfer property list. This function is available only in the parallel HDF5 library and is not a collective function.

Parameters:

hid_t `plist`

IN: Identifier of a dataset transfer property list.

H5D_transfer_t * `data_xfer_mode`

OUT: Pointer to location to return the data transfer mode.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_sym_k

Signature:

herr_t `H5Pset_sym_k`(*hid_t* `plist`, *int* `ik`, *int* `lk`)

Purpose:

Sets the size of parameters used to control the symbol table nodes.

Description:

`H5Pset_sym_k` sets the size of parameters used to control the symbol table nodes. This function is only valid for file creation property lists. Passing in a value of 0 for one of the parameters retains the current value.

`ik` is one half the rank of a tree that stores a symbol table for a group. Internal nodes of the symbol table are on average 75% full. That is, the average rank of the tree is 1.5 times the value of `ik`.

`lk` is one half of the number of symbols that can be stored in a symbol table node. A symbol table node is the leaf of a symbol table tree which is used to store a group. When symbols are inserted randomly into a group, the group's symbol table nodes are 75% full on average. That is, they contain 1.5 times the number of symbols specified by `lk`.

Parameters:

hid_t `plist`

IN: Identifier for property list to query.

int `ik`

IN: Symbol table tree rank.

int `lk`

IN: Symbol table node size.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_sym_k

Signature:

herr_t `H5Pget_sym_k`(*hid_t* `plist`, *int* * `ik`, *int* * `lk`)

Purpose:

Retrieves the size of the symbol table B-tree 1/2 rank and the symbol table leaf node 1/2 size.

Description:

`H5Pget_sym_k` retrieves the size of the symbol table B-tree 1/2 rank and the symbol table leaf node 1/2 size. This function is only valid for file creation property lists. If a parameter valued is set to NULL, that parameter is not retrieved. See the description for `H5Pset_sym_k` for more information.

Parameters:

hid_t `plist`

IN: Property list to query.

int * `ik`

OUT: Pointer to location to return the symbol table's B-tree 1/2 rank.

int * `size`

OUT: Pointer to location to return the symbol table's leaf node 1/2 size.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_istore_k

Signature:

herr_t `H5Pset_istore_k`(*hid_t* `plist`, *int* `ik`)

Purpose:

Sets the size of the parameter used to control the B-trees for indexing chunked datasets.

Description:

`H5Pset_istore_k` sets the size of the parameter used to control the B-trees for indexing chunked datasets. This function is only valid for file creation property lists. Passing in a value of 0 for one of the parameters retains the current value.

`ik` is one half the rank of a tree that stores chunked raw data. On average, such a tree will be 75% full, or have an average rank of 1.5 times the value of `ik`.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

int `ik`

IN: 1/2 rank of chunked storage B-tree.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_istore_k

Signature:

herr_t `H5Pget_istore_k`(*hid_t* `plist`, *int* * `ik`)

Purpose:

Queries the 1/2 rank of an indexed storage B-tree.

Description:

`H5Pget_istore_k` queries the 1/2 rank of an indexed storage B-tree. The argument `ik` may be the null pointer (NULL). This function is only valid for file creation property lists.

See `H5Pset_istore_k` for details.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

int * `ik`

OUT: Pointer to location to return the chunked storage B-tree 1/2 rank.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_layout

Signature:

herr_t `H5Pset_layout`(*hid_t* `plist`, *H5D_layout_t* `layout`)

Purpose:

Sets the type of storage used store the raw data for a dataset.

Description:

`H5Pset_layout` sets the type of storage used store the raw data for a dataset. This function is only valid for dataset creation property lists. Valid parameters for `layout` are:

H5D_COMPACT ***(Not yet implemented.)***

Store raw data and object header contiguously in file. This should only be used for very small amounts of raw data (suggested less than 1KB).

H5D_CONTIGUOUS

Store raw data separately from object header in one large chunk in the file.

H5D_CHUNKED

Store raw data separately from object header in one large chunk in the file and store chunks of the raw data in separate locations in the file.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

H5D_layout_t `layout`

IN: Type of storage layout for raw data.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_layout

Signature:

H5D_layout_t `H5Pget_layout`(*hid_t* `plist`)

Purpose:

Returns the layout of the raw data for a dataset.

Description:

`H5Pget_layout` returns the layout of the raw data for a dataset. This function is only valid for dataset creation property lists. Valid types for `layout` are:

H5D_COMPACT ***(Not yet implemented.)***

Raw data and object header stored contiguously in file.

H5D_CONTIGUOUS

Raw data stored separately from object header in one large chunk in the file.

H5D_CHUNKED

Raw data stored separately from object header in chunks in separate locations in the file.

Parameters:

hid_t `plist`

IN: Identifier for property list to query.

Returns:

Returns the layout type of a a dataset creation property list if successful. Otherwise returns H5D_LAYOUT_ERROR (-1).

Name: H5Pset_chunk

Signature:

herr_t `H5Pset_chunk`(*hid_t* `plist`, *int* `ndims`, *const hsize_t* * `dim`)

Purpose:

Sets the size of the chunks used to store a chunked layout dataset.

Description:

`H5Pset_chunk` sets the size of the chunks used to store a chunked layout dataset. This function is only valid for dataset creation property lists. The `ndims` parameter currently must be the same size as the rank of the dataset. The values of the `dim` array define the size of the chunks to store the dataset's raw data. As a side-effect, the layout of the dataset is changed to `H5D_CHUNKED`, if it is not already.

Parameters:

hid_t `plist`

IN: Identifier for property list to query.

int `ndims`

IN: The number of dimensions of each chunk.

const hsize_t * `dim`

IN: An array containing the size of each chunk.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_chunk

Signature:

int `H5Pget_chunk`(*hid_t* `plist`, *int* `max_ndims`, *hsize_t* * `dims`)

Purpose:

Retrieves the size of chunks for the raw data of a chunked layout dataset.

Description:

`H5Pget_chunk` retrieves the size of chunks for the raw data of a chunked layout dataset. This function is only valid for dataset creation property lists. At most, `max_ndims` elements of `dims` will be initialized.

Parameters:

hid_t `plist`

IN: Identifier of property list to query.

int `max_ndims`

OUT: Size of the `dims` array.

hsize_t * `dims`

OUT: Array to store the chunk dimensions.

Returns:

Returns chunk dimensionality successful; otherwise returns a negative value.

Name: H5Pset_alignment

Signature:

herr_t `H5Pset_alignment`(*hid_t* `plist`, *hsize_t* `threshold`, *hsize_t* `alignment`)

Purpose:

Sets alignment properties of a file access property list.

Description:

`H5Pset_alignment` sets the alignment properties of a file access property list so that any file object = THRESHOLD bytes will be aligned on an address which is a multiple of ALIGNMENT. The addresses are relative to the end of the user block; the alignment is calculated by subtracting the user block size from the absolute file address and then adjusting the address to be a multiple of ALIGNMENT.

Default values for THRESHOLD and ALIGNMENT are one, implying no alignment. Generally the default values will result in the best performance for single-process access to the file. For MPI-IO and other parallel systems, choose an alignment which is a multiple of the disk block size.

Parameters:

hid_t `plist`

IN: Identifier for a file access property list.

hsize_t `threshold`

IN: Threshold value.

hsize_t `alignment`

IN: Alignment value.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_alignment

Signature:

herr_t `H5Pget_alignment`(*hid_t* `plist`, *hsize_t* `*threshold`, *hsize_t* `*alignment`)

Purpose:

Retrieves the current settings for alignment properties from a file access property list.

Description:

`H5Pget_alignment` retrieves the current settings for alignment properties from a file access property list. The `threshold` and/or `alignment` pointers may be null pointers (NULL).

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

hsize_t `*threshold`

OUT: Pointer to location of return threshold value.

hsize_t `*alignment`

OUT: Pointer to location of return alignment value.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_external

Signature:

herr_t `H5Pset_external`(*hid_t* `plist`, *const char* `*name`, *off_t* `offset`, *hsize_t* `size`)

Purpose:

Adds an external file to the list of external files.

Description:

`H5Pset_external` adds an external file to the list of external files.

If a dataset is split across multiple files then the files should be defined in order. The total size of the dataset is the sum of the `size` arguments for all the external files. If the total size is larger than the size of a dataset then the dataset can be extended (provided the data space also allows the extending).

The `size` argument specifies number of bytes reserved for data in the external file. If `size` is set to `H5F_UNLIMITED`, the external file can be of unlimited size and no more files can be added to the external files list.

Parameters:

hid_t `plist`

IN: Identifier of a dataset creation property list.

const char `*name`

IN: Name of an external file.

off_t `offset`

IN: Offset, in bytes, from the beginning of the file to the location in the file where the data starts.

hsize_t `size`

IN: Number of bytes reserved in the file for the data.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_external_count

Signature:

int `H5Pget_external_count`(*hid_t* `plist`,)

Purpose:

Returns the number of external files for a dataset.

Description:

`H5Pget_external_count` returns the number of external files for the specified dataset.

Parameters:

hid_t `plist`

IN: Identifier of a dataset creation property list.

Returns:

Returns the number of external files if successful; otherwise returns a negative value.

Name: H5Pget_external

Signature:

herr_t `H5Pget_external`(*hid_t* `plist`, *int* `idx`, *size_t* `name_size`, *char* `*name`, *off_t* `*offset`, *hsize_t* `*size`)

Purpose:

Returns information about an external file.

Description:

`H5Pget_external` returns information about an external file. The external file is specified by its index, `idx`, which is a number from zero to N-1, where N is the value returned by `H5Pget_external_count()`. At most `name_size` characters are copied into the `name` array. If the external file name is longer than `name_size` with the null terminator, the return value is not null terminated (similar to `strncpy()`).

If `name_size` is zero or `name` is the null pointer, the external file name is not returned. If `offset` or `size` are null pointers then the corresponding information is not returned.

Parameters:

hid_t `plist`

IN: Identifier of a dataset creation property list.

int `idx`

IN: External file index.

size_t `name_size`

IN: Maximum length of `name` array.

char `*name`

OUT: Name of the external file.

off_t `*offset`

OUT: Pointer to a location to return an offset value.

hsize_t `*size`

OUT: Pointer to a location to return the size of the external file data.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_fill_value

Signature:

herr_t `H5Pset_fill_value`(*hid_t* `plist_id`, *hid_t* `type_id`, *const void* *`value`)

Purpose:

Sets a dataset fill value.

Description:

`H5Pset_fill_value` sets the fill value for a dataset creation property list.

The `value` is interpreted as being of type `type_id`. This need not be the same type as the dataset, but the library must be able to convert `value` to the dataset type when the dataset is created.

Notes:

If a fill value is set for a dataset (even if the fill value is all zeros), the fill value will be written to the file. If no fill value is set, then HDF5 relies on the underlying file driver (usually a Unix file system) to initialize unwritten parts of the file to zeros.

Creating a contiguous dataset with a fill value can be a very expensive operation since the optimization has not yet been implemented that would delay the writing of the fill values until after some data has been written.

Parameters:

hid_t `plist_id`

IN: Property list identifier.

hid_t `type_id`,

IN: The datatype identifier of `value`.

const void *`value`

IN: The fill value.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_fill_value

Signature:

herr_t `H5Pget_fill_value`(*hid_t* `plist_id`, *hid_t* `type_id`, *void* *`value`)

Purpose:

Retrieves a dataset fill value.

Description:

`H5Pget_fill_value` queries the fill value property of a dataset creation property list.

The fill value is returned through the `value` pointer.

Memory is allocated by the caller.

The fill value will be converted from its current data type to the type specified by `type_id`.

Parameters:

hid_t `plist_id`

IN: Property list identifier.

hid_t `type_id`,

IN: The datatype identifier of `value`.

const void *`value`

IN: The fill value.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_filter

Signature:

herr_t `H5Pset_filter`(*hid_t* `plist`, *H5Z_filter_t* `filter`, *unsigned int* `flags`, *size_t* `cd_nelmts`, *const unsigned int* `cd_values[]`)

Purpose:

Adds a filter to the filter pipeline.

Description:

`H5Pset_filter` adds the specified `filter` and corresponding properties to the end of an output filter pipeline. If `plist` is a dataset creation property list, the filter is added to the permanent filter pipeline; if `plist` is a dataset transfer property list, the filter is added to the transient filter pipeline.

The array `cd_values` contains `cd_nelmts` integers which are auxiliary data for the filter. The integer values will be stored in the dataset object header as part of the filter information.

The `flags` argument is a bit vector with the following fields specifying certain general properties of the filter:

`H5Z_FLAG_OPTIONAL`	If this bit is set then the filter is optional. If the filter fails (see below) during an `H5Dwrite()` operation then the filter is just excluded from the pipeline for the chunk for which it failed; the filter will not participate in the pipeline during an `H5Dread()` of the chunk. This is commonly used for compression filters: if the compression result would be larger than the input then the compression filter returns failure and the uncompressed data is stored in the file. If this bit is clear and a filter fails then `H5Dwrite()` or `H5Dread()` also fails.

Note: This function currently supports only the permanent filter pipeline; `plist_id` must be a dataset creation property list.

Parameters:

hid_t `plist`

IN: Property list identifier.

H5Z_filter_t `filter`

IN: Filter to be added to the pipeline.

unsigned int `flags`

IN: Bit vector specifying certain general properties of the filter.

size_t `cd_nelmts`

IN: Number of elements in `cd_values`

const unsigned int `cd_values[]`

IN: Auxiliary data for the filter.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_nfilters

Signature:

int `H5Pget_nfilters`(*hid_t* `plist`)

Purpose:

Returns the number of filters in the pipeline.

Description:

`H5Pget_nfilters` returns the number of filters defined in the filter pipeline associated with the property list `plist`.

In each pipeline, the filters are numbered from 0 through *N*-1, where *N* is the value returned by this function. During output to the file, the filters are applied in increasing order; during input from the file, they are applied in decreasing order.

`H5Pget_nfilters` returns the number of filters in the pipeline, including zero (`0`) if there are none.

Note:

This function currently supports only the permanent filter pipeline; `plist_id` must be a dataset creation property list.

Parameters:

hid_t `plist`

IN: Property list identifier.

Returns:

Returns the number of filters in the pipeline if successful; otherwise returns a negative value.

Name: H5Pget_filter

Signature:

H5Z_filter_t `H5Pget_filter`(*hid_t* `plist`, *int* `filter_number`, *unsigned int* *`flags`, *size_t* *`cd_nelmts`, *unsigned int* *`cd_values`, *size_t* `namelen`, *char* `name[]`)

Purpose:

Returns information about a filter in a pipeline.

Description:

`H5Pget_filter` returns information about a filter, specified by its filter number, in a filter pipeline, specified by the property list with which it is associated.

If `plist` is a dataset creation property list, the pipeline is a permanent filter pipeline; if `plist` is a dataset transfer property list, the pipeline is a transient filter pipeline.

On input, `cd_nelmts` indicates the number of entries in the `cd_values` array, as allocated by the caller; on return,`cd_nelmts` contains the number of values defined by the filter.

`filter_number` is a value between zero and *N*-1, as described in `H5Pget_nfilters()`. The function will return a negative value if the filter number is out of range.

If `name` is a pointer to an array of at least `namelen` bytes, the filter name will be copied into that array. The name will be null terminated if `namelen` is large enough. The filter name returned will be the name appearing in the file, the name registered for the filter, or an empty string.

The structure of the `flags` argument is discussed in `H5Pset_filter()`.

Note:

This function currently supports only the permanent filter pipeline; `plist` must be a dataset creation property list.

Parameters:

hid_t `plist`

IN: Property list identifier.

int `filter_number`

IN: Sequence number within the filter pipeline of the filter for which information is sought.

unsigned int *`flags`

OUT: Bit vector specifying certain general properties of the filter.

size_t *`cd_nelmts`

IN/OUT: Number of elements in `cd_values`

unsigned int *`cd_values`

OUT: Auxiliary data for the filter.

size_t `namelen`

IN: Anticipated number of characters in `name`.

char `name[]`

OUT: Name of the filter.

Returns:

Returns the filter identification number if successful. Otherwise returns H5Z_FILTER_ERROR (-1).

Name: H5Pget_driver

Signature:

H5F_driver_t `H5Pget_driver`(*hid_t* `plist`,)

Purpose:

Returns a low-level file driver identifier.

Description:

`H5Pget_driver` returns the identifier of the low-level file driver. Valid identifiers are:

- H5F_LOW_STDIO (0)
- H5F_LOW_SEC2 (1)
- H5F_LOW_MPIO (2)
- H5F_LOW_CORE (3)

- H5F_LOW_SPLIT (4)
- H5F_LOW_FAMILY (5)

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

Returns:

Returns a low-level driver identifier if successful. Otherwise returns H5F_LOW_ERROR (-1).

Name: H5Pset_stdio

Signature:

herr_t `H5Pset_stdio`(*hid_t* `plist`)

Purpose:

Sets the low level file driver to use the functions declared in the stdio.h.

Description:

`H5Pset_stdio` sets the low level file driver to use the functions declared in the stdio.h file: fopen(), fseek() or fseek64(), fread(), fwrite(), and fclose().

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_stdio

Signature:

herr_t `H5Pget_stdio`(*hid_t* `plist`)

Purpose:

Determines whether the file access property list is set to the stdio driver.

Description:

`H5Pget_stdio` checks to determine whether the file access property list is set to the stdio driver. In the future, additional arguments may be added to this function to match those added to H5Pset_stdio().

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

Returns:

Returns a non-negative value if the file access property list is set to the stdio driver. Otherwise returns a negative value.

Name: H5Pset_sec2

Signature:

herr_t `H5Pset_sec2`(*hid_t* `plist`,)

Purpose:

Sets the low-level file driver to use the declared functions.

Description:

`H5Pset_sec2` sets the low-level file driver to use the functions declared in the unistd.h file: open(), lseek() or lseek64(), read(), write(), and close().

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_sec2

Signature:

returntype `H5Pget_sec2`(*hid_t* `plist`)

Purpose:

Checks whether the file access property list is set to the sec2 driver.

Description:

`H5Pget_sec2` checks to determine whether the file access property list is set to the sec2 driver. In the future, additional arguments may be added to this function to match those added to H5Pset_sec2().

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

Returns:

Returns a non-negative value if the file access property list is set to the sec2 driver. Otherwise returns a negative value.

Name: H5Pset_core

Signature:

herr_t `H5Pset_core`(*hid_t* `plist`, *size_t* `increment`)

Purpose:

Sets the low-level file driver to use malloc() and free().

Description:

`H5Pset_core` sets the low-level file driver to use `malloc()` and `free()`. This driver is restricted to temporary files which are not larger than the amount of virtual memory available. The `increment` argument determines the file block size and memory will be allocated in multiples of INCREMENT bytes. A liberal `increment` results in fewer calls to `realloc()` and probably less memory fragmentation.

Parameters:

hid_t `plist`

IN: Identifier of a file access property list.

size_t `increment`

IN: File block size in bytes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_core

Signature:

herr_t `H5Pget_core`(*hid_t* `plist`, *size_t* `*increment`)

Purpose:

Determines whether the file access property list is set to the core driver.

Description:

`H5Pget_core` checks to determine whether the file access property list is set to the core driver. On success, the block size is returned through the `increment` if it is not the null pointer. In the future, additional arguments may be added to this function to match those added to `H5Pset_core()`.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

size_t `*increment`

OUT: Pointer to a location to return the file block size (in bytes).

Returns:

Returns a non-negative value if the file access property list is set to the core driver. Otherwise returns a negative value.

Name: H5Pset_split

Signature:

herr_t `H5Pset_split`(*hid_t* `plist`, *const char* `*meta_ext`, *hid_t* `meta_plist`, *const char* `*raw_ext`, *hid_t* `raw_plist`)

Purpose:

Sets the low-level driver to split meta data from raw data.

Description:

`H5Pset_split` sets the low-level driver to split meta data from raw data, storing meta data in one file and raw data in another file. The meta file will have a name which is formed by adding *meta_extension* (recommended default value: `.meta`) to the end of the base name and will be accessed according to the *meta_properties*. The raw file will have a name which is formed by appending *raw_extension* (recommended default value: `.raw`) to the base name and will be accessed according to the *raw_properties*. Additional parameters may be added to this function in the future.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

const char `*meta_ext`

IN: Name of the extension for the metafile filename. Recommended default value: `.meta`.

hid_t `meta_plist`

IN: Identifier of the meta file access property list.

const char `*raw_ext`

IN: Name extension for the raw file filename. Recommended default value: `.raw`.

hid_t `raw_plist`

IN: Identifier of the raw file access property list.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_split

Signature:

herr_t `H5Pget_split`(*hid_t* `plist`, *size_t* `meta_ext_size`, *char* `*meta_ext`, *hid_t* `*meta_properties`, *size_t* `raw_ext_size`, *char* `*raw_ext`, *hid_t* `*raw_properties`)

Purpose:

Determines whether the file access property list is set to the split driver.

Description:

`H5Pget_split` checks to determine whether the file access property list is set to the split driver. On successful return, `meta_properties` and `raw_properties` will point to copies of the meta and raw access property lists which should be closed by calling `H5Pclose()` when the application is finished with them, but if the meta and/or raw file has no property list then a negative value is returned for that property list identifier. Also, if `meta_extension` and/or `raw_extension` are non-null pointers, at most `meta_ext_size` or `raw_ext_size` characters of the meta or raw file name extension will be copied to the specified buffer. If the actual name is longer than what was requested then the result will not be null terminated (similar to `strncpy()`). In the future, additional arguments may be added to this function to match those added to `H5Pset_split()`.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

size_t `meta_ext_size`

IN: Number of characters of the meta file extension to be copied to the `meta_ext` buffer.

char `*meta_ext`

OUT: Meta file extension.

hid_t `*meta_properties`

OUT: Pointer to a copy of the meta file access property list.

size_t `raw_ext_size`

IN: Number of characters of the raw file extension to be copied to the `raw_ext` buffer.

char `*raw_ext`

OUT: Raw file extension.

hid_t `*raw_properties`

OUT: Pointer to a copy of the raw file access property list.

Returns:

Returns a non-negative value if the file access property list is set to the split driver. Otherwise returns a negative value.

Name: H5Pset_gc_references

Signature:

herr_t `H5Pset_split`(*hid_t* `plist`, *unsigned* `gc_ref`)

Purpose:

Sets garbage collecting references flag.

Description:

`H5Pset_gc_references` sets the flag for garbage collecting references for the file.

Dataset region references and other reference types use space in an HDF5 file's global heap. If garbage collection is on and the user passes in an uninitialized value in a reference structure, the heap might get corrupted. When garbage collection is off, however, and the user re-uses a reference, the previous heap block will be orphaned and not returned to the free heap space.

When garbage collection is on, the user must initialize the reference structures to 0 or risk heap corruption.

The default value for garbage collecting references is off.

Parameters:

hid_t `plist`

IN: File access property list identifier.

unsigned `gc_ref`

IN:

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_gc_references

Signature:

herr_t `H5Pget_split`(*hid_t* `plist`, *unsigned* *`gc_ref`)

Purpose:

Returns garbage collecting references setting.

Description:

`H5Pget_gc_references` returns the current setting for the garbage collection references property from the specified file access property list. The garbage collection references property is set by H5Pset_gc_references.

Parameters:

hid_t `plist`

IN: File access property list identifier.

unsigned `gc_ref`

OUT:

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_family

Signature:

herr_t `H5Pset_family`(*hid_t* `plist`, *hsize_t* `memb_size`, *hid_t* `memb_plist`)

Purpose:

Sets the file access properties list to the `family` driver.

Description:

`H5Pset_family` sets the file access properties to use the `family` driver; any previously defined driver properties are erased from the property list. See "File Families" in the *HDF5 Files* section of the *HDF5 User's Guide*.

Each member of the file family will use `memb_plist` as its file access property list.

The `memb_size` argument gives the logical size in bytes of each family member; the actual size could be smaller depending on whether the file contains holes. The member size is only used when creating a new file or truncating an existing file; otherwise the member size comes from the size of the first member of the family being opened.

Note: If the size of the `off_t` type is four bytes, then the maximum family member size is usually 2^31-1 because the byte at offset 2,147,483,647 is generally inaccessible.

Additional parameters may be added to this function in the future.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

hsize_t `memb_size`

IN: Logical size, in bytes, of each family member.

hid_t `memb_plist`

IN: Identifier of the file access property list for each member of the family.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_family

Signature:

herr_t `H5Pget_family`(*hid_t* `tid`, *hsize_t* `*memb_size`, *hid_t* `*memb_plist`)

Purpose:

Determines whether the file access property list is set to the family driver.

Description:

`H5Pget_family` checks to determine whether the file access property list is set to the family driver. On successful return, *access_properties* will point to a copy of the member access property list which should be closed by calling `H5Pclose()` when the application is finished with it. If *memb_size* is non-null then it will contain the logical size in bytes of each family member. In the future, additional arguments may be added to this function to match those added to `H5Pset_family()`.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

hsize_t `*memb_size`

OUT: Logical size, in bytes, of each family member.

hid_t `*memb_plist`

OUT: Identifier of the file access property list for each member of the family.

Returns:

Returns a non-negative value if the file access property list is set to the family driver. Otherwise returns a negative value.

Name: H5Pset_cache

Signature:

herr_t `H5Pset_cache`(*hid_t* `plist`, *int* `mdc_nelmts`, *size_t* `rdcc_nbytes`, *double* `rdcc_w0`)

Purpose:

Sets the number of elements in the meta data cache and the total number of bytes in the raw data chunk cache.

Description:

`H5Pset_cache` sets the number of elements (objects) in the meta data cache and the total number of bytes in the raw data chunk cache.

Sets or queries the meta data cache and raw data chunk cache parameters. The *plist* is a file access property list. The number of elements (objects) in the meta data cache is *mdc_nelmts*. The total size of the raw data chunk cache and the preemption policy is *rdcc_nbytes* and *w0*. For `H5Pget_cache()` any (or all) of the pointer arguments may be null pointers.

The RDCC_W0 value should be between 0 and 1 inclusive and indicates how much chunks that have been fully read are favored for preemption. A value of zero means fully read chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of one means fully read chunks are always preempted before other chunks.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

int `mdc_nelmts`

IN: Number of elements (objects) in the meta data cache.

size_t `rdcc_nbytes`

IN: Total size of the raw data chunk cache, in bytes.

double `rdcc_w0`

IN: Preemption policy.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_cache

Signature:

herr_t `H5Pget_cache`(*hid_t* `plist`, *int* `*mdc_nelmts`, *size_t* `*rdcc_nbytes`, *double* `*rdcc_w0`)

Purpose:

Retrieves maximum sizes of meta data cache and RDCC_WO.

Description:

Retrieves the maximum possible number of elements in the meta data cache and the maximum possible number of bytes and the RDCC_W0 value in the raw data chunk cache. Any (or all) arguments may be null pointers in which case the corresponding datum is not returned.

Parameters:

hid_t `plist`

IN: Identifier of the file access property list.

int `*mdc_nelmts`

IN/OUT: Number of elements (objects) in the meta data cache.

size_t `*rdcc_nbytes`

IN/OUT: Total size of the raw data chunk cache, in bytes.

double `*rdcc_w0`

IN/OUT: Preemption policy.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_hyper_cache

Signature:

herr_t `H5Pset_hyper_cache`(*hid_t* `plist`, *unsigned* `cache`, *unsigned* `limit`)

Purpose:

Indicates whether to cache hyperslab blocks during I/O.

Description:

Given a dataset transfer property list, `H5Pset_hyper_cache` indicates whether to cache hyperslab blocks during I/O, a process which can significantly increase I/O speeds.

The parameter `limit` sets the maximum size of the hyperslab block to cache. If a block is smaller than that limit, it may still not be cached if no memory is available. Setting the limit to `0` (zero) indicates no limitation on the size of block to attempt to cache.

The default is to cache blocks with no limit on block size for serial I/O and to not cache blocks for parallel I/O.

Parameters:

hid_t `plist`

IN: Dataset transfer property list identifier.

unsigned `cache`

IN:

unsigned `limit`

IN: Maximum size of the hyperslab block to cache. `0` (zero) indicates no limit.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_hyper_cache

Signature:

herr_t `H5Pget_hyper_cache`(*hid_t* `plist`, *unsigned* `cache`, *unsigned* `limit`)

Purpose:

Returns information regarding the caching of hyperslab blocks during I/O.

Description:

Given a dataset transfer property list, `H5Pget_hyper_cache` returns instructions regarding the caching of hyperslab blocks during I/O. These parameters are set with the `H5Pset_hyper_cache` function.

Parameters:

hid_t `plist`

IN: Dataset transfer property list identifier.

unsigned `cache`

OUT:

unsigned `limit`

OUT: Maximum size of the hyperslab block to cache. `0` (zero) indicates no limit.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_btree_ratios

Signature:

herr_t `H5Pset_btree_ratios`(*hid_t* `plist`, *double* `left`, *double* `middle`, *double* `right`)

Purpose:

Sets B-tree split ratios for a dataset transfer property list.

Description:

`H5Pset_btree_ratios` sets the B-tree split ratios for a dataset transfer property list. The split ratios determine what percent of children go in the first node when a node splits.

The ratio `left` is used when the splitting node is the left-most node at its level in the tree; the ratio `right` is used when the splitting node is the right-most node at its level; and the ratio `middle` is used for all other cases.

A node which is the only node at its level in the tree uses the ratio `right` when it splits.

All ratios are real numbers between 0 and 1, inclusive.

Parameters:

hid_t `plist`

IN: The dataset transfer property list identifier.

double `left`

IN: The B-tree split ratio for left-most nodes.

double `right`

IN: The B-tree split ratio for right-most nodes and lone nodes.

double `middle`

IN: The B-tree split ratio for all other nodes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_btree_ratios

Signature:

herr_t `H5Pget_btree_ratios`(*hid_t* `plist`, *double* `*left`, *double* `*middle`, *double* `*right`)

Purpose:

Gets B-tree split ratios for a dataset transfer property list.

Description:

`H5Pget_btree_ratios` returns the B-tree split ratios for a dataset transfer property list.

The B-tree split ratios are returned through the non-`NULL` arguments `left`, `middle`, and `right`, as set by the `H5Pset_btree_ratios` function.

Parameters:

hid_t `plist`

IN: The dataset transfer property list identifier.

double `left`

OUT: The B-tree split ratio for left-most nodes.

double `right`

OUT: The B-tree split ratio for right-most nodes and lone nodes.

double `middle`

OUT: The B-tree split ratio for all other nodes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_buffer

Signature:

herr_t `H5Pset_buffer`(*hid_t* `plist`, *size_t* `size`, *void* `*tconv`, *void* `*bkg`)

Purpose:

Sets type conversion and background buffers.

Description:

Given a dataset transfer property list, `H5Pset_buffer` sets the maximum size for the type conversion buffer and background buffer and optionally supplies pointers to application-allocated buffers. If the buffer size is smaller than the entire amount of data being transferred between the application and the file, and a type conversion buffer or background buffer is required, then strip mining will be used.

Note that there are minimum size requirements for the buffer. Strip mining can only break the data up along the first dimension, so the buffer must be large enough to accommodate a complete slice that encompasses all of the remaining dimensions. For example, when strip mining a 100x200x300 hyperslab of a simple data space, the buffer must be large enough to hold 1x200x300 data elements. When strip mining a 100x200x300x150 hyperslab of a simple data space, the buffer must be large enough to hold 1x200x300x150 data elements.

If `tconv` and/or `bkg` are null pointers, then buffers will be allocated and freed during the data transfer.

The default value for the maximum buffer is 1 Mb.

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

size_t `size`

IN: Size for the type conversion and background buffers.

void `tconv`

IN: Pointer to application-allocated type conversion buffer.

void `bkg`

IN: Pointer to application-allocated background buffer.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_buffer

Signature:

size_t `H5Pget_buffer`(*hid_t* `plist`, *void* `**tconv`, *void* `**bkg`)

Purpose:

Reads buffer settings.

Description:

`H5Pget_buffer` reads values previously set with H5Pset_buffer().

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

void `**tconv`

OUT: Address of the pointer to application-allocated type conversion buffer.

void `**bkg`

OUT: Address of the pointer to application-allocated background buffer.

Returns:

Returns buffer size if successful; otherwise 0 on failure.

Name: H5Pset_preserve

Signature:

herr_t `H5Pset_preserve`(*hid_t* `plist`, *hbool_t* `status`)

Purpose:

Sets the dataset transfer property list status to TRUE or FALSE.

Description:

`H5Pset_preserve` sets the dataset transfer property list status to TRUE or FALSE.

When reading or writing compound data types and the destination is partially initialized and the read/write is intended to initialize the other members, one must set this property to TRUE. Otherwise the I/O pipeline treats the destination datapoints as completely uninitialized.

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

hbool_t `status`

IN: Status of for the dataset transfer property list (TRUE/FALSE).

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_preserve

Signature:

int `H5Pget_preserve`(*hid_t* `plist`)

Purpose:

Checks status of the dataset transfer property list.

Description:

`H5Pget_preserve` checks the status of the dataset transfer property list.

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

Returns:

Returns TRUE or FALSE if successful; otherwise returns a negative value.

Name: H5Pset_deflate

Signature:

herr_t `H5Pset_deflate`(*hid_t* `plist`, *int* `level`)

Purpose:

Sets compression method and compression level.

Description:

`H5Pset_deflate` sets the compression method for a dataset creation property list to `H5D_COMPRESS_DEFLATE` and the compression level to `level`, which should be a value from zero to nine, inclusive. Lower compression levels are faster but result in less compression. This is the same algorithm as used by the GNU gzip program.

Parameters:

hid_t `plist`

IN: Identifier for the dataset creation property list.

int `level`

IN: Compression level.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pset_vlen_mem_manager

Signature:

herr_t `H5Pset_vlen_mem_manager`(*hid_t* `plist`, *H5MM_allocate_t* `alloc`, *void* *`alloc_info`, *H5MM_free_t* `free`, *void* *`free_info`)

Purpose:

Sets the memory manager for variable-length datatype allocation in `H5Dread` and `H5Dvlen_reclaim`.

Description:

`H5Pset_vlen_mem_manager` sets the memory manager for variable-length datatype allocation in `H5Dread` and free in `H5Dvlen_reclaim`.

The `alloc` and `free` parameters identify the memory management routines to be used. If the user has defined custom memory management routines, `alloc` and/or `free` should be set to make those routine calls (i.e., the name of the routine is used as the value of the parameter); if the user prefers to use the system's `malloc` and/or `free`, the `alloc` and `free` parameters, respectively, should be set to `NULL`

The prototypes for these user-defined functions would appear as follows:
typedef void *(*`H5MM_allocate_t`)(*size_t* `size`, *void* *`alloc_info`) ;
typedef void (*`H5MM_free_t`)(*void* *`mem`, *void* *`free_info`) ;
The `alloc_info` and `free_info` parameters can be used to pass along any required information to the user's memory management routines.

In summary, if the user has defined custom memory management routines, the name(s) of the routines are passed in the `alloc` and `free` parameters and the custom routines' parameters are passed in the `alloc_info` and `free_info` parameters. If the user wishes to use the system `malloc` and `free` functions, the `alloc` and/or `free` parameters are set to `NULL` and the `alloc_info` and `free_info` parameters are ignored.

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

H5MM_allocate_t `alloc`

IN: User's allocate routine, or `NULL` for system `malloc`.

void *`alloc_info`

IN: Extra parameter for user's allocation routine. Ignored if preceding parameter is `NULL`.

H5MM_free_t `free`

IN: User's free routine, or `NULL` for system `free`.

void *`free_info`

IN: Extra parameter for user's free routine. Ignored if preceding parameter is `NULL`.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_vlen_mem_manager

Signature:

herr_t `H5Pget_vlen_mem_manager`(*hid_t* `plist`, *H5MM_allocate_t* *`alloc`, *void* **`alloc_info`, *H5MM_free_t* *`free`, *void* **`free_info`)

Purpose:

Gets the memory manager for variable-length datatype allocation in `H5Dread` and `H5Treclaim_vlen`.

Description:

`H5Pget_vlen_mem_manager` is the companion function to `H5Pset_vlen_mem_manager`, returning the parameters set by that function.

Parameters:

hid_t `plist`

IN: Identifier for the dataset transfer property list.

H5MM_allocate_t `alloc`

OUT: User’s allocate routine, or `NULL` for system `malloc`.

void `*alloc_info`

OUT: Extra parameter for user’s allocation routine. Ignored if preceding parameter is `NULL`.

H5MM_free_t `free`

OUT: User’s free routine, or `NULL` for system `free`.

void `*free_info`

OUT: Extra parameter for user’s free routine. Ignored if preceding parameter is `NULL`.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Last modified: 20 October 1999

H5R: Reference Interface

Reference API Functions

The Reference interface allows the user to create references to specific objects and data regions in an HDF5 file.

- H5Rcreate
- H5Rdereference
- H5Rget_region
- H5Rget_object_type

Name: H5Rcreate

Signature:

herr_t `H5Rcreate`(*void* *`ref`, *hid_t* `loc_id`, *const char* *`name`, *H5R_type_t* `ref_type`, *hid_t* `space_id`)

Purpose:

Creates a reference.

Description:

`H5Rcreate` creates the reference, `ref`, of the type specified in `ref_type`, pointing to the object `name` located at `loc_id`.

The parameters `loc_id` and `name` are used to locate the object.

The parameter `space_id` identifies the region to be pointed to (for dataset region references).

Parameters:

void *`ref`

OUT: Reference created by the function call.

hid_t `loc_id`

IN: Location identifier used to locate the object being pointed to.

const char *`name`

IN: Name of object at location `loc_id`.

H5R_type_t `ref_type`

IN: Type of reference.

hid_t `space_id`

IN: Dataspace identifier with selection. Used for dataset region references.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Rdereference

Signature:

hid_t `H5Rdereference`(*hid_t* `dataset`, *H5R_type_t* `ref_type`, *void* `*ref`)

Purpose:

Opens the HDF5 object referenced.

Description:

Given a reference to some object, `H5Rdereference` opens that object and returns an identifier.

The parameter `ref_type` specifies the reference type of `ref`. `ref_type` may contain either of the following values:

- `H5R_OBJECT` (`0`)
- `H5R_DATASET_REGION` (`1`)

Parameters:

hid_t `dataset`

IN: Dataset containing reference object.

H5R_type_t `ref_type`

IN: The reference type of `ref`.

void `*ref`

IN: Reference to open.

Returns:

Returns valid identifier if successful; otherwise returns a negative value.

Name: H5Rget_region

Signature:

hid_t `H5Rget_region`(*hid_t* `dataset`, *H5R_type_t* `ref_type`, *void* `*ref`)

Purpose:

Retrieves a dataspace with the specified region selected.

Description:

Given a reference to an object `ref`, `H5Rget_region` creates a copy of the dataspace of the dataset pointed to and defines a selection in the copy which is the region pointed to.

The parameter `ref_type` specifies the reference type of `ref`. `ref_type` may contain the following value:

- `H5R_DATASET_REGION` (`1`)

Parameters:

hid_t `dataset`,

IN: Dataset containing reference object.

H5R_type_t `ref_type`,

IN: The reference type of `ref`.

void `*ref`

IN: Reference to open.

Returns:

Returns a valid identifier if successful; otherwise returns a negative value.

Name: H5Rget_object_type

Signature:

int `H5Rget_object_type`(*hid_t* `id`, *void* `*ref`)

Purpose:

Retrieves the type of object that an object reference points to.

Description:

Given a reference to an object `ref`, `H5Rget_object_type` returns the type of the object pointed to.

Parameters:

hid_t `id`,

IN: The dataset containing the reference object or the location identifier of the object that the dataset is located within.

void `*ref`

IN: Reference to query.

Returns:

Returns an object type as defined in `H5Gpublic.h`; otherwise returns `H5G_UNKNOWN`.

Last modified: 30 October 1998

H5S: Dataspace Interface

Dataspace Object API Functions

These functions create and manipulate the dataspace in which to store the elements of a dataset.

H5Screate	H5Sget_simple_extent_npoints	H5Sget_select_elem_npoints
H5Scopy	H5Sget_simple_extent_type	H5Sget_select_elem_pointlist
H5Sclose	H5Sextent_copy	H5Sget_select_bounds
H5Screate_simple	H5Sset_extent_simple	H5Sselect_elements
H5Sis_simple	H5Sset_extent_none	H5Sselect_all
H5Soffset_simple	H5Sget_select_npoints	H5Sselect_none
H5Sget_simple_extent_dims	H5Sget_select_hyper_nblocks	H5Sselect_valid
H5Sget_simple_extent_ndims	H5Sget_select_hyper_blocklist	H5Sselect_hyperslab

The following H5S functions are included in the HDF5 specification, but have not yet been implemented. They are described in the *The Dataspace Interface (H5S)* section of the *HDF5 User's Guide*..

- H5Scommit
- H5Sis_subspace
- H5Slock
- H5Sopen
- H5Sselect_op
- H5Sselect_order
- H5Ssubspace
- H5Ssubspace_name
- H5Ssubspace_location

Name: H5Screate

Signature:

hid_t `H5Screate`(*H5S_class_t* `type`)

Purpose:

Creates a new dataspace of a specified type.

Description:

`H5Screate` creates a new dataspace of a particular `type`. The types currently supported are `H5S_SCALAR` and `H5S_SIMPLE`; others are planned to be added later.

Parameters:

H5S_class_t `type`

The type of dataspace to be created.

Returns:

Returns a dataspace identifier if successful; otherwise returns a negative value.

Name: H5Screate_simple

Signature:

hid_t `H5Screate_simple`(*int* `rank`, *const hsize_t* * `dims`, *const hsize_t* * `maxdims`)

Purpose:

Creates a new simple data space and opens it for access.

Description:

`H5Screate_simple` creates a new simple data space and opens it for access. The `rank` is the number of dimensions used in the dataspace. The `dims` argument is the size of the simple dataset and the `maxdims` argument is the upper limit on the size of the dataset. `maxdims` may be the null pointer in which case the upper limit is the same as `dims`. If an element of `maxdims` is zero then the corresponding dimension is unlimited, otherwise no element of `maxdims` should be smaller than the corresponding element of `dims`. The dataspace identifier returned from this function should be released with `H5Sclose` or resource leaks will occur.

Parameters:

int `rank`

Number of dimensions of dataspace.

const hsize_t * `dims`

An array of the size of each dimension.

const hsize_t * `maxdims`

An array of the maximum size of each dimension.

Returns:

Returns a dataspace identifier if successful; otherwise returns a negative value.

Name: H5Scopy

Signature:

hid_t `H5Scopy`(*hid_t* `space_id`)

Purpose:

Creates an exact copy of a dataspace.

Description:

`H5Scopy` creates a new dataspace which is an exact copy of the dataspace identified by `space_id`. The dataspace identifier returned from this function should be released with `H5Sclose` or resource leaks will occur.

Parameters:

hid_t `space_id`

Identifier of dataspace to copy.

Returns:

Returns a dataspace identifier if successful; otherwise returns a negative value.

Name: H5Sselect_elements

Signature:

herr_t `H5Sselect_elements`(*hid_t* `space_id`, *H5S_seloper_t* `op`, *const size_t* `num_elements`, *const hssize_t* `*coord[ ]`)

Purpose:

Selects array elements to be included in the selection for a dataspace.

Description:

`H5Sselect_elements` selects array elements to be included in the selection for the `space_id` dataspace. The number of elements selected must be set with the `num_elements`. The `coord` array is a two-dimensional array of size *dataspace rank* by `num_elements` (ie. a list of coordinates in the array). The order of the element coordinates in the `coord` array also specifies the order in which the array elements are iterated through when I/O is performed. Duplicate coordinate locations are not checked for.

The selection operator `op` determines how the new selection is to be combined with the previously existing selection for the dataspace. The following operators are supported:

`H5S_SELECT_SET`	Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
`H5S_SELECT_OR`	Adds the new selection to the existing selection.

When operators other than `H5S_SELECT_SET` are used to combine a new selection with an existing selection, the selection ordering is reset to 'C' array ordering.

Parameters:

hid_t `space_id`

Identifier of the dataspace.

H5S_seloper_t `op`

operator specifying how the new selection is to be combined with the existing selection for the dataspace.

const size_t `num_elements`

Number of elements to be selected.

const hssize_t *`coord`[ ]

A 2-dimensional array specifying the coordinates of the elements being selected.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sselect_all

Signature:

herr_t `H5Sselect_all`(*hid_t* `space_id`)

Purpose:

Selects the entire dataspace.

Description:

`H5Sselect_all` selects the entire extent of the dataspace `space_id`.

More specifically, `H5Sselect_all` selects the special `5S_SELECT_ALL` region for the dataspace `space_id`. `H5S_SELECT_ALL` selects the entire dataspace for any dataspace it is applied to.

Parameters:

hid_t `space_id`

IN: The identifier for the dataspace in which the selection is being made.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sselect_none

Signature:

herr_t `H5Sselect_none`(*hid_t* `space_id`)

Purpose:

Resets the selection region to include no elements.

Description:

`H5Sselect_none` resets the selection region for the dataspace `space_id` to include no elements.

Parameters:

hid_t `space_id`

IN: The identifier for the dataspace in which the selection is being reset.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sselect_valid

Signature:

htri_t `H5Sselect_valid`(*hid_t* `space_id`)

Purpose:

Verifies that the selection is within the extent of the dataspace.

Description:

`H5Sselect_valid` verifies that the selection for the dataspace `space_id` is within the extent of the dataspace if the current offset for the dataspace is used.

Parameters:

hid_t `space_id`

The identifier for the dataspace in which the selection is being reset.

Returns:

Returns a positive value, for `TRUE`, if the selection is contained within the extent or `0` (zero), for `FALSE`, if it is not. Returns a negative value on error conditions such as the selection or extent not being defined.

Name: H5Sget_simple_extent_npoints

Signature:

hsize_t `H5Sget_simple_extent_npoints`(*hid_t* `space_id`)

Purpose:

Determines the number of elements in a dataspace.

Description:

`H5Sget_simple_extent_npoints` determines the number of elements in a dataspace. For example, a simple 3-dimensional dataspace with dimensions 2, 3, and 4 would have 24 elements.

Parameters:

hid_t `space_id`

ID of the dataspace object to query

Returns:

Returns the number of elements in the dataspace if successful; otherwise returns 0.

Name: H5Sget_select_npoints

Signature:

hssize_t `H5Sget_select_npoints`(*hid_t* `space_id`)

Purpose:

Determines the number of elements in a dataspace selection.

Description:

`H5Sget_select_npoints` determines the number of elements in the current selection of a dataspace.

Parameters:

hid_t `space_id`

Dataspace identifier.

Returns:

Returns the number of elements in the selection if successful; otherwise returns a negative value.

Name: H5Sget_simple_extent_ndims

Signature:

int `H5Sget_simple_extent_ndims`(*hid_t* `space_id`)

Purpose:

Determines the dimensionality of a dataspace.

Description:

`H5Sget_simple_extent_ndims` determines the dimensionality (or rank) of a dataspace.

Parameters:

hid_t `space_id`

Identifier of the dataspace

Returns:

Returns the number of dimensions in the dataspace if successful; otherwise returns a negative value.

Name: H5Sget_simple_extent_dims

Signature:

int `H5Sget_simple_extent_dims`(*hid_t* `space_id`, *hsize_t* *`dims`, *hsize_t* *`maxdims`)

Purpose:

Retrieves dataspace dimension size and maximum size.

Description:

`H5Sget_simple_extent_dims` returns the size and maximum sizes of each dimension of a dataspace through the `dims` and `maxdims` parameters.

Parameters:

hid_t `space_id`

IN: Identifier of the dataspace object to query

hsize_t *`dims`

OUT: Pointer to array to store the size of each dimension.

hsize_t *`maxdims`

OUT: Pointer to array to store the maximum size of each dimension.

Returns:

Returns the number of dimensions in the dataspace if successful; otherwise returns a negative value.

Name: H5Sget_simple_extent_type

Signature:

H5S_class_t `H5Sget_simple_extent_type`(*hid_t* `space_id`)

Purpose:

Determine the current class of a dataspace.

Description:

`H5Sget_simple_extent_type` queries a dataspace to determine the current class of a dataspace.

The function returns a class name, one of the following: `H5S_SCALAR`, `H5S_SIMPLE`, or `H5S_NONE`.

Parameters:

hid_t `space_id`

Dataspace identifier.

Returns:

Returns a dataspace class name if successful; otherwise H5S_NO_CLASS (-1).

Name: H5Sset_extent_simple

Signature:

herr_t `H5Sset_extent_simple`(*hid_t* `space_id`, *int* `rank`, *const hsize_t* *`current_size`, *const hsize_t* *`maximum_size`)

Purpose:

Sets or resets the size of an existing dataspace.

Description:

`H5Sset_extent_simple` sets or resets the size of an existing dataspace.

`rank` is the dimensionality, or number of dimensions, of the dataspace.

`current_size` is an array of size `rank` which contains the new size of each dimension in the dataspace. `maximum_size` is an array of size `rank` which contains the maximum size of each dimension in the dataspace.

Any previous extent is removed from the dataspace, the dataspace type is set to `H5S_SIMPLE`, and the extent is set as specified.

Parameters:

hid_t `space_id`

Dataspace identifier.

int `rank`

Rank, or dimensionality, of the dataspace.

const hsize_t *`current_size`

Array containing current size of dataspace.

const hsize_t *`maximum_size`

Array containing maximum size of dataspace.

Returns:

Returns a dataspace identifier if successful; otherwise returns a negative value.

Name: H5Sis_simple

Signature:

htri_t `H5Sis_simple`(*hid_t* `space_id`)

Purpose:

Determines whether a dataspace is a simple dataspace.

Description:

`H5Sis_simple` determines whether a dataspace is a simple dataspace. [Currently, all dataspace objects are simple dataspaces, complex dataspace support will be added in the future]

Parameters:

hid_t `space_id`

Identifier of the dataspace to query

Returns:

When successful, returns a positive value, for `TRUE`, or `0` (zero), for `FALSE`. Otherwise returns a negative value.

Name: H5Soffset_simple

Signature:

herr_t `H5Soffset_simple`(*hid_t* `space_id`, *const hssize_t* `*offset`)

Purpose:

Sets the offset of a simple dataspace.

Description:

`H5Soffset_simple` sets the offset of a simple dataspace `space_id`. The `offset` array must be the same number of elements as the number of dimensions for the dataspace. If the `offset` array is set to `NULL`, the offset for the dataspace is reset to 0.

This function allows the same shaped selection to be moved to different locations within a dataspace without requiring it to be redefined.

Parameters:

hid_t `space_id`

IN: The identifier for the dataspace object to reset.

const hssize_t `*offset`

IN: The offset at which to position the selection.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sextent_copy

Signature:

herr_t `H5Sextent_copy`(*hid_t* `dest_space_id`, *hid_t* `source_space_id`)

Purpose:

Copies the extent of a dataspace.

Description:

`H5Sextent_copy` copies the extent from `source_space_id` to `dest_space_id`. This action may change the type of the dataspace.

Parameters:

hid_t `dest_space_id`

IN: The identifier for the dataspace to which the extent is copied.

hid_t `source_space_id`

IN: The identifier for the dataspace from which the extent is copied.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sset_extent_none

Signature:

herr_t `H5Sset_extent_none`(*hid_t* `space_id`)

Purpose:

Removes the extent from a dataspace.

Description:

`H5Sset_extent_none` removes the extent from a dataspace and sets the type to `H5S_NO_CLASS`.

Parameters:

hid_t `space_id`

The identifier for the dataspace from which the extent is to be removed.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sselect_hyperslab

Signature:

herr_t `H5Sselect_hyperslab`(*hid_t* `space_id`, *H5S_seloper_t*`op`, *const hssize_t* *`start`, *const hsize_t* *`stride` *const hsize_t* *`count`, *const hsize_t* *`block`)

Purpose:

Selects a hyperslab region to add to the current selected region.

Description:

`H5Sselect_hyperslab` selects a hyperslab region to add to the current selected region for the dataspace specified by `space_id`.

The `start`, `stride`, `count`, and `block` arrays must be the same size as the rank of the dataspace.

The selection operator `op` determines how the new selection is to be combined with the already existing selection for the dataspace.

The following operators are supported:

`H5S_SELECT_SET`	Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
`H5S_SELECT_OR`	Adds the new selection to the existing selection.

The `start` array determines the starting coordinates of the hyperslab to select.

The `stride` array chooses array locations from the dataspace with each value in the `stride` array determining how many elements to move in each dimension. Setting a value in the `stride` array to 1 moves to each element in that dimension of the dataspace; setting a value of 2 in a location in the `stride` array moves to every other element in that dimension of the dataspace. In other words, the `stride` determines the number of elements to move from the `start` location in each dimension. Stride values of 0 are not allowed. If the `stride` parameter is `NULL`, a contiguous hyperslab is selected (as if each value in the `stride` array was set to all 1's).

The `count` array determines how many blocks to select from the dataspace, in each dimension.

The `block` array determines the size of the element block selected from the dataspace. If the `block` parameter is set to `NULL`, the block size defaults to a single element in each dimension (as if the `block` array was set to all 1's).

For example, in a 2-dimensional dataspace, setting `start` to [1,1], `stride` to [4,4], `count` to [3,7], and `block` to [2,2] selects 21 2x2 blocks of array elements starting with location (1,1) and selecting blocks at locations (1,1), (5,1), (9,1), (1,5), (5,5), etc.

Regions selected with this function call default to C order iteration when I/O is performed.

Parameters:

hid_t `space_id`

IN: Identifier of dataspace selection to modify

H5S_seloper_t `op`

IN: Operation to perform on current selection.

const hssize_t `*start`

IN: Offset of start of hyperslab

const hsize_t `*count`

IN: Number of blocks included in hyperslab.

const hsize_t `*stride`

IN: Hyperslab stride.

const hsize_t `*block`

IN: Size of block in hyperslab.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sget_select_hyper_nblocks

Signature:

hssize_t `H5Sget_select_hyper_nblocks`(*hid_t* `space_id`)

Purpose:

Get number of hyperslab blocks.

Description:

`H5Sget_select_hyper_nblocks` returns the number of hyperslab blocks in the current dataspace selection.

Parameters:

hid_t `space_id`

IN: Identifier of dataspace to query.

Returns:

Returns the number of hyperslab blocks in the current dataspace selection if successful. Otherwise returns a negative value.

Name: H5Sget_select_hyper_blocklist

Signature:

herr_t `H5Sget_select_hyper_blocklist`(*hid_t* `space_id`, *hsize_t* `startblock`, *hsize_t* `numblocks`, *hsize_t* `*buf`)

Purpose:

Gets the list of hyperslab blocks currently selected.

Description:

`H5Sget_select_hyper_blocklist` returns a list of the hyperslab blocks currently selected. Starting with the `startblock`-th block in the list of blocks, `numblocks` blocks are put into the user's buffer. If the user's buffer fills up before `numblocks` blocks are inserted, the buffer will contain only as many blocks as fit.

The block coordinates have the same dimensionality (rank) as the dataspace they are located within. The list of blocks is formatted as follows:
<"start" coordinate>, immediately followed by
<"opposite" corner coordinate>, followed by
the next "start" and "opposite" coordinates,
etc.
until all of the selected blocks have been listed.

No guarantee is implied as the order in which blocks are listed.

Parameters:

hid_t `space_id`

IN: Dataspace identifier of selection to query.

hsize_t `startblock`

IN: Hyperslab block to start with.

hsize_t `numblocks`

IN: Number of hyperslab blocks to get.

hsize_t `*buf`

OUT: List of hyperslab blocks selected.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sget_select_elem_npoints

Signature:

hssize_t `H5Sget_select_elem_npoints`(*hid_t* `space_id`)

Purpose:

Gets the number of element points in the current selection.

Description:

`H5Sget_select_elem_npoints` returns the number of element points in the current dataspace selection.

Parameters:

hid_t `space_id`

IN: Identifier of dataspace to query.

Returns:

Returns the number of element points in the current dataspace selection if successful. Otherwise returns a negative value.

Name: H5Sget_select_elem_pointlist

Signature:

herr_t `H5Sget_select_elem_pointlist`(*hid_t* `space_id` *hsize_t* `startpoint`, *hsize_t* `numpoints`, *hsize_t* `*buf`)

Purpose:

Gets the list of element points currently selected.

Description:

`H5Sget_select_elem_pointlist` returns the list of element points in the current dataspace selection. Starting with the `startpoint`-th point in the list of points, `numpoints` points are put into the user's buffer. If the user's buffer fills up before `numpoints` points are inserted, the buffer will contain only as many points as fit.

The element point coordinates have the same dimensionality (rank) as the dataspace they are located within. The list of element points is formatted as follows:
<coordinate>, followed by
the next coordinate,
etc.
until all of the selected element points have been listed.

The points are returned in the order they will be iterated through when the selection is read/written from/to disk.

Parameters:

hid_t `space_id`

IN: Dataspace identifier of selection to query.

hsize_t `startpoint`

IN: Element point to start with.

hsize_t `numpoints`

IN: Number of element points to get.

hsize_t `*buf`

OUT: List of element points selected.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Sget_select_bounds

Signature:

herr_t `H5Sget_select_bounds`(*hid_t* `space_id` *hsize_t* `*start`, *hsize_t* `*end`)

Purpose:

Gets the bounding box containing the current selection.

Description:

`H5Sget_select_bounds` retrieves the coordinates of the bounding box containing the current selection and places them into user-supplied buffers.

The `start` and `end` buffers must be large enough to hold the dataspace rank number of coordinates.

The bounding box exactly contains the selection. I.e., if a 2-dimensional element selection is currently defined as containing the points (4,5), (6,8), and (10,7), then the bounding box will be (4, 5), (10, 8).

The bounding box calculation includes the current offset of the selection within the dataspace extent.

Calling this function on a `none` selection will return `FAIL`.

Parameters:

hid_t `space_id`

IN: Identifier of dataspace to query.

hsize_t `*start`

OUT: Starting coordinates of the bounding box.

hsize_t `*end`

OUT: Ending coordinates of the bounding box, i.e., the coordinates of the diagonally opposite corner.

Returns:

Returns a negative value if successful; otherwise returns a negative value.

Name: H5Sclose

Signature:

herr_t `H5Sclose`(*hid_t* `space_id`)

Purpose:

Releases and terminates access to a dataspace.

Description:

`H5Sclose` releases a dataspace. Further access through the dataspace identifier is illegal. Failure to release a dataspace with this call will result in resource leaks.

Parameters:

hid_t `space_id`

Identifier of dataspace to release.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Last modified: 20 October 1999

H5T: Datatype Interface

Datatype Object API Functions

These functions create and manipulate the datatype which describes elements of a dataset.

General Datatype Operations

H5Tcreate
H5Topen
H5Tcommit
H5Tcommitted
H5Tcopy
H5Tequal
H5Tlock
H5Tget_class
H5Tget_size
H5Tget_super
H5Tclose

Conversion Functions

H5Tconvert
H5Tfind
H5Tset_overflow
H5Tget_overflow
H5Tregister
H5Tunregister

Variable-length Datatypes

H5Tvlen_create

Atomic Datatype Properties

H5Tset_size
H5Tget_order
H5Tset_order
H5Tget_precision
H5Tset_precision
H5Tget_offset
H5Tset_offset
H5Tget_pad
H5Tset_pad
H5Tget_sign
H5Tset_sign
H5Tget_fields
H5Tset_fields
H5Tget_ebias
H5Tset_ebias
H5Tget_norm
H5Tset_norm
H5Tget_inpad
H5Tset_inpad
H5Tget_cset
H5Tset_cset
H5Tget_strpad
H5Tset_strpad

Compound Datatype Properties

H5Tget_nmembers
H5Tget_member_name
H5Tget_member_offset
H5Tget_member_dims
H5Tget_member_type
H5Tinsert
H5Tpack
H5Tinsert_array

Enumeration Datatypes

H5Tenum_create
H5Tenum_insert
H5Tenum_nameof
H5Tenum_valueof
H5Tget_member_value

Opaque Datatypes

H5Tset_tag
H5Tget_tag

The Datatype interface, H5T, provides a mechanism to describe the storage format of individual data points of a data set and is hopefully designed in such a way as to allow new features to be easily added without disrupting applications that use the data type interface. A dataset (the H5D interface) is composed of a collection or raw data points of homogeneous type organized according to the data space (the H5S interface).

A datatype is a collection of datatype properties, all of which can be stored on disk, and which when taken as a whole, provide complete information for data conversion to or from that datatype. The interface provides functions to set and query properties of a datatype.

A *data point* is an instance of a *datatype*, which is an instance of a *type class*. We have defined a set of type classes and properties which can be extended at a later time. The atomic type classes are those which describe types which cannot be decomposed at the datatype interface level; all other classes are compound.

See the “Datatype Interface (H5T)” section in the *HDF5 User’s Guide* for further information, inclding a complete list of all supported datatypes.

Name: H5Topen

Signature:

hid_t`H5Topen`(*hid_t* `loc_id`, *const char* * `name`)

Purpose:

Opens a named datatype.

Description:

`H5Topen` opens a named datatype at the location specified by `loc_id` and returns an identifier for the datatype. `loc_id` is either a file or group identifier. The identifier should eventually be closed by calling `H5Tclose()` to release resources.

Parameters:

hid_t `loc_id`

IN: A file or group identifier.

const char * `name`

IN: A datatype name, defined within the file or group identified by `loc_id`.

Returns:

Returns a named datatype identifier if successful; otherwise returns a negative value.

Name: H5Tcommit

Signature:

herr_t`H5Tcommit`(*hid_t* `loc_id`, *const char* * `name`, *hid_t* `type`)

Purpose:

Commits a transient datatype to a file, creating a new named datatype.

Description:

`H5Tcommit` commits a transient datatype (not immutable) to a file, turned it into a named datatype. The `loc_id` is either a file or group identifier which, when combined with `name`, refers to a new named datatype.

Parameters:

hid_t `loc_id`

IN: A file or group identifier.

const char * `name`

IN: A datatype name.

hid_t `type`

IN: A datatype identifier.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tcommitted

Signature:

htri_t`H5Tcommitted`(*hid_t* `type`)

Purpose:

Determines whether a datatype is a named type or a transient type.

Description:

`H5Tcommitted` queries a type to determine whether the type specified by the `type` identifier is a named type or a transient type. If this function returns a positive value, then the type is named (that is, it has been committed, perhaps by some other application). Datasets which return committed datatypes with `H5Dget_type()` are able to share the datatype with other datasets in the same file.

Parameters:

hid_t `type`

IN: Datatype identifier.

Returns:

When successful, returns a positive value, for `TRUE`, if the datatype has been committed, or `0` (zero), for `FALSE`, if the datatype has not been committed. Otherwise returns a negative value.

Name: H5Tinsert_array

Signature:

herr_t`H5Tinsert_array`(*hid_t* `parent_id`, *const char* *`name`, *size_t* `offset`, *int* `ndims`, *const size_t* *`dim`, *const int* *`perm`, *hid_t* `member_id`)

Purpose:

Adds an array member to a compound datatype.

Description:

`H5Tinsert_array` adds a new member to the compound datatype `parent_id`. The member is an array with `ndims` dimensionality and the size of the array is `dim`. The new member's name, `name`, must be unique within the compound datatype. The `offset` argument defines the byte offset of the start of the member in an instance of the compound datatype and `member_id` is the type identifier of the new member. The total member size should be relatively small.

The functionality of the `perm` parameter has not yet been implemented. Currently, `perm` is best set to `NULL`. (When implemented, `perm` will specify the mapping of dimensions within a struct. At that time, a `NULL` value for `perm` will mean no mappiing change is to take place. Thus, using a value of `NULL` ensures that application behavior will remain unchanged upon implementation of this functionality.)

Parameters:

hid_t `parent_id`

IN: Identifier of the parent compound datatype.

const char *`name`

IN: Name of new member.

size_t `offset`

IN: Offset to start of new member within compound datatype.

int `ndims`

IN: Dimensionality of new member. Valid values are `0` (zero) through `4` (four).

const size_t *`dim`

IN: Size of new member array.

const int *`perm`

IN: Pointer to buffer to store the permutation vector of the field.

hid_t `member_id`

IN: Identifier of the datatype of the new member.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tfind

Signature:

H5T_conv_t `H5Tfind`(*hid_t* `src_id`, *hid_t* `dst_id`, *H5T_cdata_t* **`pcdata`)

Purpose:

Finds a conversion function.

Description:

`H5Tfind` finds a conversion function that can handle a conversion from type `src_id` to type `dst_id`. The `pcdata` argument is a pointer to a pointer to type conversion data which was created and initialized by the soft type conversion function of this path when the conversion function was installed on the path.

Parameters:

hid_t `src_id`

IN: Identifier for the source datatype.

hid_t `dst_id`

IN: Identifier for the destination datatype.

H5T_cdata_t `**pcdata`

IN: Pointer to type conversion data.

Returns:

Returns a pointer to a suitable conversion function if successful. Otherwise returns NULL.

Name: H5Tconvert

Signature:

herr_t `H5Tconvert(`*hid_t* `src_id`, *hid_t* `dst_id`, *size_t* `nelmts`, *void* `*buf`, *void* `*background`)

Purpose:

Converts data from between specified datatypes.

Description:

`H5Tconvert` converts `nelmts` elements from the type specified by the `src_id` identifier to type `dst_id`. The source elements are packed in `buf` and on return the destination will be packed in `buf`. That is, the conversion is performed in place. The optional background buffer is an array of `nelmts` values of destination type which are merged with the converted values to fill in cracks (for instance, `background` might be an array of structs with the `a` and `b` fields already initialized and the conversion of `buf` supplies the `c` and `d` field values).

Parameters:

hid_t `src_id`

Identifier for the source datatype.

hid_t `dst_id`

Identifier for the destination datatype.

size_t `nelmts`

Size of array `buf`.

void `*buf`

Array containing pre- and post-conversion values.

void `*background`

Optional background buffer.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tset_overflow

Signature:

herr_t `H5Tset_overflow`(*H5T_overflow_t* `func`)

Purpose:

Sets the overflow handler to a specified function.

Description:

`H5Tset_overflow` sets the overflow handler to be the function specified by `func`. `func` will be called for all datatype conversions that result in an overflow.

See the definition of `H5T_overflow_t` in `H5Tpublic.h` for documentation of arguments and return values. The prototype for `H5T_overflow_t` is as follows:

```
herr_t (*H5T_overflow_t)(hid_t src_id, hid_t dst_id, void *src_buf, void *dst_buf);
```

The NULL pointer may be passed to remove the overflow handler.

Parameters:

H5T_overflow_t `func`

Overflow function.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_overflow

Signature: H5Tget_overflow ()

H5T_overflow_t `H5Tget_overflow`(`void`)

Purpose:

Returns a pointer to the current global overflow function.

Description:

`H5Tset_overflow` returns a pointer to the current global overflow function. This is an application-defined function that is called whenever a datatype conversion causes an overflow.

Parameters:

None.

Returns:

Returns a pointer to an application-defined function if successful. Otherwise returns NULL; this can happen if no overflow handling function is registered.

Name: H5Tcreate

Signature:

hid_t `H5Tcreate`(*H5T_class_t* `class`, *size_t*`size`)

Purpose:

Creates a new dataype.

Description:

`H5Tcreate` creates a new dataype of the specified class with the specified number of bytes.

The following datatype classes are supported with this function:

- `H5T_COMPOUND`
- `H5T_OPAQUE`
- `H5T_ENUM`

Use `H5Tcopy` to create integer or floating-point datatypes.

The datatype identifier returned from this function should be released with `H5Tclose` or resource leaks will result.

Parameters:

H5T_class_t `class`

Class of datatype to create.

size_t `size`

The number of bytes in the datatype to create.

Returns:

Returns datatype identifier if successful; otherwise returns a negative value.

Name: H5Tvlen_create

Signature:

hid_t `H5Tvlen_create`(*hid_t* `base_type_id`)

Purpose:

Creates a new variable-length dataype.

Description:

`H5Tvlen_create` creates a new variable-length (VL) dataype.

The base datatype will be the datatype that the sequence is composed of, characters for character strings, vertex coordinates for polygon lists, etc. The base type specified for the VL datatype can be of any HDF5 datatype, including another VL datatype, a compound datatype or an atomic datatype.

When necessary, use `H5Tget_super` to determine the base type of the VL datatype.

The datatype identifier returned from this function should be released with `H5Tclose` or resource leaks will result.

Parameters:

hid_t `base_type_id`

Base type of datatype to create.

Returns:

Returns datatype identifier if successful; otherwise returns a negative value.

Name: H5Tcopy

Signature:

hid_t `H5Tcopy`(*hid_t* `type_id`)

Purpose:

Copies an existing datatype.

Description:

`H5Tcopy` copies an existing datatype. The returned type is always transient and unlocked.

The `type_id` argument can be either a datatype identifier, a predefined datatype (defined in `H5Tpublic.h`), or a dataset identifier. If `type_id` is a dataset identifier instead of a datatype identifier, then this function returns a transient, modifiable datatype which is a copy of the dataset's datatype.

The datatype identifier returned should be released with `H5Tclose` or resource leaks will occur.

Parameters:

hid_t `type_id`

Identifier of datatype to copy. Can be a datatype identifier, a predefined datatype (defined in `H5Tpublic.h`), or a dataset identifier.

Returns:

Returns a datatype identifier if successful; otherwise returns a negative value

---

Name: H5Tequal

Signature:

htri_t `H5Tequal`(*hid_t* `type_id1`, *hid_t*`type_id2`)

Purpose:

Determines whether two datatype identifiers refer to the same datatype.

Description:

`H5Tequal` determines whether two datatype identifiers refer to the same datatype.

Parameters:

hid_t `type_id1`

Identifier of datatype to compare.

hid_t `type_id2`

Identifier of datatype to compare.

Returns:

When successful, returns a positive value, for `TRUE`, if the datatype identifiers refer to the same datatype, or `0` (zero), for `FALSE`. Otherwise returns a negative value.

Name: H5Tlock

Signature:

herr_t `H5Tlock`(*hid_t* `type_id`)

Purpose:

Locks a datatype.

Description:

`H5Tlock` locks the datatype specified by the `type_id` identifier, making it read-only and non-destrucible. This is normally done by the library for predefined datatypes so the application does not inadvertently change or delete a predefined type. Once a datatype is locked it can never be unlocked.

Parameters:

hid_t `type_id`

Identifier of datatype to lock.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_class

Signature:

H5T_class_t `H5Tget_class`(*hid_t* `type_id`)

Purpose:

Returns the datatype class identifier.

Description:

`H5Tget_class` returns the datatype class identifier.

Valid class identifiers, as defined in `H5Tpublic.h`, are:

- `H5T_INTEGER` (0)
- `H5T_FLOAT` (1)
- `H5T_TIME` (2)
- `H5T_STRING` (3)
- `H5T_BITFIELD` (4)
- `H5T_OPAQUE` (5)
- `H5T_COMPOUND` (6)
- `H5T_ENUM` (7)
- `H5T_REFERENCE` (8)

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns datatype class identifier if successful; otherwise H5T_NO_CLASS (-1).

Name: H5Tget_size

Signature:

size_t `H5Tget_size`(*hid_t* `type_id`)

Purpose:

Returns the size of a datatype.

Description:

`H5Tget_size` returns the size of a datatype in bytes.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns the size of the datatype in bytes if successful; otherwise 0.

Name: H5Tset_size

Signature:

herr_t `H5Tset_size`(*hid_t* `type_id`, *size_t*`size`)

Purpose:

Sets the total size for an atomic datatype.

Description:

`H5Tset_size` sets the total size in bytes, `size`, for an atomic datatype (this operation is not permitted on compound datatypes). If the size is decreased so that the significant bits of the datatype extend beyond the edge of the new size, then the ‘offset’ property is decreased toward zero. If the ‘offset’ becomes zero and the significant bits of the datatype still hang over the edge of the new size, then the number of significant bits is decreased. Adjusting the size of an H5T_STRING automatically sets the precision to 8*size. All datatypes have a positive size.

Parameters:

hid_t `type_id`

Identifier of datatype to change size.

size_t `size`

Size in bytes to modify datatype.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_super

Signature:

hid_t `H5Tget_super`(*hid_t* `type`)

Purpose:

Returns the base datatype from which a datatype is derived.

Description:

`H5Tget_super` returns the base datatype from which the datatype `type` is derived.

In the case of an enumeration type, the return value is an integer type.

Parameters:

hid_t `type`

Datatype identifier for the derived datatype.

Returns:

Returns the datatype identifier for the base datatype if successful; otherwise returns a negative value.

Name: H5Tget_order

Signature:

H5T_order_t `H5Tget_order`(*hid_t* `type_id`)

Purpose:

Returns the byte order of an atomic datatype.

Description:

`H5Tget_order` returns the byte order of an atomic datatype.

Possible return values are:

`H5T_ORDER_LE` (`0`)

Little endian byte ordering (default).

`H5T_ORDER_BE` (`1`)

Big endian byte ordering.

`H5T_ORDER_VAX` (`2`)

VAX mixed byte ordering (not currently supported).

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a byte order constant if successful; otherwise `H5T_ORDER_ERROR` (-1).

Name: H5Tset_order

Signature:

herr_t `H5Tset_order`(*hid_t* `type_id`, *H5T_order_t*`order`)

Purpose:

Sets the byte ordering of an atomic datatype.

Description:

`H5Tset_order` sets the byte ordering of an atomic datatype. Byte orderings currently supported are:

H5T_ORDER_LE (`0`)

Little-endian byte ordering (default).

H5T_ORDER_BE (`1`)

Big-endian byte ordering.

H5T_ORDER_VAX (`2`)

VAX mixed byte ordering (not currently supported).

Parameters:

hid_t `type_id`

Identifier of datatype to set.

H5T_order_t `order`

Byte ordering constant.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_precision

Signature:

size_t `H5Tget_precision`(*hid_t* `type_id`)

Purpose:

Returns the precision of an atomic datatype.

Description:

`H5Tget_precision` returns the precision of an atomic datatype. The precision is the number of significant bits which, unless padding is present, is 8 times larger than the value returned by H5Tget_size().

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns the number of significant bits if successful; otherwise 0.

Name: H5Tset_precision

Signature:

herr_t `H5Tset_precision`(*hid_t* `type_id`, *size_t*`precision`)

Purpose:

Sets the precision of an atomic datatype.

Description:

`H5Tset_precision` sets the precision of an atomic datatype. The precision is the number of significant bits which, unless padding is present, is 8 times larger than the value returned by H5Tget_size().

If the precision is increased then the offset is decreased and then the size is increased to insure that significant bits do not "hang over" the edge of the datatype.

Changing the precision of an H5T_STRING automatically changes the size as well. The precision must be a multiple of 8.

When decreasing the precision of a floating point type, set the locations and sizes of the sign, mantissa, and exponent fields first.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

size_t `precision`

Number of bits of precision for datatype.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_offset

Signature:

size_t `H5Tget_offset`(*hid_t* `type_id`)

Purpose:

Retrieves the bit offset of the first significant bit.

Description:

`H5Tget_offset` retrieves the bit offset of the first significant bit. The signficant bits of an atomic datum can be offset from the beginning of the memory for that datum by an amount of padding. The ‘offset’ property specifies the number of bits of padding that appear to the "right of" the value. That is, if we have a 32-bit datum with 16-bits of precision having the value 0x1122 then it will be layed out in memory as (from small byte address toward larger byte addresses):

Byte Position	**Big-Endian Offset=0**	**Big-Endian Offset=16**	**Little-Endian Offset=0**	**Little-Endian Offset=16**
0:	[pad]	[0x11]	[0x22]	[pad]
1:	[pad]	[0x22]	[0x11]	[pad]
2:	[0x11]	[pad]	[pad]	[0x22]
3:	[0x22]	[pad]	[pad]	[0x11]

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a positive offset value if successful; otherwise 0.

Name: H5Tset_offset

Signature:

herr_t `H5Tset_offset`(*hid_t* `type_id`, *size_t* `offset`)

Purpose:

Sets the bit offset of the first significant bit.

Description:

`H5Tset_offset` sets the bit offset of the first significant bit. The signficant bits of an atomic datum can be offset from the beginning of the memory for that datum by an amount of padding. The ‘offset’ property specifies the number of bits of padding that appear to the "right of" the value. That is, if we have a 32-bit datum with 16-bits of precision having the value 0x1122 then it will be layed out in memory as (from small byte address toward larger byte addresses):

Byte Position	Big-Endian Offset=0	Big-Endian Offset=16	Little-Endian Offset=0	Little-Endian Offset=16
0:	[pad]	[0x11]	[0x22]	[pad]
1:	[pad]	[0x22]	[0x11]	[pad]
2:	[0x11]	[pad]	[pad]	[0x22]
3:	[0x22]	[pad]	[pad]	[0x11]

If the offset is incremented then the total size is incremented also if necessary to prevent significant bits of the value from hanging over the edge of the datatype.

The offset of an H5T_STRING cannot be set to anything but zero.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

size_t `offset`

Offset of first significant bit.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_pad

Signature:

herr_t `H5Tget_pad`(*hid_t* `type_id`, *H5T_pad_t* * `lsb`, *H5T_pad_t* * `msb`)

Purpose:

Retrieves the padding type of the least and most-significant bit padding.

Description:

`H5Tget_pad` retrieves the padding type of the least and most-significant bit padding. Valid types are:

H5T_PAD_ZERO (`0`)

Set background to zeros.

H5T_PAD_ONE (`1`)

Set background to ones.

H5T_PAD_BACKGROUND (`2`)

Leave background alone.

Parameters:

hid_t `type_id`

IN: Identifier of datatype to query.

H5T_pad_t * `lsb`

OUT: Pointer to location to return least-significant bit padding type.

H5T_pad_t * `msb`

OUT: Pointer to location to return most-significant bit padding type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tset_pad

Signature:

herr_t `H5Tset_pad`(*hid_t* `type_id`, *H5T_pad_t* `lsb`, *H5T_pad_t* `msb`)

Purpose:

Sets the least and most-significant bits padding types.

Description:

`H5Tset_pad` sets the least and most-significant bits padding types.

H5T_PAD_ZERO (`0`)

Set background to zeros.

H5T_PAD_ONE (`1`)

Set background to ones.

H5T_PAD_BACKGROUND (`2`)

Leave background alone.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

H5T_pad_t `lsb`

Padding type for least-significant bits.

H5T_pad_t `msb`

Padding type for most-significant bits.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_sign

Signature:

H5T_sign_t `H5Tget_sign`(*hid_t* `type_id`)

Purpose:

Retrieves the sign type for an integer type.

Description:

`H5Tget_sign` retrieves the sign type for an integer type. Valid types are:

H5T_SGN_NONE (`0`)

Unsigned integer type.

H5T_SGN_2 (`1`)

Two's complement signed integer type.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a valid sign type if successful; otherwise `H5T_SGN_ERROR` (-1).

Name: H5Tset_sign

Signature:

herr_t `H5Tset_sign`(*hid_t* `type_id`, *H5T_sign_t* `sign`)

Purpose:

Sets the sign proprety for an integer type.

Description:

`H5Tset_sign` sets the sign proprety for an integer type.

H5T_SGN_NONE (`0`)

Unsigned integer type.

H5T_SGN_2 (`1`)

Two's complement signed integer type.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

H5T_sign_t `sign`

Sign type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_fields

Signature:

herr_t `H5Tget_fields`(*hid_t* `type_id`, *size_t* * `epos`, *size_t* * `esize`, *size_t* * `mpos`, *size_t* * `msize`)

Purpose:

Retrieves floating point datatype bit field information.

Description:

`H5Tget_fields` retrieves information about the locations of the various bit fields of a floating point datatype. The field positions are bit positions in the significant region of the datatype. Bits are numbered with the least significant bit number zero. Any (or even all) of the arguments can be null pointers.

Parameters:

hid_t `type_id`

IN: Identifier of datatype to query.

size_t * `epos`

OUT: Pointer to location to return exponent bit-position.

size_t * `esize`

OUT: Pointer to location to return size of exponent in bits.

size_t * `mpos`

OUT: Pointer to location to return mantissa bit-position.

size_t * `msize`

OUT: Pointer to location to return size of mantissa in bits.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tset_fields

Signature:

herr_t `H5Tset_fields`(*hid_t* `type_id`, *size_t* `epos`, *size_t* `esize`, *size_t* `mpos`, *size_t* `msize`)

Purpose:

Sets locations and sizes of floating point bit fields.

Description:

`H5Tset_fields` sets the locations and sizes of the various floating point bit fields. The field positions are bit positions in the significant region of the datatype. Bits are numbered with the least significant bit number zero.

Fields are not allowed to extend beyond the number of bits of precision, nor are they allowed to overlap with one another.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

size_t `epos`

Exponent bit position.

size_t `esize`

Size of exponent in bits.

size_t `mpos`

Mantissa bit position.

size_t `msize`

Size of mantissa in bits.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_ebias

Signature:

size_t `H5Tget_ebias`(*hid_t* `type_id`)

Purpose:

Retrieves the exponent bias of a floating-point type.

Description:

`H5Tget_ebias` retrieves the exponent bias of a floating-point type.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns the bias if successful; otherwise 0.

Name: H5Tset_ebias

Signature:

herr_t `H5Tset_ebias`(*hid_t* `type_id`, *size_t* `ebias`)

Purpose:

Sets the exponent bias of a floating-point type.

Description:

`H5Tset_ebias` sets the exponent bias of a floating-point type.

Parameters:

hid_t `type_id`

Identifier of datatype to set.

size_t `ebias`

Exponent bias value.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_norm

Signature:

H5T_norm_t `H5Tget_norm`(*hid_t* `type_id`)

Purpose:

Retrieves mantissa normalization of a floating-point datatype.

Description:

`H5Tget_norm` retrieves the mantissa normalization of a floating-point datatype. Valid normalization types are:

H5T_NORM_IMPLIED (`0`)

MSB of mantissa is not stored, always 1

H5T_NORM_MSBSET (`1`)

MSB of mantissa is always 1

H5T_NORM_NONE (`2`)

Mantissa is not normalized

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a valid normalization type if successful; otherwise `H5T_NORM_ERROR` (-1).

Name: H5Tset_norm

Signature:

herr_t `H5Tset_norm`(*hid_t* `type_id`, *H5T_norm_t* `norm`)

Purpose:

Sets the mantissa normalization of a floating-point datatype.

Description:

`H5Tset_norm` sets the mantissa normalization of a floating-point datatype. Valid normalization types are:

H5T_NORM_IMPLIED (`0`)

MSB of mantissa is not stored, always 1

H5T_NORM_MSBSET (`1`)

MSB of mantissa is always 1

H5T_NORM_NONE (`2`)

Mantissa is not normalized

Parameters:

hid_t `type_id`

Identifier of datatype to set.

H5T_norm_t `norm`

Mantissa normalization type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_inpad

Signature:

H5T_pad_t `H5Tget_inpad`(*hid_t* `type_id`)

Purpose:

Retrieves the internal padding type for unused bits in floating-point datatypes.

Description:

`H5Tget_inpad` retrieves the internal padding type for unused bits in floating-point datatypes. Valid padding types are:

H5T_PAD_ZERO (`0`)

Set background to zeros.

H5T_PAD_ONE (`1`)

Set background to ones.

H5T_PAD_BACKGROUND (`2`)

Leave background alone.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a valid padding type if successful; otherwise `H5T_PAD_ERROR` (-1).

Name: H5Tset_inpad

Signature:

herr_t `H5Tset_inpad`(*hid_t* `type_id`, *H5T_pad_t* `inpad`)

Purpose:

Fills unused internal floating point bits.

Description:

If any internal bits of a floating point type are unused (that is, those significant bits which are not part of the sign, exponent, or mantissa), then `H5Tset_inpad` will be filled according to the value of the padding value property `inpad`. Valid padding types are:

H5T_PAD_ZERO (`0`)

Set background to zeros.

H5T_PAD_ONE (`1`)

Set background to ones.

H5T_PAD_BACKGROUND (`2`)

Leave background alone.

Parameters:

hid_t `type_id`

Identifier of datatype to modify.

H5T_pad_t `pad`

Padding type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_cset

Signature:

H5T_cset_t `H5Tget_cset`(*hid_t* `type_id`)

Purpose:

Retrieves the character set type of a string datatype.

Description:

`H5Tget_cset` retrieves the character set type of a string datatype. Valid character set types are:

H5T_CSET_ASCII (`0`)

Character set is US ASCII

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a valid character set type if successful; otherwise `H5T_CSET_ERROR` (-1).

Name: H5Tset_cset

Signature:

herr_t `H5Tset_cset`(*hid_t* `type_id`, *H5T_cset_t* `cset`)

Purpose:

Sets character set to be used.

Description:

`H5Tset_cset` the character set to be used.

HDF5 is able to distinguish between character sets of different nationalities and to convert between them to the extent possible. Valid character set types are:

H5T_CSET_ASCII (`0`)

Character set is US ASCII.

Parameters:

hid_t `type_id`

Identifier of datatype to modify.

H5T_cset_t `cset`

Character set type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_strpad

Signature:

H5T_str_t `H5Tget_strpad`(*hid_t* `type_id`)

Purpose:

Retrieves the string padding method for a string datatype.

Description:

`H5Tget_strpad` retrieves the string padding method for a string datatype. Valid string padding types are:

H5T_STR_NULL (`0`)

Pad with zeros (as C does)

H5T_STR_SPACE (`1`)

Pad with spaces (as FORTRAN does)

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns a valid string padding type if successful; otherwise `H5T_STR_ERROR` (-1).

Name: H5Tset_strpad

Signature:

herr_t `H5Tset_strpad`(*hid_t* `type_id`, *H5T_str_t* `strpad`)

Purpose:

Defines the storage mechanism for character strings.

Description:

The method used to store character strings differs with the programming language: C usually null terminates strings while Fortran left-justifies and space-pads strings. `H5Tset_strpad` defines the storage mechanism for the string. Valid string padding values are:

H5T_STR_NULL (`0`)

Pad with zeros (as C does)

H5T_STR_SPACE (`1`)

Pad with spaces (as FORTRAN does)

Parameters:

hid_t `type_id`

Identifier of datatype to modify.

H5T_str_t `strpad`

String padding type.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_nmembers

Signature:

int `H5Tget_nmembers`(*hid_t* `type_id`)

Purpose:

Retrieves the number of fields in a compound datatype.

Description:

`H5Tget_nmembers` retrieves the number of fields a compound datatype has.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

Returns:

Returns number of members datatype has if successful; otherwise returns a negative value.

Name: H5Tget_member_name

Signature:

char * `H5Tget_member_name`(*hid_t* `type_id`, *int* `field_idx`)

Purpose:

Retrieves the name of a field of a compound datatype.

Description:

`H5Tget_member_name` retrieves the name of a field of a compound datatype. Fields are stored in no particular order, with indexes 0 through N-1, where N is the value returned by `H5Tget_nmembers()`. The name of the field is allocated with `malloc()` and the caller is responsible for freeing the memory used by the name.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

int `field_idx`

Field index (0-based) of the field name to retrieve.

Returns:

Returns a valid pointer if successful; otherwise NULL.

Name: H5Tget_member_offset

Signature:

size_t `H5Tget_member_offset`(*hid_t* `type_id`, *int* `memb_no`)

Purpose:

Retrieves the offset of a field of a compound datatype.

Description:

`H5Tget_member_offset` retrieves the byte offset of the beginning of a field within a compound datatype with respect to the beginning of the compound data type datum.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

int `memb_no`

Number of the field whose offset is requested.

Returns:

Returns the byte offset of the field if successful; otherwise returns `0` (zero). Note that zero is a valid offset and that this function will fail only if a call to `H5Tget_member_dims()` fails with the same arguments.

Name: H5Tget_member_dims

Signature:

int `H5Tget_member_dims`(*hid_t* `type_id`, *int* `field_idx`, *size_t* *`dims`, *int* *`perm`)

Purpose:

Returns the dimensionality of the field.

Description:

`H5Tget_member_dims` returns the dimensionality of the field. The dimensions and permuation vector are returned through arguments `dims` and `perm`, both arrays of at least four elements. Either (or even both) may be null pointers.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

int `field_idx`

Field index (0-based) of the field `dims` to retrieve.

size_t * `dims`

Pointer to buffer to store the dimensions of the field.

int * `perm`

Pointer to buffer to store the permutation vector of the field.

Returns:

Returns the number of dimensions, a number from 0 to 4, if successful. Otherwise returns a negative value.

Name: H5Tget_member_type

Signature:

hid_t `H5Tget_member_type`(*hid_t* `type_id`, *int* `field_idx`)

Purpose:

Returns the datatype of the specified member.

Description:

`H5Tget_member_type` returns the datatype of the specified member. The caller should invoke H5Tclose() to release resources associated with the type.

Parameters:

hid_t `type_id`

Identifier of datatype to query.

int `field_idx`

Field index (0-based) of the field type to retrieve.

Returns:

Returns the identifier of a copy of the datatype of the field if successful; otherwise returns a negative value.

Name: H5Tinsert

Signature:

herr_t `H5Tinsert`(*hid_t* `type_id`, *const char* * `name`, *size_t* `offset`, *hid_t* `field_id`)

Purpose:

Adds a new member to a compound datatype.

Description:

`H5Tinsert` adds another member to the compound datatype `type_id`. The new member has a `name` which must be unique within the compound datatype. The `offset` argument defines the start of the member in an instance of the compound datatype, and `field_id` is the datatype identifier of the new member.

Note: Members of a compound datatype do not have to be atomic datatypes; a compound datatype can have a member which is a compound datatype.

Parameters:

hid_t `type_id`

Identifier of compound datatype to modify.

const char * `name`

Name of the field to insert.

size_t `offset`

Offset in memory structure of the field to insert.

hid_t `field_id`

Datatype identifier of the field to insert.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tpack

Signature:

herr_t `H5Tpack`(*hid_t* `type_id`)

Purpose:

Recursively removes padding from within a compound datatype.

Description:

`H5Tpack` recursively removes padding from within a compound datatype to make it more efficient (space-wise) to store that data.

Parameters:

hid_t `type_id`

Identifier of datatype to modify.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tregister

Signature:

herr_t `H5Tregister`(*H5T_pers_t* `pers`, *const char* * `name`, *hid_t* `src_id`, *hid_t* `dst_id`, *H5T_conv_t* `func`)

Purpose:

Registers a conversion function.

Description:

`H5Tregister` registers a hard or soft conversion function for a datatype conversion path.

The parameter `pers` indicates whether a conversion function is `HARD` or `SOFT`.

A conversion path can have only one hard function. When `pers` is `HARD`, `func` replaces any previous hard function. If `pers` is `HARD` and `func` is the null pointer, then any hard function registered for this path is removed.

When `pers` is `SOFT`, `H5Tregister` adds the function to the end of the master soft list and replaces the soft function in all applicable existing conversion paths. Soft functions are used when determining which conversion function is appropriate for this path.

The `name` is used only for debugging and should be a short identifier for the function.

The path is specified by the source and destination datatypes `src_id` and `dst_id`. For soft conversion functions, only the class of these types is important.

The type of the conversion function pointer is declared as:

`typedef` *herr_t* (`*H5T_conv_t`) (*hid_t* `src_id`, *hid_t* `dst_id`, *H5T_cdata_t* `*cdata`, *size_t* `nelmts`, *void* `*buf`, *void* `*bkg`);

Parameters:

H5T_pers_t `pers`

`HARD` for hard conversion functions; `SOFT` for soft conversion functions.

const char * `name`

Name displayed in diagnostic output.

hid_t `src_id`

Identifier of source datatype.

hid_t `dst_id`

Identifier of destination datatype.

H5T_conv_t `func`

Function to convert between source and destination datatypes.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tunregister

Signature:

herr_t `H5Tunregister`(*H5T_conv_t* `func`)

Purpose:

Removes a conversion function from all conversion paths.

Description:

`H5Tunregister` removes a conversion function from all conversion paths.

The conversion function pointer type declaration is described in `H5Tregister`.

Parameters:

H5T_conv_t `func`

Function to remove from conversion paths.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tenum_create

Signature:

hid_t `H5Tenum_create(`*hid_t* `parent_id`)

Purpose:

Creates a new enumeration datatype.

Description:

`H5Tenum_create` creates a new enumeration datatype based on the specified base datatype, `parent_id`, which must be an integer type.

Parameters:

hid_t `parent_id`

IN: Datatype identifier for the base datatype.

Returns:

Returns the datatype identifier for the new enumeration datatype if successful; otherwise returns a negative value.

Name: H5Tenum_insert

Signature:

herr_t `H5Tenum_insert(`*hid_t* `type`, *const char* `*name`, *void* `*value`)

Purpose:

Inserts a new enumeration datatype member.

Description:

`H5Tenum_insert` inserts a new enumeration datatype member into an enumeration datatype.

`type` is the enumeration datatype, `name` is the name of the new member, and `value` points to the value of the new member.

`name` and `value` must both be unique within `type`.

`value` points to data which is of the datatype defined when the enumeration datatype was created.

Parameters:

hid_t `type`

IN: Datatype identifier for the enumeration datatype.

const char *`name`

IN: Name of the new member.

void *`value`

IN: Pointer to the value of the new member.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tenum_nameof

Signature:

herr_t `H5Tenum_nameof(`*hid_t* `type` *void* *`value`, *char* *`name`, *size_t* `size`)

Purpose:

Returns the symbol name corresponding to a specified member of an enumeration datatype.

Description:

`H5Tenum_nameof` finds the symbol name that corresponds to the specified `value` of the enumeration datatype `type`.

At most `size` characters of the symbol name are copied into the `name` buffer. If the entire symbol name and null terminator do not fit in the `name` buffer, then as many characters as possible are copied (not null terminated) and the function fails.

Parameters:

hid_t `type`

IN: Enumeration datatype identifier.

void *`value`,

IN: Value of the enumeration datatype.

char *`name`,

OUT: Buffer for output of the symbol name.

size_t `size`

IN: Anticipated size of the symbol name, in bytes (characters).

Returns:

Returns a non-negative value if successful. Otherwise returns a negative value and, if `size` allows it, the first character of `name` is set to `NULL`.

Name: H5Tenum_valueof

Signature:

herr_t `H5Tenum_valueof`(*hid_t* `type` *char* *`name`, *void* *`value`)

Purpose:

Returns the value corresponding to a specified member of an enumeration datatype.

Description:

`H5Tenum_valueof` finds the value that corresponds to the specified `name` of the enumeration datatype `type`.

The `value` argument should be at least as large as the value of `H5Tget_size(type)` in order to hold the result.

Parameters:

hid_t `type`

IN: Enumeration datatype identifier.

const char *`name`,

IN: Symbol name of the enumeration datatype.

void *`value`,

OUT: Buffer for output of the value of the enumeration datatype.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_member_value

Signature:

hid_t `H5Tget_member_value`(*hid_t* `type` *int* `memb_no`, *void* *`value`)

Purpose:

Returns the value of an enumeration datatype member.

Description:

`H5Tget_member_value` returns the value of the enumeration datatype member `memb_no`.

The member value is returned in a user-supplied buffer pointed to by `value`.

Parameters:

hid_t `type`

IN: Datatype identifier for the enumeration datatype.

int `memb_no`,

IN: Number of the enumeration datatype member.

void `*value`

OUT: Pointer to a buffer for output of the value of the enumeration datatype member.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tset_tag

Signature:

herr_t `H5Tset_tag`(*hid_t* `type_id` *const char* `*tag`)

Purpose:

Tags an opaque datatype.

Description:

`H5Tset_tag` tags an opaque datatype `type_id` with a unique ASCII identifier `tag`.

Parameters:

hid_t `type_id`

IN: Datatype identifier for the opaque datatype to be tagged.

const char `*tag`

IN: Unique ASCII string with which the opaque datatype is to be tagged.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Tget_tag

Signature:

char *`H5Tget_tag`(*hid_t* `type_id`)

Purpose:

Gets the tag associated with an opaque datatype.

Description:

`H5Tget_tag` returns the tag associated with the opaque datatype `type_id`.

The tag is returned via a pointer to an allocated string, which the caller must free.

Parameters:

hid_t `type_id`

Datatype identifier for the opaque datatype.

Returns:

Returns a pointer to an allocated string if successful; otherwise returns `NULL`.

Name: H5Tclose

Signature:

herr_t `H5Tclose`(*hid_t* `type_id`)

Purpose:

Releases a datatype.

Description:

`H5Tclose` releases a datatype. Further access through the datatype identifier is illegal. Failure to release a datatype with this call will result in resource leaks.

Parameters:

hid_t `type_id`

Identifier of datatype to release.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Last modified: 20 October 1999

H5Z: Compression Interface

Compression API Functions

This function enable the user to configure a new compression method for the local environment.

- H5Zregister

HDF5 supports compression of raw data by compression methods built into the library or defined by an application. A compression method is associated with a dataset when the dataset is created and is applied independently to each storage chunk of the dataset. The dataset must use the `H5D_CHUNKED` storage layout.

The HDF5 library does not support compression for contiguous datasets because of the difficulty of implementing random access for partial I/O. Compact dataset compression is not supported because it would not produce significant results.

See the *Compression* section of the *HDF5 User's Guide* for further information.

Name: H5Zregister

Signature:

herr_t `H5Zregister`(*H5Z_method_t* `method`, *const char* *`name`, *H5Z_func_t*`cfunc`, *H5Z_func_t* `ufunc`)

Purpose:

Registers new compression and uncompression functions for a method specified by a method number.

Description:

`H5Zregister` registers new compression and uncompression functions for a method specified by a method number, `method`. `name` is used for debugging and may be the null pointer. Either or both of `cfunc` (the compression function) and `ufunc` (the uncompression method) may be null pointers.

The statistics associated with a method number are not reset by this function; they accumulate over the life of the library.

Parameters:

H5Z_method_t `method`

Number specifying compression method.

const char *`name`

Name associated with the method number.

H5Z_func_t `cfunc`

Compression method.

H5Z_func_t `ufunc`

Uncompression method.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Last modified: 30 October 1998

H5RA: Ragged Array Interface

Ragged Array API Functions

The H5RA Interface is strictly experimental at this time; the interface may change dramatically or support for ragged arrays may be unavailable in future in releases. As a result, future releases may be unable to retrieve data stored with this interface.

Use these functions at your own risk!
Do not create any archives using this interface!

These functions enable the user to store and retrieve data in ragged arrays.

- H5RAcreate
- H5RAopen
- H5RAclose
- H5RAwrite
- H5RAread

This version of the Ragged Array interface implements a two-dimensional array where each row of the array is a different length. It is intended for applications where the distribution of row lengths is such that most rows are near an average length with only a few rows that are significantly shorter or longer. The raw data is split among two datasets, `raw` and `over`: the `raw` dataset is a two-dimensional chunked dataset whose width is large enough to hold most of the rows while the `over` dataset is a heap that stores the ends of rows that overflow the first dataset. A third dataset, called `meta`, contains one record for each row and describes what elements, if any, overflow the `raw` dataset and where they are stored in the `over` dataset. All three datasets are contained in a single group whose name is the name of the ragged array.

Name: H5RAcreate

Signature:

hid_t `H5RAcreate`(*hid_t* `loc_id`, *const char* *`name`, *hid_t* `type_id`, *hid_t* `plist_id`)

Purpose:

Creates a ragged array.

Description:

`H5RAcreate` creates a new ragged array with the name specified in `name`. A ragged array is implemented as a group containing three datasets. The dataset `raw` is a fixed width dataset which will hold the majority of the data. The dataset `over` is a one dimensional heap which will hold the end of rows which are too long to fit in `raw` Finally, the `meta` dataset contains information about the `over` array. All elements of the ragged array are stored with the same data type.

The property list `plist_id` should contain information about chunking. The chunk width will determine the width of the `raw` dataset while the chunk length should be such that the total chunk size is reasonably large since I/O will be performed in units of chunks). If the `plist_id` does not have a chunk size defined (e.g., `H5P_DEFAULT`) then this function will fail.

Parameters:

hid_t `loc_id`

Location identifier of the dataset.

const char `*name`

The assigned name of the data set to be stored in the ragged array.

hid_t `type_id`

Data type identifier for the ragged array data.

hid_t `plist_id`

Property list of the dataset.

Returns:

Returns a ragged array identifier if successful; otherwise returns a negative value.

Name: H5RAopen

Signature:

hid_t `H5RAopen`(*hid_t* `loc_id`, *const char* `*name`)

Purpose:

Opens a ragged array.

Description:

`H5RAopen` opens an existing ragged array.

The name of the array, `name`, should be the same that was used when the array was created, i.e., the name of the group which implements the array.

Parameters:

hid_t `loc_id`

The location identifier of the dataset.

const char `*name`

The name of the ragged array.

Returns:

Returns a ragged array identifier if successful; otherwise returns a negative value.

Name: H5RAclose

Signature:

herr_t `H5RAclose`(*hid_t* `array_id`)

Purpose:

Closes a ragged array.

Description:

`H5RAclose` closes the ragged array specified with the array identifier `array_id`.

Parameters:

hid_t `array_id`

The array identifier for the ragged array to be closed.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5RAwrite

Signature:

herr_t `H5RAwrite`(*hid_t* `array_id`, *hssize_t* `start_row`, *hsize_t* `nrows`, *hid_t* `type_id`, *hsize_t* `size[/*nrows*/]`, *void* `*buf[/*nrows*/]`)

Purpose:

Writes to a ragged array.

Description:

`H5RAwrite` writes a contiguous set of rows to a ragged array beginning at row number `start_row` and continuing for `nrows` rows.

Each row of the ragged array contains `size[]` elements of type `type_id` and each row is stored in a buffer pointed to by `buf[]`.

Datatype conversion takes place at the time of a read or write and is automatic. See the "Data Conversion" section of *The Data Type Interface (H5T)* in the *HDF5 User's Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `array_id`

Array identifier for the ragged array to be written to.

hssize_t `start_row`

Row at which the write will start.

hsize_t `nrows`

Number of rows to be written

hid_t `type_id`

Data type identifier for the data to be written.

hsize_t `size[/*nrows*/]`

Lengths of the rows to be written.

void `*buf[/*nrows*/]`

Pointers to buffers containing the data to be written.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5RAread

Signature:

herr_t `H5RAread`(*hid_t* `array_id`, *hssize_t* `start_row`, *hsize_t* `nrows`, *hid_t* `type_id`, *hsize_t* `size[/*nrows*/]`, *void* `*buf[/*nrows*/]`)

Purpose:

Description:

`H5RAread` reads the contents of one or more rows of the ragged array pointed to by `array_id`.

The rows to be read begin at row `start_row` and continue for `nrows` rows.

All raw data is converted to type `type_id`.

The caller must allocate the `size[]` and `buf[]` arrays.

Memory for the data can be allocated by either the caller or the library. In the former case, the caller should initialize the `buf[]` array with pointers to valid memory and the `size[]` array with the lengths of the buffers. In the latter case, the caller should initialize `buf[]` with null pointers (the input value of `size[]` is irrelevant in this case) and the library will allocate memory for each row by calling `malloc()`.

Datatype conversion takes place at the time of a read or write and is automatic. See the Data Conversion section of *The Data Type Interface (H5T)* in the *HDF5 User's Guide* for a discussion of data conversion, including the range of conversions currently supported by the HDF5 libraries.

Parameters:

hid_t `array_id`

Array identifier for the ragged array to be read from.

hssize_t `start_row`

Row at which the read will start.

hsize_t `nrows`

Number of rows to be read

hid_t `type_id`

Data type identifier for the data to be read.

hsize_t `size[/*nrows*/]`

Lengths of the rows to be read.

void `*buf[/*nrows*/]`

Pointers to buffers into which the data is to be read.

Returns:

Returns a non-negative value if successful. The values of the `size[]` array will be the true length of each row. If a row is longer than the caller-allocated length, then `size[]` will contain the true length of the row although not all elements of that row will be stored in the buffer.

Returns a negative value on failure. The `buf[]` array will contain it's original pointers (null or otherwise), although the caller-supplied buffers may have been modified. The `size[]` array may also have been modified.

Last modified: 30 October 1998

HDF5 Tools

HDF5 Tool Interfaces

These tools enable the user to examine HDF5 files interactively.

- h5dump -- A tool for displaying HDF5 file contents
- h5ls -- A tool for listing specified features of HDF5 file contents
- h5repart -- A tool for repartitioning a file, creating a family of files
- h5toh4 -- A tool for converting an HDF5 file to an HDF4 file.

Tool Name: h5dump

Syntax:

`h5dump [-h] [-bb] [-header] [-a` *names*`] [-d` *names*`] [-g` *names*`] [-l` *names*`] [-t` *names*`]` *file*

Purpose:

Displays HDF5 file contents.

Description:

`h5dump` enables the user to interactively examine the contents of an HDF5 file and dump those contents, in human readable form, to an ASCII file.

`h5dump` displays HDF5 file content on standard output. It may display the content of the whole HDF5 file or selected objects, which can be groups, datasets, links, attributes, or data types.

The `-header` option displays object header information only.

Names are the absolute names of the objects. `h5dump` displays objects in the order same as the command order. If a name does not start with a slash, `h5dump` begins searching for the specified object starting at the root group.

If an object is hard linked with multiple names, `h5dump` displays the content of the object in the first occurrence. Only the link information is displayed in later occurrences.

`h5dump` assigns a name for any unnamed data type in the form of #*oid1*`:`*oid2*, where *oid1* and *oid2* are the object identifiers assigned by the library. The unnamed types are displayed within the root group.

Data types are displayed with standard type names. For example, if a data set is created with `H5T_NATIVE_INT` type and the standard type name for integer on that machine is `H5T_STD_I32BE`, `h5dump` displays `H5T_STD_I32BE` as the type of the data set.

The `h5dump` output is described in detail in the *DDL for HDF5*, the *Data Description Language* document.

Options and Parameters:

`-h`

Prints information on this command.

`-bb`

Displays the contents of the boot block. The default is not to display.

`-header`

Displays header information only; no data is displayed.

`-a` *names*

Displays the specified attribute(s).

`-d` *names*

Displays the specified dataset(s).

`-g` *names*

Displays the specified group(s) and all the members.

`-l` *names*

Displays the values of the specified soft link(s).

`-t` *names*

Displays the specified named data type(s).

file

The file to be examined.

Current Status:

The current version of `h5dump` displays the following information:

- Group
    - group attribute (see Attribute)
    - group member
- Dataset
    - dataset attribute (see Attribute)
    - dataset type (see Data type)
    - dataset space (see Data space)
    - dataset data

- Attribute
    - attribute type (see Data type)
    - attribute space (see Data space)
    - attribute data
- Data type
    - integer type

      - H5T_STD_I8BE, H5T_STD_I8LE, H5T_STD_I16BE, ...
  - floating point type

    - H5T_IEEE_F32BE, H5T_IEEE_F32LE, H5T_IEEE_F64BE, ...
  - string type
  - compound type

    - named, unnamed and transient compound type

    - integer, floating or string type member
  - reference type

    - object references

    - data regions
  - enum type
- Data space
  - scalar and simple space
- Soft link
- Hard link
- Loop detection

See Also:

HDF5 Data Description Language syntax (*DDL for HDF5*)

Tool Name: h5ls

Syntax:

`h5ls [`*options*`]` *file* `[`*objects...*`]`

Purpose:

Prints information about a file or dataset.

Description:

`h5ls` prints selected information about file objects in the specified format.

Options and Parameters:

`-h` or `-?` or `--help`

Print a usage message and exit.

`-d` or `--dump`

Print the values of datasets.

`-w`*N* or `--width=`*N*

Set the number of columns of output.

`-v` or `--verbose`

Generate more verbose output.

`-V` or `--version`

Print version number and exit.

file

The file name may include a printf(3C) integer format such as `%%05d` to open a file family.

objects

The names of zero or more objects about which information should be displayed. If a group is mentioned then information about each of its members is displayed. If no object names are specified then information about all of the objects in the root group is displayed.

Tool Name: h5repart

Syntax:

`h5repart [-v] [-V] [-[b|m]`*N*`[g|m|k]]` *source_file dest_file*

Purpose:

Repartitions a file or family of files.

Description:

`h5repart` splits a single file into a family of files, joins a family of files into a single file, or copies one family of files to another while changing the size of the family members. `h5repart` can also be used to copy a single file to a single file with holes.

Sizes associated with the `-b` and `-m` options may be suffixed with `g` for gigabytes, `m` for megabytes, or `k` for kilobytes.

File family names include an integer `printf` format such as `%d`.

Options and Parameters:

`-v`

Produce verbose output.

`-V`

Print a version number and exit.

`-b`*N*

The I/O block size, defaults to 1kB

`-m`*N*

The destination member size or 1GB

source_file

The name of the source file

dest_file

The name of the destination files

Tool Name: h5toh4

Syntax:

`h5toh4 -h`
`h5toh4` *h5file h4file*
`h5toh4` *h5file*
`h5toh4 -m` *h5file1 h5file2 h5file3 ...*

Purpose:

Converts an HDF5 file into an HDF4 file.

Description:

`h5toh4` is an HDF5 utility which reads an HDF5 file, *h5file*, and converts all supported objects and pathways to produce an HDF4 file, *h4file*. If *h4file* already exists, it will be replaced.

If only one file name is given, the name must end in `.h5` and is assumed to represent the HDF5 input file. `h5toh4` replaces the `.h5` suffix with `.hdf` to form the name of the resulting HDF4 file and proceeds as above. If a file with the name of the intended HDF4 file already exists, `h5toh4` exits with an error without changing the contents of any file.

The `-m` option allows multiple HDF5 file arguments. Each file name is treated the same as the single file name case above.

The `-h` option causes the following syntax summary to be displayed:

```
h5toh4 file.h5 file.hdf
h5toh4 file.h5
h5toh4 -m file1.h5 file2.h5 ...
```

The following HDF5 objects occurring in an HDF5 file are converted to HDF4 objects in the HDF4 file:

- HDF5 group objects are converted into HDF4 Vgroup objects. HDF5 hardlinks and softlinks pointing to objects are converted to HDF4 Vgroup references.

- HDF5 dataset objects of integer datatype are converted into HDF4 SDS objects. These datasets may have up to 32 fixed dimensions. The slowest varying dimension may be extendable. 8-bit, 16-bit, and 32-bit integer datatypes are supported.

- HDF5 dataset objects of floating point datatype are converted into HDF4 SDS objects. These datasets may have up to 32 fixed dimensions. The slowest varying dimension may be extendable. 32-bit and 64-bit floating point datatypes are supported.

- HDF5 dataset objects of single dimension and compound datatype are converted into HDF4 Vdata objects. The length of that single dimension may be fixed or extendable. The members of the compound datatype are constrained to be no more than rank 4.

- HDF5 dataset objects of single dimension and fixed length string datatype are converted into HDF4 Vdata objects. The HDF4 Vdata is a single field whose order is the length of the HDF5 string type. The number of records of the Vdata is the length of the single dimension which may be fixed or extendable.

Other objects are not converted and are not recorded in the resulting *h4file*.

Attributes associated with any of the supported HDF5 objects are carried over to the HDF4 objects. Attributes may be of integer, floating point, or fixed length string datatype and they may have up to 32 fixed dimensions.

All datatypes are converted to big-endian. Floating point datatypes are converted to IEEE format.

Options and Parameters:

`-h`

Displays a syntax summary.

`-m`

Converts multiple HDF5 files to multiple HDF4 files.

h5file

The HDF5 file to be converted.

h4file

The HDF4 file to be created.

Last modified: 29 April 1999

HDF5 Glossary

Release 1.2 , October 1999

Relationships among Terms

- atomic datatype
- attribute
- chunked layout
- chunking
- compound datatype
- contiguous layout
- dataset
- dataspace
- datatype
  - atomic
  - compound
  - enumeration
  - named
  - opaque
  - variable-length
- enumeration datatype
- file
  - group
  - path
  - root group
  - super block
- file access mode
- group
  - member
  - root group
- hard link
- hyperslab
- identifier
- link
  - hard
  - soft
- member
- name
- named datatype
- opaque datatype
- path
- property list
  - data transfer
  - dataset access
  - dataset creation
  - file access
  - file creation
- root group
- selection
  - hyperslab
- serialization
- soft link
- storage layout
  - chunked
  - chunking
  - contiguous
- super block
- variable-length datatype

atomic datatype

A datatype which cannot be decomposed into smaller units at the API level.

attribute

A small dataset that can be used to describe the nature and/or the intended usage of the object it is attached to.

chunked layout

The storage layout of a chunked dataset.

chunking

A storage layout where a dataset is partitioned into fixed-size multi-dimensional chunks. Chunking tends to improve performance and facilitates dataset extensibility.

compound datatype

A collection of one or more atomic types or small arrays of such types. Similar to a struct in C or a common block in Fortran.

contiguous layout

The storage layout of a dataset that is not chunked, so that the entire data portion of the dataset is stored in a single contiguous block.

data transfer property list

The data transfer property list is used to control various aspects of the I/O, such as caching hints or collective I/O information.

dataset

A multi-dimensional array of data elements, together with supporting metadata.

dataset access property list

A property list containing information on how a dataset is to be accessed.

dataset creation property list

A property list containing information on how raw data is organized on disk and how the raw data is compressed.

dataspace

An object that describes the dimensionality of the data array. A dataspace is either a regular N-dimensional array of data points, called a simple dataspace, or a more general collection of data points organized in another manner, called a complex dataspace.

datatype

An object that describes the storage format of the individual data points of a data set. There are two categories of datatypes: atomic and compound datatypes. An atomic type is a type which cannot be decomposed into smaller units at the API level. A compound datatype is a collection of one or more atomic types or small arrays of such types.

enumeration datatype

A one-to-one mapping between a set of symbols and a set of integer values, and an order is imposed on the symbols by their integer values. The symbols are passed between the application and library as character strings and all the values for a particular enumeration datatype are of the same integer type, which is not necessarily a native type.

file

A container for storing grouped collections of multi-dimensional arrays containing scientific data.

file access mode

Determines whether an existing file will be overwritten, opened for read-only access, or opened for read/write access. All newly created files are opened for both reading and writing.

file access property list

File access property lists are used to control different methods of performing I/O on files:

file creation property list

The property list used to control file metadata.

group

A structure containing zero or more HDF5 objects, together with supporting metadata. The two primary HDF5 objects are datasets and groups.

hard link

A direct association between a name and the object where both exist in a single HDF5 address space.

hyperslab

A portion of a dataset. A hyperslab selection can be a logically contiguous collection of points in a dataspace or a regular pattern of points or blocks in a dataspace.

identifier

A unique entity provided by the HDF5 library and used to access an HDF5 object, such as a file, goup, dataset, datatype, etc.

link

An association between a name and the object in an HDF5 file group.

member

A group or dataset that is in another dataset, *dataset A*, is a member of *dataset A*.

name

A slash-separated list of components that uniquely identifies an element of an HDF5 file. A name begins that begins with a slash is an absolute name which is accessed beginning with the root group of the file; all other names are relative names and the associated objects are accessed beginning with the current or specified group.

named datatype

A datatype that is named and stored in a file. Naming is permanent; a datatype cannot be changed after being named.

opaque datatype

A mechanism for describing data which cannot be otherwise described by HDF5. The only properties associated with opaque types are a size in bytes and an ASCII tag.

path

The slash-separated list of components that forms the name uniquely identifying an element of an HDF5 file.

property list

A collection of name/value pairs that can be passed to other HDF5 functions to control features that are typically unimportant or whose default values are usually used.

root group

The group that is the entry point to the group graph in an HDF5 file. Every HDF5 file has exactly one root group.

selection

A subset of a dataset or a dataspace, up to the entire dataset or dataspace.

serialization

The flattening of an *N*-dimensional data object into a 1-dimensional object so that, for example, the data object can be transmitted over the network as a 1-dimensional bitstream.

soft link

An indirect association between a name and an object in an HDF5 file group.

storage layout

The manner in which a dataset is stored, either contiguous or chunked, in the HDF5 file.

super block

A block of data containing the information required to portably access HDF5 files on multiple platforms, followed by information about the groups and datasets in the file. The super block contains information about the size of offsets, lengths of objects, the number of entries in group tables, and additional version information for the file.

variable-length datatype

A sequence of an existing datatype (atomic, variable-length (VL), or compound) which are not fixed in length from one dataset location to another.

Last modified: 18 October 1999