HDF5 User’s Guide

Release 1.6.6

August, 2007

The HDF Group (THG)
http://hdfgroup.org/

(Printed: August 2007)

Copyright Notice and License Terms for
HDF5 (Hierarchical Data Format 5) Software Library and Utilities

DISCLAIMER: THIS SOFTWARE IS PROVIDED BY THE HDF GROUP (THG) AND THE CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall THG or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Portions of HDF5 were developed with support from the University of California, Lawrence Livermore National Laboratory (UC LLNL). The following statement applies to those portions of the product and must be retained in any redistribution of source code, binaries, documentation, and/or accompanying materials:

> This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.
>
> **DISCLAIMER:** This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately- owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Table of Contents

Chapter 1
HDF5 Data Model

1. Introduction

1.1. Introduction and Definitions

The Hierarchical Data Format (HDF) implements a model for managing and storing data. The model includes an abstract data model and an abstract storage model (the data format), and libraries to implement the abstract model and to map the storage model to different storage mechanisms. The HDF5 library provides a programming interface to a concrete implementation of the abstract models. The library also implements a model of data transfer, i.e., efficient movement of data from one stored representation to another stored representation. Figure 1 illustrates the relationships between the models and implementations. This chapter explains these models in detail.

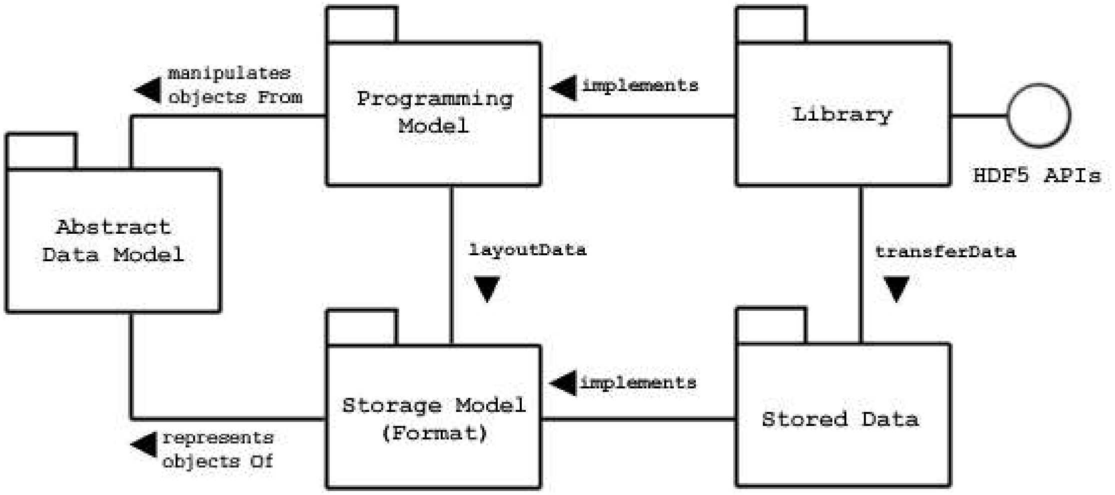


Figure 1

The *Abstract Data Model* is a conceptual model of data, data types, and data organization. The Abstract Data Model is independent of storage medium or programming environment. The *Storage Model* is a standard representation for the objects of the *Abstract Data Model*. The *HDF5 File Format Specification* defines the *Storage Model*.

The *Programming Model* is a model of the computing environment, which includes many platforms, from small single systems to large multiprocessors and clusters. The *Programming Model* manipulates (instantiates, populates, and retrieves) objects from the *Abstract Data Model*.

The *Library* is the concrete implementation of the *Programming Model*. The *Library* exports the HDF5 APIs as its interface. In addition to implementing the objects of the *Abstract Data Model*, the *Library* manages data transfers from one stored form to another (e.g., read from disk to memory, write from memory to disk, etc.).

The *Stored Data* is the concrete implementation of the *Storage Model*. The *Storage Model* is mapped to several storage mechanisms, including single disk files, multiple files (family of files), and memory representations.

The HDF5 Library is a C module that implements the *Programming Model* and *Abstract Data Model*. The HDF5 Library calls the Operating System or other Storage Management software (e.g., the MPI/IO Library) to store and retrieve persistent data. The HDF5 Library may also link to other software, such as filters for compression. The HDF5 Library is linked to an application program, which may be written in C, C++, Fortran 90, or Java. The application program implements problem specific algorithms and data structures, and calls the HDF5 Library to store and retrieve data. Figure 2 shows the dependencies of these modules.

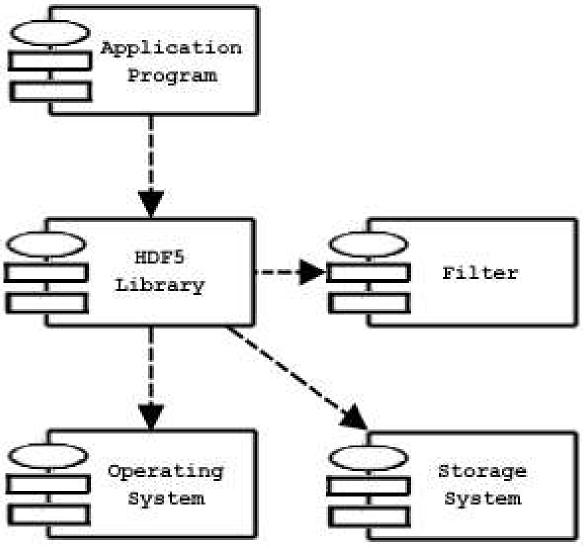


Figure 2

It is important to realize that each of the software components manages data using models and data structures that are appropriate to the component. When data is passed between layers-during storage or retrieval-it is transformed from one representation to another. Figure 3 suggests some of the kinds of data structures used in the different layers.

The *Application Program* uses data structures that represent the problem and algorithms, including variables, tables, arrays, and meshes, among other data structures. Obviously, an application might have quite a few different kinds of data structures, and different numbers and sizes of objects, depending on its design and function.

The HDF5 *Library* implements the objects of the HDF5 *Abstract Data Model*. These include Groups, Datasets, and Attributes and other objects as defined in this Chapter. The Application Program maps the application data structures to a hierarchy of HDF5 objects. Each application will create a mapping best suited to its purposes.

The objects of the HDF5 *Abstract Data Model* are mapped to the objects of the HDF5 *Storage Model*, and stored in a storage medium. The stored objects include header blocks, free lists, data blocks, B-trees, and other objects. Each Group, Dataset, etc. is stored as one or more header and data blocks, organized as defined in the *HDF5 File Format Specification*. The HDF5 *Library* can also use other libraries and modules, such as compression.

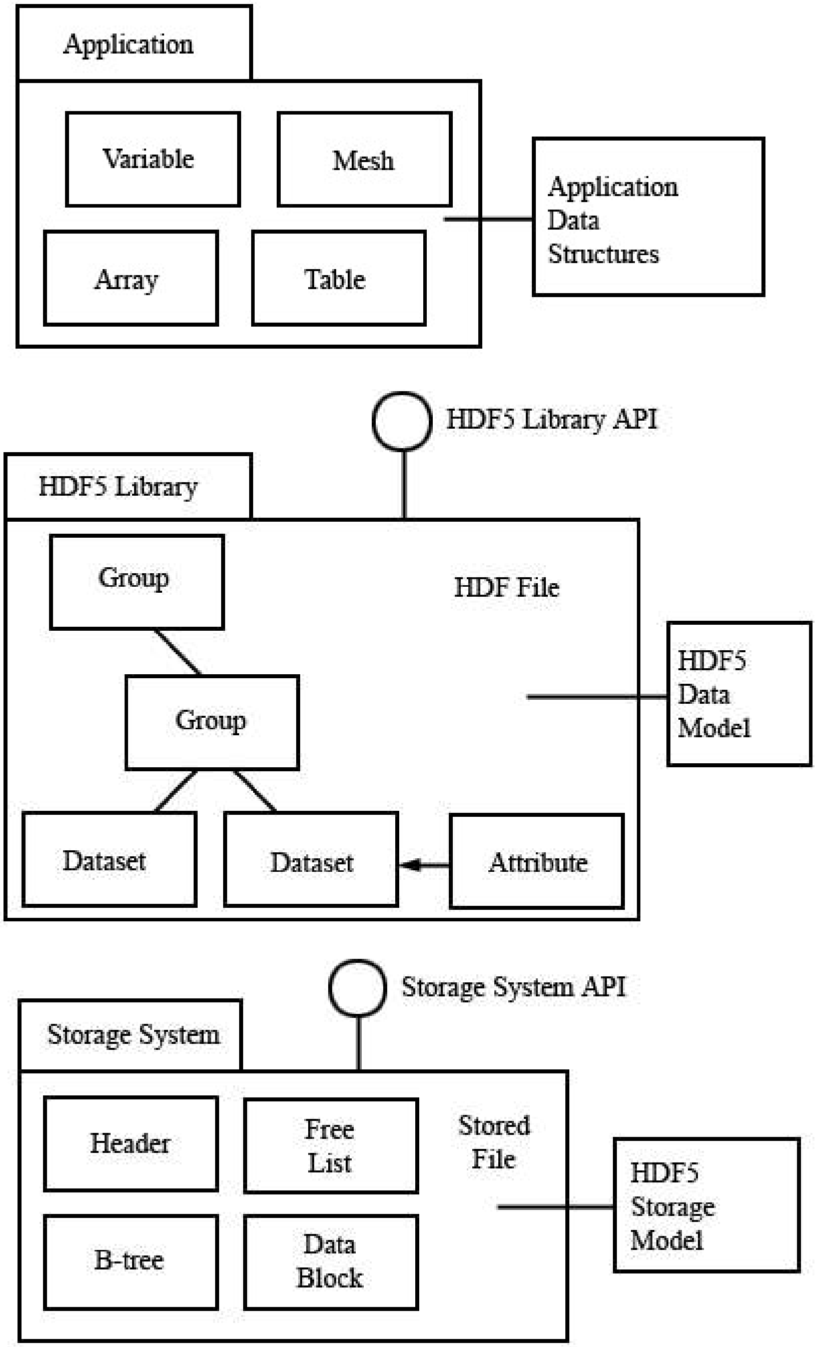


Figure 3

The important point to note is that there is not necessarily any simple correspondence between the objects of the *Application Program*, the *Abstract Data Model*, and those of the *Format Specification*. The organization of the data of *Application Program*, and how it is mapped to the HDF5 *Abstract Data Model* is up to the application developer. The Application Program only needs to deal with the *Library* and the *Abstract Data Model*. Most applications need not consider any details of the *HDF5 File Format Specification*, or the details of how objects of *Abstract Data Model* are translated to and from storage.

2. The Abstract Data Model

2.1. Purpose and Summary of the Abstract Data Model

The *Abstract Data Model* (ADM) defines concepts for defining and describing complex data stored in files. The HDF5 ADM is a very general model which is designed to conceptually cover many specific models of data. Many different kinds of data can be mapped to objects of the HDF5 ADM, and therefore stored and retrieved using HDF5. The ADM is not, however, a model of any particular problem or application domain. Users need to map their data to the concepts of the ADM.

The key concepts include:

- *File* - a contiguous string of bytes in a computer store (memory, disk, etc.). The bytes represent zero or more objects of the model.
- *Group* - a collection of objects (including groups).
- *Dataset* - a multidimensional array of Data Elements, with Attributes and other metadata.
- *Datatype* - a description of a specific class of data element, including its storage layout as a pattern of bits.
- *Dataspace* - a description of the dimensions of a multidimensional array.
- *Attribute* - a named data value associated with a group, dataset, or named datatype
- *Property List* - a collection of parameters controlling options in the library. Some properties are permanently stored as part of the object; others are transient and apply to a specific access. Each class of property list has specific properties.

2.2. Definitions

File

Abstractly, an HDF5 File is a container for an organized collection of objects. The objects are Groups and Datasets and other objects as defined below. The objects are organized as a rooted, directed graph. Every HDF5 file has at least one object, the root group (Figure 4). All objects are members of the root group or descendents of the root group.

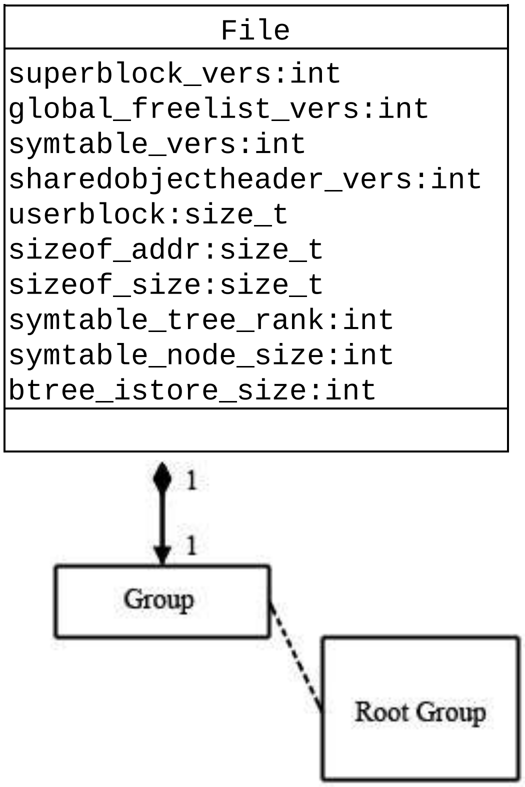


Figure 4

HDF5 objects have a unique identity *within a single HDF5 file*, and can be accessed only by its names within the hierarchy of the file. HDF5 objects in different files do not necessarily have unique identities, and it is not possible to access a permanent HDF5 object except through a file. See the section "The Structure of an HDF5 File" below for an explanation of the structure of the HDF5 file.

When the file is created, the *File Creation Properties* specify settings for the file. File Creation Properties include version information and parameters of global data structures. When the file is opened, the *File Access Properties* specify settings for the current access to the file. File Access Properties include parameters for storage drivers and parameters for caching and garbage collection. The *File Creation Properties* are permanent for the life of the file, the *File Access Properties* can be changed by closing and reopening the file.

An HDF5 file can be "mounted" as part of another HDF5 file. This is analogous to Unix File System mounts. The root of the mounted file is attached to a Group in the mounting file, and all the contents can be accessed as if the mounted file were part of the mounting file.

Group

An HDF5 *Group* is analogous to a file system directory. Abstractly, a Group contains zero or more objects, and every object must be a member of at least one Group. (The root Group is a special case; it may not be a member of any group.)

Group membership is actually implemented via *Link* objects (Figure 5). A Link object is owned by a Group and points to a *Named Object*. Each Link has a *name*, and each link points to exactly one object. Each Named Object has at least one and possibly many Links to it.

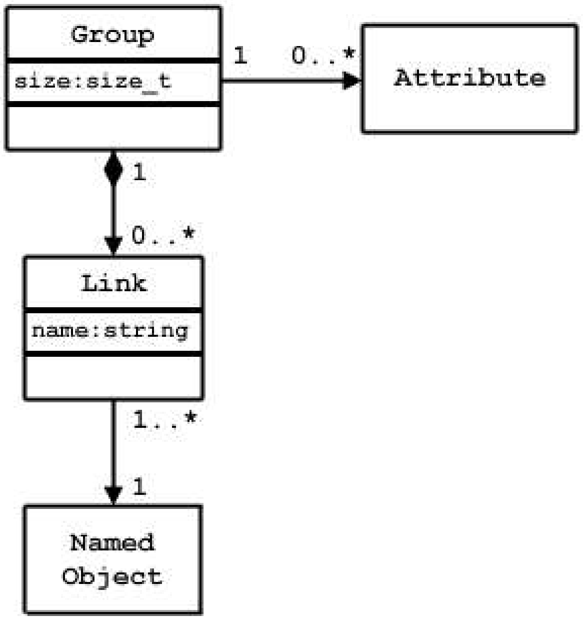


Figure 5

There are three classes of Named Objects: *Group*, *Dataset*, and *Named Datatype* (Figure 6). Each of these objects is the member of at least one Group, which means there is at least one *Link* to it.

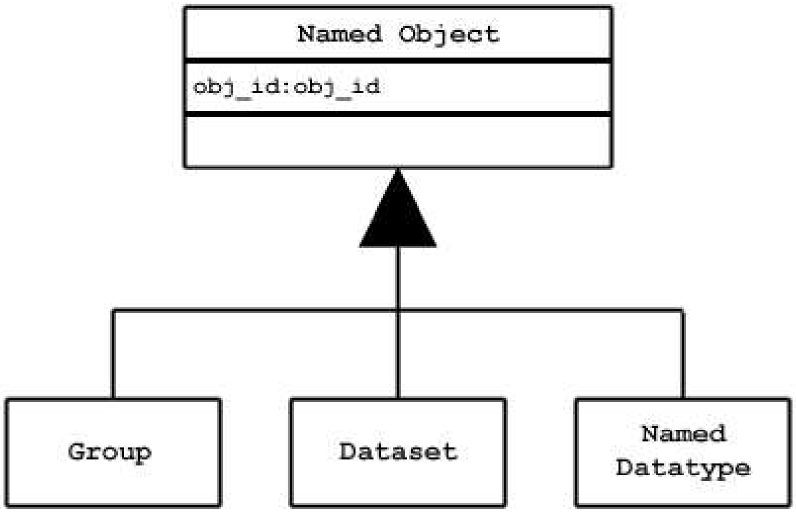


Figure 6

An HDF5 *Dataset* is a multidimensional (rectangular) array of *Data Elements* (Figure 7). The shape of the array (number of dimensions, size of each dimension) is described by the *Dataspace* object (see below).

A Data Element is a single unit of data which may be a number, a character, an array of numbers or characters, or a record of heterogeneous data elements. A Data Element is a set of bits, the layout of the bits is described by the *Datatype* (see below).

The *Dataspace* and *Datatype* are set when the *Dataset* is created, they can not be changed for the life of the Dataset. The *Dataset Creation Properties* are set when the Dataset is created. The Dataset Creation Properties include the fill value and storage properties such as chunking and compression. These properties cannot be changed after the Dataset is created.

The Dataset object manages the storage and access to the Data. While the Data is conceptually a contiguous rectangular array, it is physically stored and transferred in different ways depending on the storage properties and the storage mechanism used. The actual storage may be a set of chunks, which may be compressed, and the access may be through different storage mechanisms and caches. The Dataset maps between the conceptual array of elements and the actual stored data.

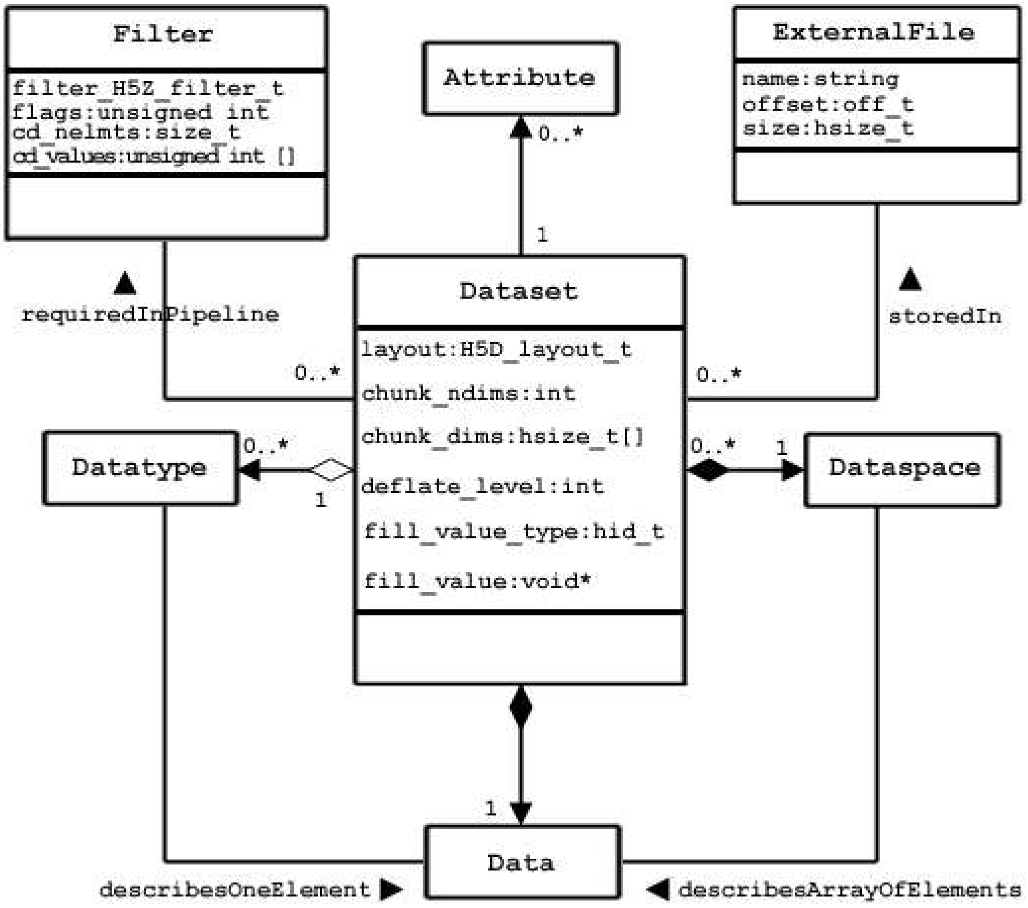
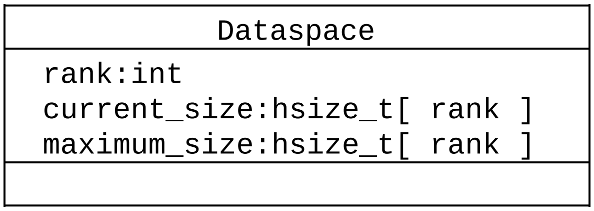


Figure 7

Dataspace

The HDF5 *Dataspace* describes the layout of the elements of a multidimensional array. Conceptually, the array is a hyper-rectangle with one to 32 dimensions. HDF5 Dataspaces can be extendable. Therefore, each dimension has a current and maximum size, and the maximum may be unlimited. The *Dataspace* describes this hyper-rectangle: it is a list of dimensions, with the current and maximum (or unlimited) size (Figure 8).



```
Dataspace
rank:int
current_size:hsize_t[ rank ]
maximum_size:hsize_t[ rank ]
```

Figure 8

Dataspace objects are also used to describe hyperslab selections from a dataset. Any subset of the elements of a Dastaset can be selected for *read* or *write* by specifying a set of hyperslabs. A non-rectangular region can be selected by the union of several (rectangular) Dataspaces.

Datatype

The HDF5 *Datatype* object describes the layout of a single data element. A data element is a single element of the array; it may be a single number, a character, an array of numbers or carriers, or other data. The Datatype object describes the storage layout of this data.

Data types are categorized into 11 classes of Datatype. Each class is interpreted according to a set of rules and has a specific set of properties to describe its storage. For instance, floating point numbers have exponent position and sizes, which are interpreted according to appropriate standards for number representation. Thus, the Dataype Class tells what the element means, and the Datatype describes how it is stored.

Figure 9 shows the classification of data types. Atomic Datatypes are indivisible, each may be a single object; a number, a string, or some other objects. The Composite Datatypes are composed of multiple elements of atomic Datatypes. In addition to the standard types, users can define additional Datatypes, such as a 24-bit integer, or a 16-bit float.

A Dataset or Attribute has a single Datatype object associated with it (Figure 7). The Datatype object may be used in the definition of several objects, but by default, a copy of the Datatype object will be private to the Dataset.

Optionally, a Datatype object can be stored in the HDF5 file. The Datatype is linked into a Group, and therefore given a name. A *Named Datatype* can be opened and used in any way that a Datatype object can be used.

The details of Datatypes, their properties, and how they are used are explained in the datatypes chapter, "HDF5 Datatypes."

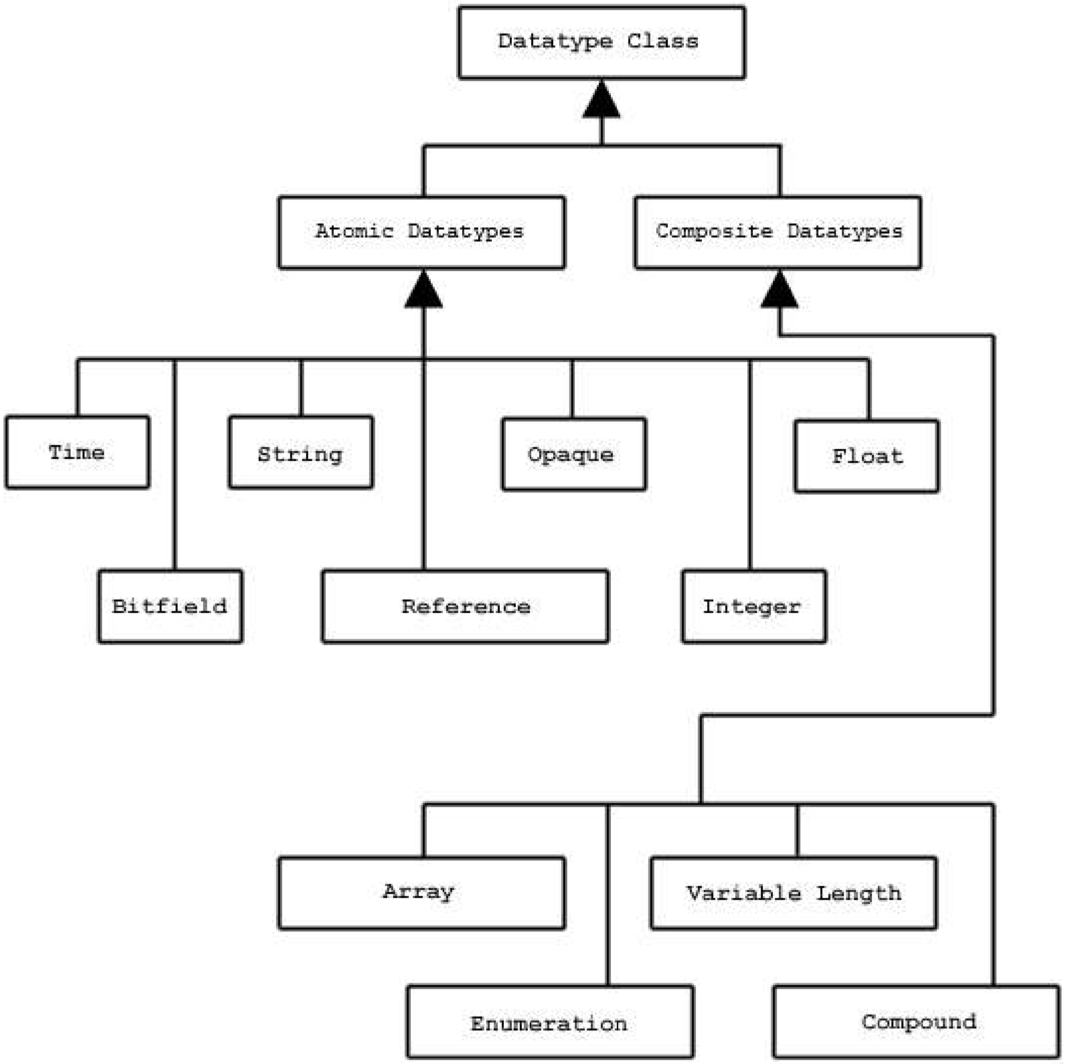


Figure 9

Attribute

Any HDF5 Named Data Object (Group, Dataset, or Named Datatype) may have zero or more user defined *Attributes*. Attributes are used to document the object. The Attributes of an object are stored with the object. An HDF5 Attribute has a *name* and data. The data is described analogously to the Dataset: the Dataspace defines the layout of an array of Data Elements, and the Datatype defines the storage layout and interpretation of the elements (Figure 10).

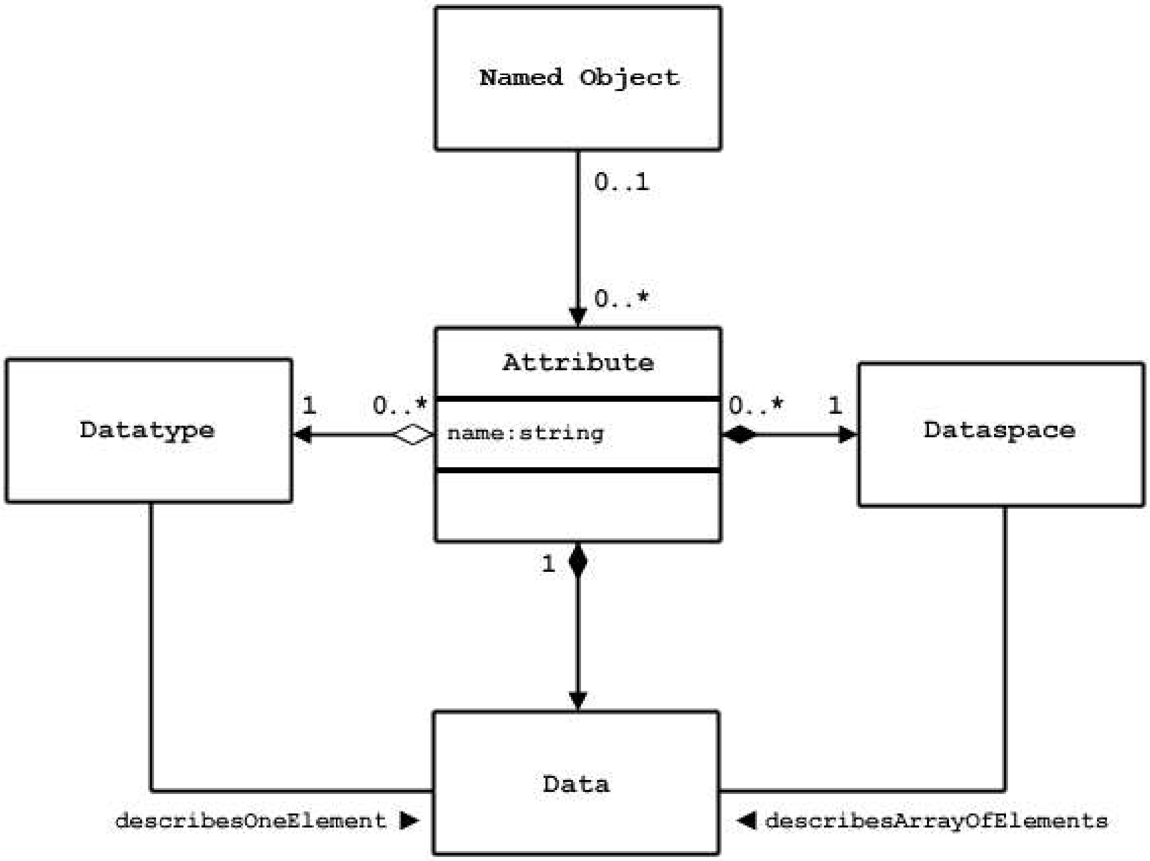


Figure 10

In fact, a Attribute is very similar to a Dataset with the following limitations:

- An attribute can only be accessed via the object, attribute names are significant only within the object. Attributes cannot be shared.
- For practical reasons, an Attribute should be a small object, no more than 1000 bytes.
- The data of an Attribute must be read or written in a single access, selection is not allowed.
- Attributes do not have Attributes.

Note that the value of an Attribute can be an *Object Reference*. A shared Attribute, or an Attribute that is a large array can be implemented as a Reference to a Dataset.

The name, Dataspace and Datatype of the Attribute are specified when it is created, and can not be changed over the life of the Attribute. The Attribute can be opened by name, by index, or by iterating through all the attributes of the object.

Property List

HDF5 has a generic Property List object, which is a collection of (*name*, *value*) pairs. Each class of Property List has a specific set of Properties. Each Property has an implicit name, an HDF5 Datatype, and a value (Figure 11). A Property List object is created and used similar to the other objects of the HDF5 library.

Property Lists are attached to the object in the library, they can be used by any part of the library. Some properties are permanent (e.g., the chunking strategy for a dataset), others are transient (e.g., buffer sizes for data transfer). A common use of a Property List is to pass parameters from the calling program to a VFL driver or a module of the pipeline.

Property Lists are conceptually similar to Attributes. Property Lists are information relevant to the behavior of the library, while Attributes are relevant to the user's data and application.

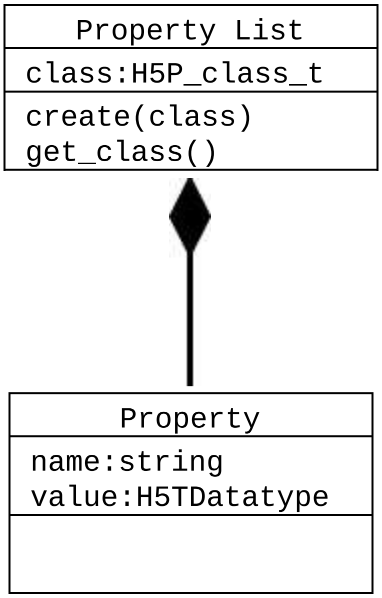


Figure 11

Properties are used to control optional behavior for file creation, file access, dataset creation, dataset transfer (read, write), and file mounting (Table 1). Details of the different Property Lists are explained in the relevant sections of this document.

Table 1.

Property List Class	Used	Examples
H5P_FILE_CREATE	Properties for file creation.	Set size of user block.
H5P_FILE_ACCESS	Properties for file access.	Set parameters for VFL driver, e.g., MPI I/O
H5P_DATASET_CREATE	Properties for dataset creation.	Set chunking, compression, fill value.
H5P_DATASET_XFER	Properties for raw data transfer (i.e., read and write).	Tune buffer sizes, memory management.
H5P_MOUNT	Properties for file mounting.	

3. The HDF5 Storage Model

3.1. The Abstract Storage Model: the HDF5 Format Specification

The *HDF5 Format Specification* defines how the HDF5 objects and data are mapped to a *linear address space*. The address space is assumed to be a contiguous array of bytes, stored on some random access medium.[1] The HDF5 Format defines the standard for how the objects of the HDF5 Abstract Data Model are mapped to the linear addresses. The stored representation is self-describing in the sense that the Format defines all the information necessary to read and reconstruct the original objects of the ADM.

The HDF5 Format Specification is organized in three parts:

1. **Level 0**: File Signature and Super Block
2. **Level 1**: File Infrastructure
    a. **Level 1A**: B-link Trees and B-tree nodes.
    b. **Level 1B**: Group
    c. **Level 1C**: Group Entry
    d. **Level 1D**: Local Heaps
    e. **Level 1E**: Global Heap
    f. **Level 1F**: Free-space index
3. **Level 2**: Data Object
    a. **Level 2A**: Data Object Headers
    b. **Level 2B**: Shared Data Object Headers
    c. **Level 2C**: Data Object Data Storage

The **Level 0** specification defines the header block for the file, which has a signature, version information, key parameters of the file layout (such as which VFL file drivers are needed) and pointers to the rest of the file. **Level 1** defines the data structures used throughout the file: the B-trees, heaps, and groups. **Level 2** defines the data structure for storing the data objects and data. In all cases, the data structures are completely specified so that every bit in the file can be faithfully interpreted.

It is important to realize that the structures defined in the HDF5 File Format are not the same as the Abstract Data Model: the object headers, heaps, and B-trees of the HDF5 File Specification are not represented in the Abstract Data Model. The HDF5 Format defines a number of objects for managing the storage, including header blocks, B-trees, and heaps. The *HDF5 Format Specification* defines how the Abstract objects (Groups, Datasets, etc.) are represented as headers, B-tree blocks, etc..

The HDF5 Library implements operations to write HDF5 objects to the linear format and to read from the linear format to create HDF5 objects. It is important to realize that a single HDF5 object, such as a *Dataset*, is usually stored as several objects (a header, one or more blocks for data, etc.), which may well not be contiguous on disk.

3.2. Concrete Storage Model

The HDF5 Format defines an abstract linear address space. This can be implemented in different storage media, such as a single file, multiple files, or memory. The HDF5 Library defines an open interface, called the *Virtual File Layer* (VFL), that allows different concrete storage models to be selected.

The Virtual File Layer defines an abstract model and API for random access storage, and an API to plug in alternative VFL driver modules. The model defines the operations that the VFL driver must and may support, and the plug-in API enables the HDF5 Library to recognize the driver and pass it control and data.

The HDF5 Library defines six VFL drivers: serial unbuffered, serial buffered, memory, MPI/IO, family of files, and split files (Figure 12, Table 2). Other drivers may also be available, such as a socket stream driver or Globus driver, and new drivers can be added.

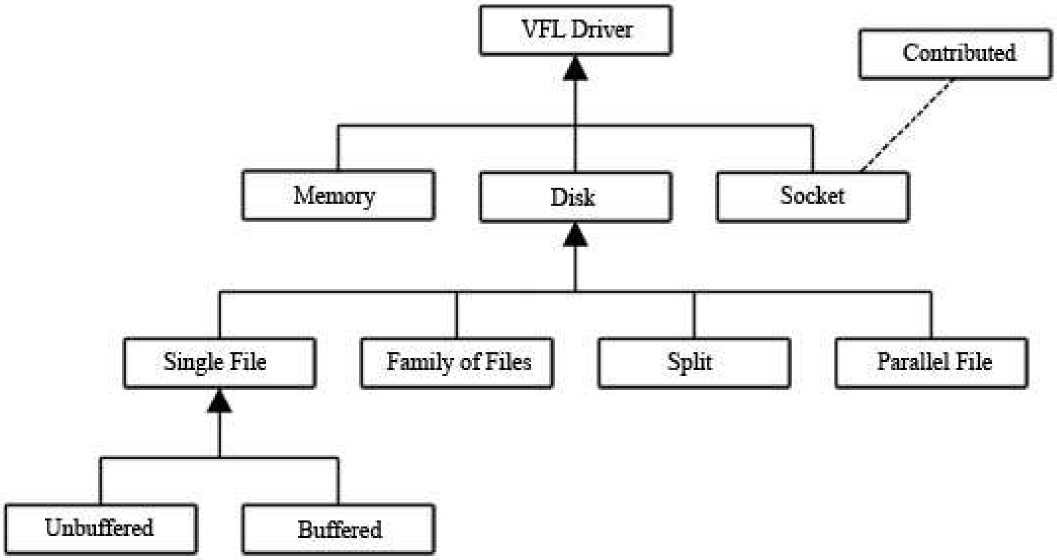


Figure 12. Conceptual hierarchy of VFL drivers.

Each driver isolates the details of reading and writing storage, so the rest of the HDF5 Library and user program can be almost the same for different storage methods. The exception to this rule is that some VFL drivers need information from the calling application, passed using property lists. For example, the MPI/IO driver requires certain control information that must be provided by the application.

Table 2

Driver	Description
Unbuffered Posix I/O (H5FD_SEC2) *Default*	Uses Posix file-system functions like read and write to perform I/O to a single file.
Buffered single file (H5FD_STDIO)	This driver uses functions from the Unix/Posix `stdio.h' to perform buffered I/O to a single file.
Memory (H5FD_CORE)	This driver performs I/O directly to memory. The I/O is memory to memory operations, but the 'file' is not persistent.
MPI/IO (H5FD_MPIIO)	This driver implements parallel file IO using MPI and MPI-IO
Family of files (H5FD_FAMILY)	The address space is partitioned into pieces and sent to separate storage locations using an underlying driver of the user's choice.
Split File (H5FD_SPLIT)	The format address space is split into meta data and raw data and each is mapped onto separate storage using underlying drivers of the user's choice.
Stream *Contributed*	This driver reads and writes the bytes to a Unix style socket, which can be a network channel. This is an example of a user defined VFL driver.

4. The Library and Programming Model

4.1. The Library Model

Table 3

Prefix	Object
H5A	Attribute object
H5D	Dataset object
H5E	Error report object
H5F	File object
H5G	Group object
H5I	Identifier object
H5P	Property List Object
H5R	Reference Object
H5S	Dataspace Object
H5T	Datatype Object
H5Z	Compression Object

The HDF5 Library implements the HDF5 Data Model and Storage Model as described above. In order to be as portable as possible, the library is implemented in portable C, which is not an object-oriented language. The HDF5 Library uses several mechanisms and conventions to implement an object model using C.

First, the HDF5 library implements the objects as data structures. In order to refer to an object, HDF5 library implements its own pointers, which are called *handles* or *identifiers*. A handle is used to invoke operations on a specific instance of an object. For example, when a Group is opened the API returns an *hid_t*. This object is an HDF Identifier Type, which is a reference to the specific Group. The *hid_t* is used to invoke operations on that Group. The *hid_t* is valid only within the context it is created, and remains valid until it is closed or the file is closed.

This mechanism is essentially the same as C++ or other object-oriented languages use to refer to objects, except the syntax is C.

Similarly, object-oriented languages collect all the methods for an object in a single name space, e.g., the methods of a C++ Class. The C language does not have any such mechanism, but this is a simulated the naming scheme in the HDF 5 library API. The operations on a particular class of objects are given names that began with the same prefix. Table 3 shows the HDF5 objects and the standard prefix used by the C API. For example, all the operations on a data type object are subroutines and to the names of all began with 'H5D'.

4.2 The Programming Model

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.

Figure 13 shows a code fragment to illustrate these steps. 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);
```

Figure 13

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

The Datatype 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 code in Figure 14 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);
```

Figure 14

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 code in Figure 15 closes the datatype, dataspace, and datasets that were created in the preceding section.

```
H5Tclose(datatype);
H5Dclose(dataset);
H5Sclose(dataspace);
```

Figure 15

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. F16 shows an example of how to write to a dataset.

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

Figure 16

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.

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 code segment in Figure 17 illustrates how to retrieve 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]);
```

Figure 17

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 in Figure 18.

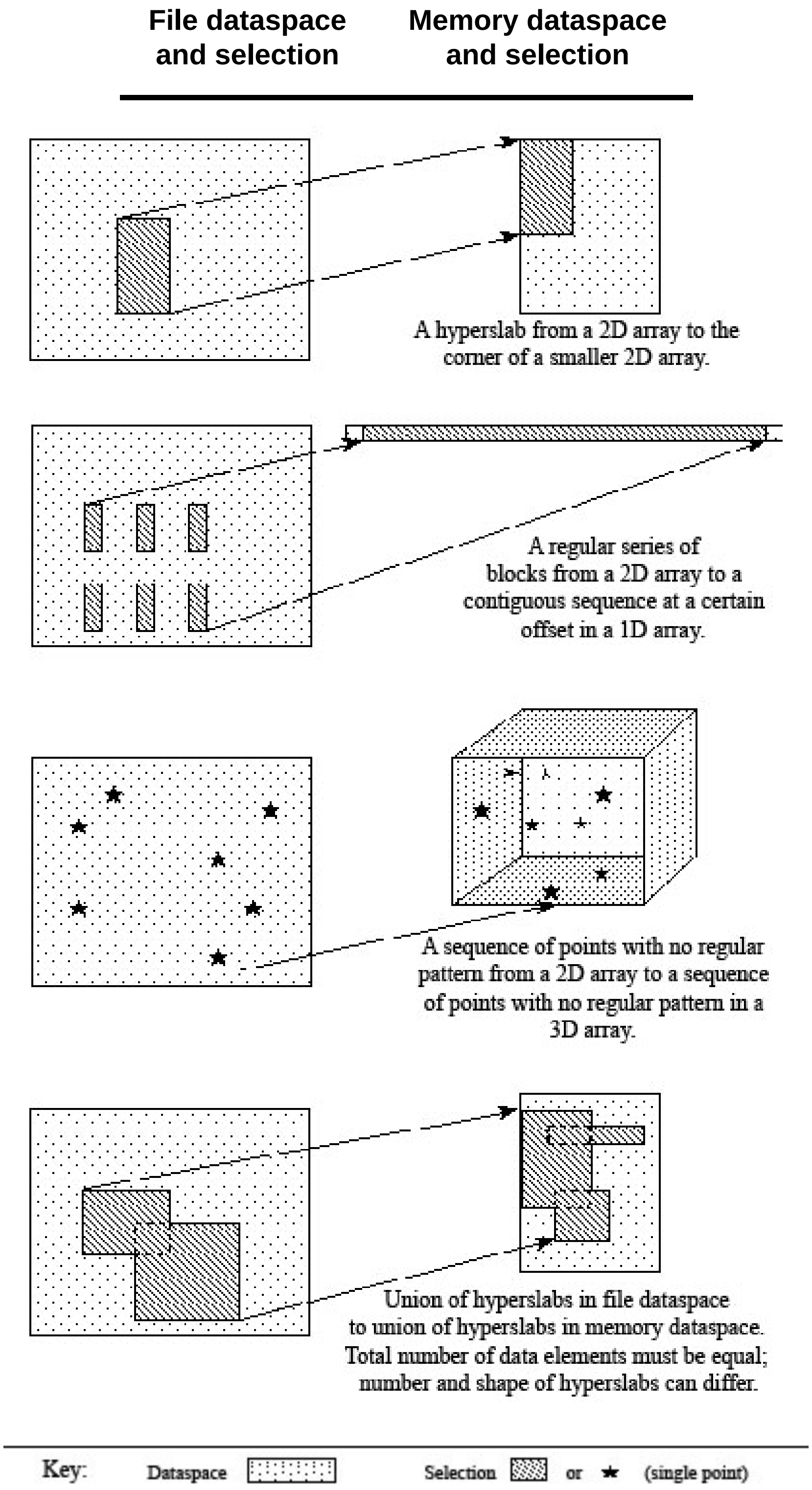

File dataspace and selection
Memory dataspace and selection
A hyperslab from a 2D array to the corner of a smaller 2D array.
A regular series of blocks from a 2D array to a contiguous sequence at a certain offset in a 1D array.
A sequence of points with no regular pattern from a 2D array to a sequence of points with no regular pattern in a 3D array.
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.
Key:
Dataspace
Selection
or
(single point)

Figure 18: Mappings between file dataspaces and selections and memory dataspaces and selections

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. 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. The parameters are shown in Table 4.

Table 4

Parameter	Definition
start	A starting location in the array. for the hyperslab.
stride	The number of elements to separate each element or block to be selected. If the *stride* parameter is set to NULL, the stride size defaults to 1 in each dimension (i.e., no elements are skipped).
count	The number of elements or blocks to select along each dimension.
block	The size of the 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 1s.

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 code in Figure 19 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);
```

Figure 19

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. Figure 20 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);
```

Figure 20

The hyperslab in Figure 20 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, as in Figure 21. The code in F22 will write 48 elements from vector to our file dataset, starting with the second element in vector.

-1	1	2	3	...	49	50	-1

Figure 21

```
/* 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)
```

Figure 22

Creating 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 a compound datatype you need to create a Datatype with class "compound", and define the total size of the data element, in bytes. A Compound Datatype consists of zero[2] or more members (defined in any order) with unique names and which occupy non-overlapping regions within the datum. Table 5 lists the properties of a member of a Compound Datatype.

Table 5

Parameter	Definition
Index number	An index number between zero and N-1, where N is the number of members in the compound. The elements are in the order of their location in the array of bytes.
name	A String that must be unique within the members of the same datatype.
datatype	An HDF5 Datatype.
offset	A fixed byte offset, which is the first byte (smallest byte address) of that member in a compound datatype.

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.

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).

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.

Figure 23 shows 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);
```

Figure 23

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, Figure 24 shows a 3x3 HDF5 dataset (a), which is then later extended into a 10x3 dataset by adding 7 rows (b), and then further extended to a 10x5 dataset by adding two columns (c).

1	1	1
1	1	1
1	1	1

a) Initially, 3 x 3

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

b) Extend to 10 x 3

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

c) Extend to 10 x 5

Figure 24

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 in Figure 24. 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);
```

Figure 25

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);
```

Figure 26

Then create a dataset.

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

Figure 27

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);
```

Figure 28

Working with groups in a file

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);
```

Figure 29

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);
```

Figure 30

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);
```

Figure 31

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);
```

Figure 32

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");
```

Figure 33

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");
```

Figure 34

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);
```

Figure 35

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);
```

Figure 36

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.

4.3. The Data Transfer Pipeline

The HDF5 Library implements data transfers between different storage locations. At the lowest levels, the HDF5 Library reads and writes blocks of bytes to and from storage using calls to the VFL drivers. In addition to this, the HDF5 Library manages caches of metadata, and a data I/O pipeline that applies compression to data blocks, transforms data elements, and implements selections.

For a given I/O requests, different combinations of actions may be performed by the pipeline. The HDF5 library automatically sets up the pipeline and passes the data through the processing steps. For example, for a *read* request (from disk to memory), the library must determine which logical blocks contain the requested data elements and fetch each block into the library's cache. If the data needs to be decompressed, then the compression algorithm is applied to the block after it is read from disk. If the data is a selection, the selected elements are extracted from the data block after it is decompressed. If the data needs to be transformed (e.g., byte swapped), then the data elements are transformed after decompression and select. And so on.

The data pipeline is automatically constructed to optimally fulfill each I/O request. This is normally transparent to the user program, the library determines what must be done based on the metadata for the file, object, and specific request. Figure 37 shows a simplified view of a data transfer with four stages. Note that the modules are used only when needed, e.g., if the data is not compressed, the compression stage is omitted.

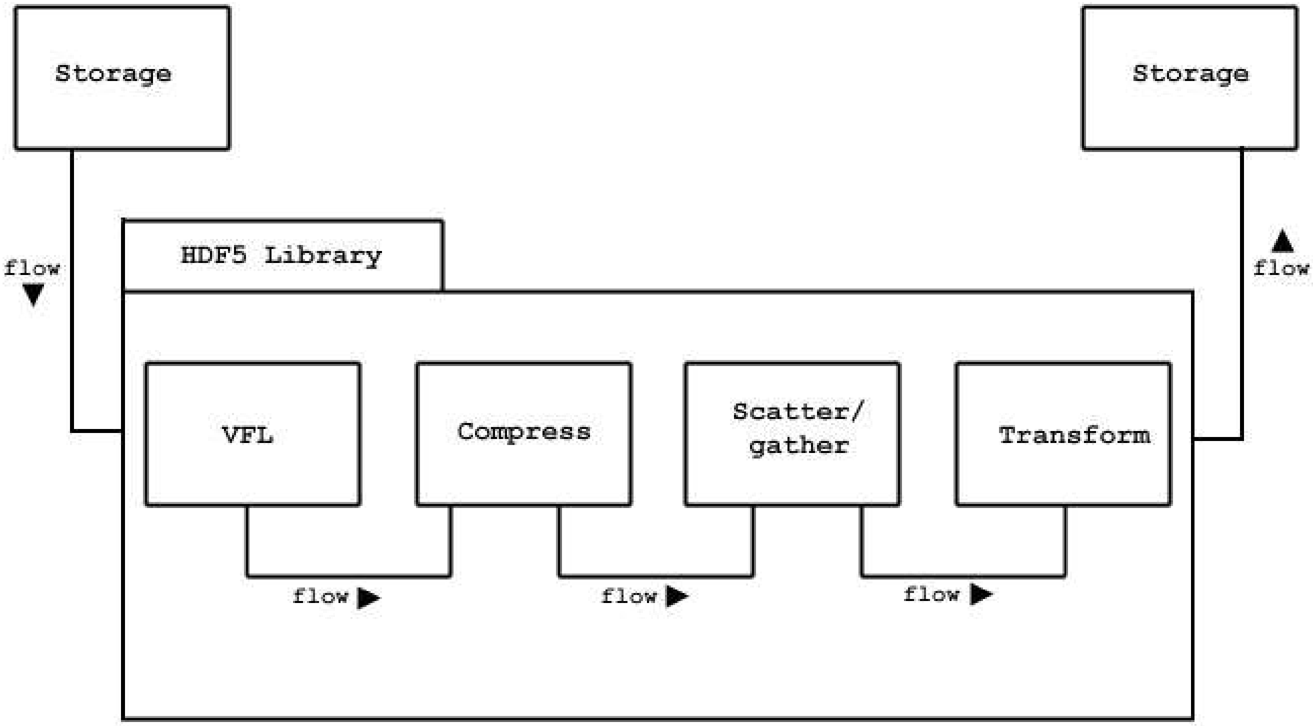


Figure 37

In some cases it is necessary or desirable to be able to pass parameters to and from modules in the pipeline, as well as us other parts of the library that are not directly called through the programming API. This is done by using HDF5 *Property List* objects. A Property List is basically a list of names and values that are passed to the library and may be read by any part of the library that is interested in those properties.

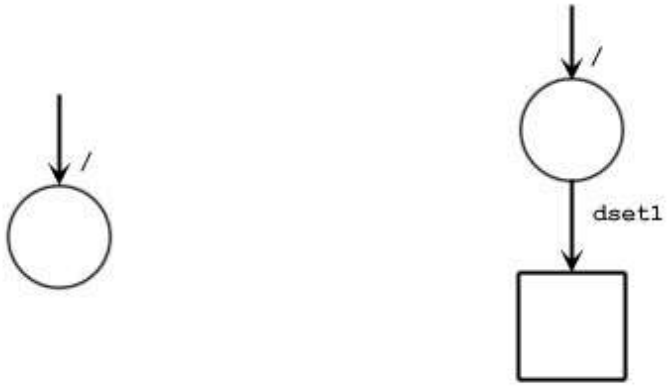
Property lists are used by many parts of the library, for instance to set the read/write properties of a file, or to set up or query parameters of MPI/IO in parallel HDF5.

Some of the stages of the pipe-line have interfaces that enable users or applications to install custom modules. For example, a new compression algorithm can be used with the HDF 5 library by linking an appropriate module into the pipeline. This requires creating a wrapper for the compression module that conforms to the H5Z API. Filters are registered with the library with `H5Zregister`, and applied to a dataset with the `H5Pset_filter` call.

5. The Structure of an HDF5 File

5.1. Overall File Structure

An HDF5 file is organized as a rooted, directed graph. The Named Data Objects are the nodes of the graph, and the links are the directed arcs. Each arc of the graph has a name, the root group has the name "/". Objects are created and then inserted into the graph with the link operation, which creates a named link from a Group to the object. For example, Figure 38 illustrates the structure of an HDF5 file when one dataset is created. An object can be the target of more than one link.



a) Newly created file: one group, `/`

b) Create a dataset called `/dset1` (`HDcreate(..., "/dset2", ...`)

Figure 38

The names on the links must be unique within each Group, but there may be many links with the same name in different groups. These are unambiguous, because some ancestor must have a different name, or else they are the same object. The graph is navigated with path names, analogous to Unix file systems. An object can be opened with a full path starting at the root group, or with a relative path and a starting node (Group). Note that all paths are relative to a single HDF5 File. In this sense, an HDF5 File is analogous to a single Unix File System. [3]

It is important to note that, just like the Unix file system, the Objects do not have *names*, the names are associated with *paths*. An object has a unique (within the file) *object id*, but a single object can have many *names* because there are many paths to the same object. An object can be renamed (moved to another Group) by adding and deleting links. In this case, the object itself never moves. For that matter, membership in a Group has no implication for the physical location of the stored object.

Deleting a link to an object does not necessarily delete the object. The object remains available as long as there is at least one link to it. After all links to an object are deleted, it can no longer be opened, although the storage may or may not be reclaimed. [4]

It is important to realize that the linking mechanism can be used to construct very complex graphs of objects. For example, it is possible for object to be shared between several groups and even to have more than one name in the same group. It is also possible for a group to be a member of itself, or create other "cycles" in the graph, such as a case where a child is the parent of one of its own ancestors.

HDF5 also has *Soft Links* similar to Unix soft links. A *Soft Link* is an object that contains a name and a path name for the target object. The Soft Link can be followed to open the target of the link, just like a regular (hard) link. Unlike hard Links, the target of a Soft Link has no count of the Soft Link to it. The reference count of an object is the number of hard Links (which must be >= 1). A second difference is that the hard link cannot be created if the target object does not exist, and always points to the same object. A Soft Link can be created with any path name, whether or not the object exists. Therefore, it may or may not be possible to follow a Soft Link, or the target object may change from one access to another access of the same Soft Link.

5.2. HDF5 Path Names and Navigation

The structure of the file constitutes the name space for the objects in the file. A path name is a string of components separated by '/'. Each component is the name of a (hard or soft) link, or the special characters "." (meaning current group). Link names (components) can be any string of ASCII characters not containing '/' (except the string ".", which is reserved). However, users are advised to avoid the use of punctuation and non-printing characters, because they may create problems for other software. Figure 39 gives a BNF grammar for HDF5 *path names*.

```
PathName ::= AbsolutePathName | RelativePathName
Separator ::= "/" ["/"]*
AbsolutePathName ::= Separator [ RelativePathName ]
RelativePathName ::= Component [ Separator RelativePathName ]*
Component ::=  "." |  Name
Name ::= Character+  -  {"."}
Character ::= {c:  c in {{ legal ASCII characters }  -  {'/'}}
```

Figure 39

An object can always be addressed by a *full or absolute path*, i.e., starting at the root group. As already noted, a given object can have more than one full path name. An object can also be addressed by a relative path, i.e., a group plus a path starting at the group.

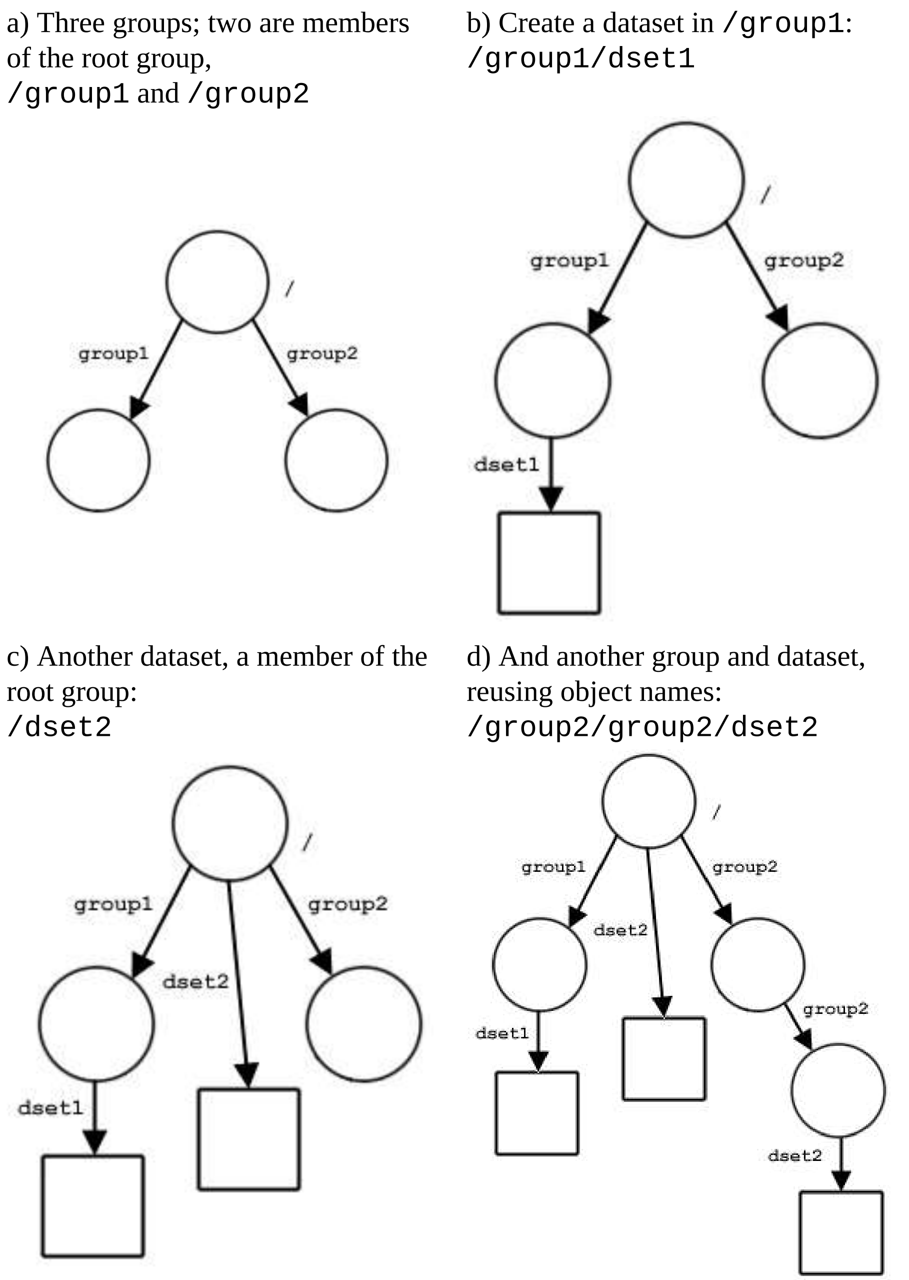
The structure of an HDF5 file is "self-describing", in that it is possible to *navigate* the file to discover all the objects in the file. Basically, the structure is traversed as a graph, starting at one node, and recursively visiting the nodes of the graph.

The members of a Group can be discovered with the H5Giterate function, and a description of the object can be retrieved with the H5Gget_obj_info function. In this way, all the members of a given group can be determined, and each can be opened to retrieve a description, or the data and attributes of the object.

5.3. Examples of HDF5 File Structures

Figure 40 shows some examples of the structure of a file with three groups and one dataset. Figure 40a shows the structure of a file with three groups, the root with two members. Figure 40b shows a dataset created in "/group1". Figure 40c shows the structure after the dataset is linked (with H5Glink) to "/group2" with the name "dset2". Note that there is only one copy of the dataset, it has two different links to it and can be accessed by two different paths: "/group1/dset1" and "/group2/dset2".

Figure 40d shows that one of the two links to the dataset can be deleted (with H5Gunlink()). In this case, the link from "/group1" is removed. The dataset is not deleted, it is still in the file but can only be accessed as "/group1/dset2".



a) Three groups; two are members of the root group, `/group1` and `/group2`

b) Create a dataset in `/group1`: `/group1/dset1`

c) Another dataset, a member of the root group: `/dset2`

d) And another group and dataset, reusing object names: `/group2/group2/dset2`

Figure 40: Examples of HDF5 file structures with groups and datasets

[1] HDF5 requires random access to the linear address space. For this reason it is not well suited for some data media, such as streams.

[2] However, a Compound Datatype with zero members can have no data, so it is useless.

[3] It could be said that HDF5 extends the organizing concepts of a file system to the internal structure of a single file.

[4] As of HDF5-1.4, the storage used for an object is reclaimed, even if all links are deleted.

Chapter 2
The HDF5 File

1. Introduction

If HDF5 data is to be written to or read from a file, that file must first be explicitly created or opened with the appropriate file driver and access privileges. Once all work with data is complete, the file must be explicitly closed.

This chapter discusses the following:

- File access modes
- Creating, opening, and closing files
- The use of file creation property lists
- The use of file access property lists, including low-level file drivers

The remaining sections of this chapter require a brief summary of the HDF5 mechanisms for handling file access modes, file access properties and file creation properties, and the use of low-level file drivers. These topics are discussed briefly in the following paragraphs. This chapter assumes an understanding of the material presented in the data model chapter, "HDF5 Data Model and File Structure."

File access modes

There are two issues regarding file access:

- What should happen when a new file is created but a file of the same name already exists? Should the create action fail or should the existing file be overwritten?
- Is a file to be opened with read-only or read-write access?

Four access modes address these concerns, with `H5Fcreate` and `H5Fopen` each accepting two of them:

- `H5Fcreate` accepts `H5F_ACC_TRUNC` or `H5F_ACC_EXCL`.
- `H5Fopen` accepts `H5F_ACC_RDONLY` or `H5F_ACC_RDWR`.

Access flag	Resulting access mode
`H5F_ACC_EXCL`	If file already exists, `H5Fcreate` fails. If file does not exist, it is created and opened with read-write access.
`H5F_ACC_TRUNC`	If file already exists, file is opened with read-write access and new data overwrites existing data, destroying all prior content, i.e., file content is truncated upon opening. If file does not exist, it is created and opened with read-write access.
`H5F_ACC_RDONLY`	Existing file is opened with read-only access. If file does not exist, `H5Fopen` fails.
`H5F_ACC_RDWR`	Existing file is opened with read-write access. If file does not exist, `H5Fopen` fails.

The HDF5 library assumes that existing files are implicitly opened for read access; passing the `H5F_ACC_RDWR` parameter to `H5Fopen` allows read-write access to the file.

`H5Fcreate` assumes read-write access. Passing `H5F_ACC_TRUNC` forces the truncation of an existing file; otherwise `H5Fcreate` does not permit overwriting an existing file.

File creation and file access properties

File creation and file access property lists control the more complex aspects of creating and accessing files.

File creation property lists control characteristics of a file, such as the size of the user-block, a user-definable data block; the size of data address parameters; properties of the B-trees are used to manage the data in the file; and certain HDF5 library versioning information.

See “File creation properties,” below, for a more detailed discussion of file creation properties and appropriate references to the *HDF5 Reference Manual*. If you have no special requirements for these file characteristics, you can simply specify `H5P_DEFAULT`, for the default file creation property list, when a file creation property list is called for.

File access property lists control properties and means of accessing a file, such as data alignment characteristics, meta data block and cache sizes, data sieve buffer size, garbage collection settings, and parallel I/O. Data alignment, meta data block and cache sizes, and data sieve buffer size are factors in improving I/O performance.

See “File access properties,” below, for a more detailed discussion of file access properties and appropriate references to the *HDF5 Reference Manual*. If you have no special requirements for these file access characteristics, you can simply specify `H5P_DEFAULT`, for the default file access property list, when a file access property list is called for.

Low-level file drivers

The concept of an HDF5 file is actually rather abstract: the address space for what is normally thought of as an HDF5 file might correspond to any of the following at the storage level:

- Single file on a standard file system
- Multiple files on a standard file system
- Multiple files on a parallel file system
- Block of memory within an application’s memory space
- More abstract situations, such as virtual files or streaming I/O

This HDF5 address space is generally referred to as an *HDF5 file* regardless of its organization at the storage level.

HDF5 accesses a file, i.e., the address space, through various types of *low-level file drivers*. The default HDF5 file storage layout is as an unbuffered permanent file, which is a single, contiguous file on local disk. Alternative layouts are designed to suit the needs of a variety of systems, environments, and applications.

2. Programming Model

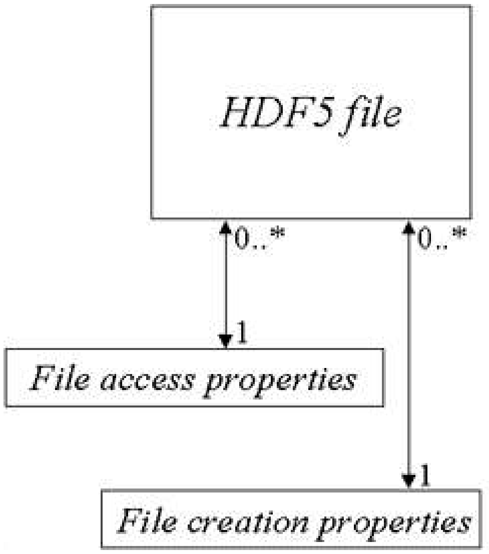


Figure 1: UML model for an HDF5 file and its file creation and file access property lists

2.1 Creating a new file

The programming model for creating a new HDF5 file can be summarized as follows:

- Define file creation property list (optional).
- Define file access property list, including low-level file driver (optional).
- Create file.

First consider the simple case where we wish to rely on the HDF5 defaults. All we have to do is create the file:

```
file_id = H5Fcreate ("SampleFile.h5",
    H5F_ACC_EXCL, H5P_DEFAULT,
    H5P_DEFAULT)
```

Note that this example specifies that `H5Fcreate` should fail if `SampleFile.h5` already exists.

Now consider the more generalized case, in which we define file creation and access property lists (though we do not assign any properties), specify that `H5Fcreate` should fail if `SampleFile.h5` already exists, and create a new file named `SampleFile.h5`. The example does not specify a driver, so the default driver, SEC2 or `H5FD_SEC2`, will be used.

```
fcplist_id = H5Pcreate (H5P_FILE_CREATE)
   <...set desired file creation properties...>
faplist_id = H5Pcreate (H5P_FILE_ACCESS)
   <...set desired file access properties...>
file_id = H5Fcreate ("SampleFile.h5", H5F_ACC_EXCL, fcplist_id, faplist_id)
```

Notes: A root group is automatically created in a file when the file is first created. File property lists, once defined, can be reused when another file is created within the same application.

2.2 Opening an existing file

The programming model for opening an existing HDF5 file can be summarized as follows:

- Define or modify file access property list, including low-level file driver (optional).
- Open file.

Now consider an example in which we re-open `SampleFile.h5`. For the sake of the example, we will open it with a different driver, stdio or `H5FD_STDIO`, and declare read-only access.

```
faplist_id = H5Pcreate (H5P_FILE_ACCESS) status = H5Pset_fapl_stdio (faplist_id) file_id = H5Fopen ("SampleFile.h5", H5F_ACC_RDONLY, faplist_id)
```

2.3 Closing a file

The programming model for closing an HDF5 file is very simple:

- Close file.

We close `SampleFile.h5` with the following line of code.

```
status = H5Fclose (file_id)
```

Note that `H5Fclose` flushes all unwritten data to storage. `file_id` is the identifier returned for `SampleFile.h5` by `H5Fopen`.

More comprehensive discussions regarding all of these steps are provided below.

3. Using h5dump

The HDF5 distribution includes a command-line utility, `h5dump`, which provides a straight-forward means of inspecting the contents of an HDF5 file. `h5dump` thus enables a programmer to verify that a program is generating the intended HDF5 file. `h5dump` displays ASCII output formatted according to the HDF5 DDL grammar.

The following `h5dump` command will display the contents of `SampleFile.h5`:

```
h5dump SampleFile.h5
```

If no datasets or groups have been created in and no data has been written to the file, the output will look something like the following:

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

Note that the root group, indicated above by `/`, was automatically created when the file was created.

`h5dump` is fully described on the Tools page of the *HDF5 Reference Manual*. The HDF5 DDL grammar is fully described in the document DDL in BNF for HDF5, an element of this *HDF5 User's Guide*.

4. File Function Summaries

4.1 File functions

C Function F90 Function	Purpose
H5Fcreate h5fcreate_f	Creates new HDF5 file.
H5Fopen h5fopen_f	Opens existing HDF5 file.
H5Fclose h5fclose_f	Closes HDF5 file.
H5Fflush h5fflush_f	Flushes data to HDF5 file on storage medium.

4.2 File creation property list functions

C Function F90 Function	Purpose
H5Pset/get_userblock h5pset/get_userblock_f	Sets/retrieves size of user block.
H5Pset/get_sizes h5pset/get_sizes_f	Sets/retrieves byte size of offsets and lengths used to address objects in HDF5 file.
H5Pset/get_sym_k h5pset/get_sym_k_f	Sets/retrieves size of parameters used to control symbol table nodes.
H5Pset/get_istore_k h5pset/get_istore_k_f	Sets/retrieves size of parameter used to control B-trees for indexing chunked datasets.
H5Pget_version h5pget_version_f	Retrieves version information for various objects for file creation property list.

4.3 File access property list functions (except file drivers)

C Function F90 Function	Purpose
H5Pset/get_meta_block_size h5pset/get_meta_block_size_f	Sets the minimum meta data block size or retrieves the current meta data block size setting.
H5Pset/get_sieve_buf_size h5pset/get_sieve_buf_size_f	Sets/retrieves maximum size of data sieve buffer.
H5Pset/get_alignment h5pset/get_alignment_f	Sets/retrieves alignment properties.
H5Pset/get_cache h5pset/get_cache_f	Sets/retrieves meta data cache and raw data chunk cache parameters.
H5Pset/get_fclose_degree h5pset/get_fclose_degree_f	Sets/retrieves file close degree property.

`H5Pset/get_gc_references` `h5pset/get_gc_references_f`	Sets/retrieves garbage collecting references flag.

4.4 File driver functions

C Function F90 Function	Purpose
`H5Pget_driver` `h5pget_driver_f`	Determines driver used to create file.
`H5Pset_fapl_sec2` `h5pset_fapl_sec2_f`	Sets driver for unbuffered permanent files or retrieves information regarding driver.
`H5Pset_fapl_stdio` (none)	Sets driver for buffered permanent files.
`H5Pset/get_fapl_mpio` `h5pset/get_fapl_mpi_f`	Sets driver for files on parallel file systems (MPI I/O) or retrieves information regarding the driver.
`H5Pset/get_fapl_family` `h5pset/get_fapl_family_f`	Sets driver for file families, designed for systems that do not support files larger than 2 gigabytes, or retrieves information regarding driver.
`H5Pset/get_fapl_multi` `h5pset/get_fapl_multi_f`	Sets driver for multiple files, separating categories of meta data and raw data, or retrieves information regarding driver.
`H5Pset_fapl_split` `h5pset_fapl_split_f`	Sets driver for split files, a limited case of multiple files with one meta data file and one raw data file.
`H5Pset/get_fapl_core` `h5pset/get_fapl_core_f`	Sets driver for buffered memory files (i.e., in RAM) or retrieves information regarding driver.
`H5Pset/get_fapl_stream` (none)	Sets driver for streaming data (i.e., no stored file) or retrieves information regarding driver.
`H5Pset_fapl_log` (none)	Sets logging driver.

5. Creating or Opening an HDF5 File

5.1 Defining the file creation and file access property lists

This step is optional; you can always rely on the default property lists in creating a new file and the default or previously-defined file access property list with an existing file.

See “File Property Lists,” below, for details of setting property list values. See “File Access Modes,” in the introduction to this chapter above, for the complete list of file access flags and their descriptions.

5.2 Working with the file

New HDF5 files are created and opened with `H5Fcreate`; existing files are opened with `H5Fopen`. Both functions return an object identifier, which must eventually be released by calling `H5Fclose`.

To create a new file, call `H5Fcreate`:

```
hid_t H5Fcreate (const char *name, unsigned flags,
     hid_t fcpl_id, hid_t fapl_id)
```

`H5Fcreate` creates a new file named *name* in the current directory. The file is opened with read and write access; if the `H5F_ACC_TRUNC` flag is set, any pre-existing file of the same name in the same directory is truncated. If either `H5F_ACC_TRUNC` is not set or `H5F_ACC_EXCL` is set and if a file of the same name exists, `H5Fcreate` will fail.

The new file is created with the properties specified in the property lists *fcpl_id* and *fapl_id*. Specifying `H5P_DEFAULT` for either the creation or access property list calls for the library's default creation or access properties.

If `H5Fcreate` successfully creates the file, it returns a file identifier for the new file. This identifier will be used by the application any time an object identifier, an OID, for the file is required. Once the application has finished working with a file, the identifier should be released and the file closed with `H5Fclose`.

To open an existing file, call `H5Fopen`:

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

`H5Fopen` opens an existing file with read-write access if `H5F_ACC_RDWR` is set and read-only access if `H5F_ACC_RDONLY` is set.

fapl_id is the file access property list identifier. Alternatively, `H5P_DEFAULT` indicates that the application relies on the default I/O access parameters. Creating and changing access property lists is documented further below.

A file can be opened more than once via multiple `H5Fopen` calls. Each such call returns a unique file identifier and the file can be accessed through any of these file identifiers as long as they remain valid. Each of these file identifiers must be released by calling `H5Fclose` when it is no longer needed.

6. Closing an HDF5 File

`H5Fclose` both closes a file and releases the file identifier returned by `H5Fopen` or `H5Fcreate`. `H5Fclose` must be called when an application is done working with a file; while the HDF5 library makes every effort to maintain file integrity, failure to call `H5Fclose` may result in the file being abandoned in an incomplete or corrupted state.

To close a file, call `H5Fclose`:

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

This function releases resources associated with an open file. After closing a file, the file identifier, *`file_id`*, cannnot be used again as it will be undefined.

`H5Fclose` fulfills three purposes: to ensure that the file is left in an uncorrupted state, to ensure that all data has been written to the file, and to release resources. Use `H5Fflush` if you wish to ensure that all data has been written to the file but it is premature to close it.

Note regarding serial mode behavior: When `H5Fclose` is called in serial mode, it closes the file and terminates new access to it, but it does not terminate access to objects that remain individually open within the file. That is, if `H5Fclose` is called for a file but one or more objects within the file remain open, those objects will remain accessible until they are individually closed. To illustrate, assume that a file, `fileA`, contains a dataset, `data_setA`, and that both are open when `H5Fclose` is called for `fileA`. `data_setA` will remain open and accessible, including writable, until it is explicitly closed. The file will be automatically and finally closed once all objects within it have been closed.

Note regarding parallel mode behavior: Once `H5Fclose` has been called in parallel mode, access is no longer available to any object within the file.

7. File Property Lists

Additional information regarding file structure and access are passed to `H5Fcreate` and `H5Fopen` through property list objects. Property lists provide a portable and extensible method of modifying file properties via simple API functions. There are two kinds of file-related property lists:

- File creation property lists
- File access property lists

In the following subsections, we discuss only one file creation property, user-block size, in detail as a model for the user. Other file creation and file access properties are mentioned and defined briefly, but the model is not expanded for each; complete syntax, parameter, and usage information for every property list function is provided in the “H5P: Property List Interface” chapter of the *HDF5 Reference Manual*.

7.1 Creating a property list

If you do not wish to rely on the default file creation and access properties, you must first create a property list with `H5Pcreate`.

```
hid_t H5Pcreate (hid_t cls_id)
```

`type` is the type of property list being created. In this case, the appropriate values are `H5P_FILE_CREATE` for a file creation property list and `H5P_FILE_ACCESS` for a file access property list.

Thus, the following calls create first a file creation property list then a file access property list with identifiers *`fcpl_id`* and *`fapl_id`*, respectively:

```
fcpl_id = H5Pcreate (H5P_FILE_CREATE)
fapl_id = H5Pcreate (H5P_FILE_ACCESS)
```

Once the property lists have been created, the properties themselves can be modified via the functions described in the following subsections.

7.2 File creation properties

File creation property lists control the file meta data, which is maintained in the super block of the file. These properties are used only when a file is first created.

User-block size

```
herr_t H5Pset_userblock (hid_t plist, hsize_t size)
herr_t H5Pget_userblock (hid_t plist, hsize_t *size)
```

The *user-block* is a fixed-length block of data located at the beginning of the file and which is ignored by the HDF5 library. This block is specifically set aside for any data or information that developers determine to be useful to their application but that will not be used by the HDF5 library. The *`size`* of the user-block is defined in bytes and may be set to any power of two, with a minimum size of 512 bytes (i.e. 512, 1024, 2048, etc). This property is set with `H5Pset_userblock` and queried via `H5Pget_userblock`.

For example, if an application was thought to reqire a 4K user-block, that could be set with the following function call:

```
status = H5Pset_userblock(fcpl_id, 4096)
```

The property list could later be queried with

```
status = H5Pget_userblock(fcpl_id, size)
```

and the value `4096` would be returned in the parameter *`size`*.

Other properties, below, are set and queried in exactly the same manner. Syntax and usage are detailed in “H5P: Property List Interface” in the *HDF5 Reference Manual*.

Offset and length sizes

This property specifies the number of bytes used to store the offset and length of objects in the HDF5 file. Values of 2, 4, and 8 bytes are currently supported to accommodate 16-bit, 32-bit, and 64-bit file address spaces.

These properties are set and queried via `H5Pset_sizes` and `H5Pget_sizes`.

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 properties are set and queried via `H5Pset_sym_k` and `H5Pget_sym_k`.

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 properties are set and queried via `H5Pset_istore_k` and `H5Pget_istore_k`.

Version information

Various objects in an HDF5 file may over time appear in different versions. The HDF5 library keeps track of the version of each object in the file.

Version information is retrieved via `H5Pget_version`.

7.3 File access properties

This section discusses file access properties that are not related to the low-level file drivers. File drivers are discussed separately in “Alternate File Storage Layouts and Low-level File Drivers,” later in this chapter.

File access property lists control various aspects of file I/O and structure.

Data alignment

Sometimes file access is faster if certain data elements are aligned in a specific manner. This can be controlled by setting alignment properties via the `H5Pset_alignment` function. Two values are involved,

◊ a threshhold value and
◊ an alignment interval.

Any allocation request at least as large as the threshold will be aligned on an address that is a multiple of the alignment interval.

Meta data block allocation size

Meta data typically exists as very small chunks of data; storing meta data elements in a file without blocking them can result in hundreds or thousands of very small data elements in the file. This can result in a highly fragmented file and seriously impede I/O. By blocking meta data elements, these small elements can be grouped in larger sets, thus alleviating both problems.

`H5Pset_meta_block_size` sets the minimum size in bytes of meta data block allocations.
`H5Pget_meta_block_size` retrieves the current minimum meta data block allocation size.

Meta data cache

Meta data and raw data I/O speed are often governed by the size and frequency of disk reads and writes. In many cases, the speed can be substantially improved by the use of an appropriate cache.

`H5Pset_cache` sets the minimum cache size for both meta data and raw data and a preemption value for raw data chunks. `H5Pget_cache` retrieves the current values.

Data sieve buffer size

Data sieve buffering is used by certain file drivers to speed data I/O, most commonly when working with dataset hyperslabs. For example, using a buffer large enough to hold several pieces of a dataset as it is read in for hyperslab selections will boost performance noticeably.

`H5Pset_sieve_buf_size` sets the maximum size in bytes of the data sieve buffer.
`H5Pget_sieve_buf_size` retrieves the current maximum size of the data sieve buffer.

Garbage collection references

Dataset region references and other reference types use space in an HDF5 file’s global heap. If garbage collection is on (`1`) and the user passes in an uninitialized value in a reference structure, the heap might become corrupted. When garbage collection is off (`0`), 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.

`H5Pset_gc_references` sets the garbage collecting references flag.

8. Alternate File Storage Layouts and Low-level File Drivers

The concept of an HDF5 file is actually rather abstract: the address space for what is normally thought of as an HDF5 file might correspond to any of the following:

- Single file on standard file system
- Multiple files on standard file system
- Multiple files on parallel file system
- Block of memory within application's memory space
- More abstract situations, such as virtual files or streaming I/O

This HDF5 address space is generally referred to as an *HDF5 file* regardless of its organization at the storage level.

HDF5 employs an extremely flexible mechanism called the *virtual file layer*, or VFL, for file I/O. A full understanding of the VFL is only necessary if you plan to write your own drivers (see "Virtual File Layer" and "List of VFL Functions" in the *HDF5 Technical Notes*). For our purposes here, it is sufficient to know that the low-level drivers used for file I/O reside in the VFL, as illustrated in the following figure.

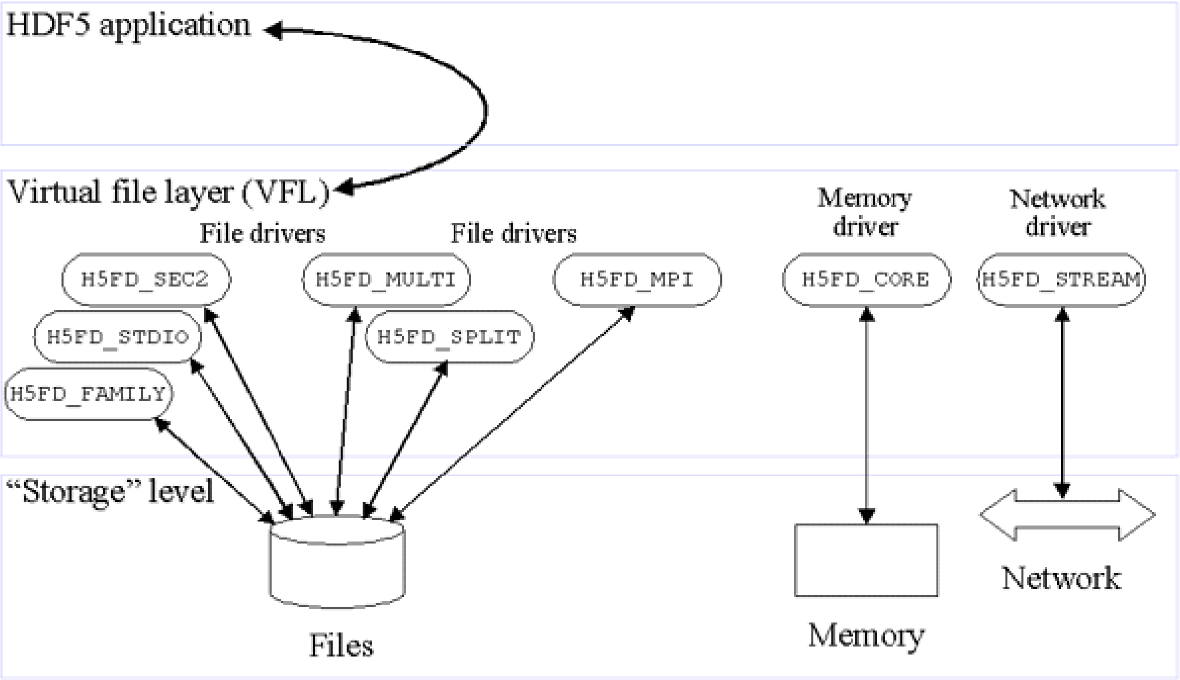


Figure 2: I/O path from application through VFL and low-level drivers to storage level

As mentioned above, HDF5 applications access HDF5 files through various *low-level file drivers*. The default HDF5 file storage layout is as an unbuffered permanent file, which is a single, contiguous file on local disk. The default driver for that layout is the SEC2 driver, `H5FD_SEC2`. Alternative layouts and drivers are designed to suit the needs of a variety of systems, environments, and applications.

The following table lists the supported drivers distributed with the HDF5 library and their associated file storage layouts.

Storage layout	Driver	Intended usage
Unbuffered permanent file	`H5FD_SEC2`	Permanent file on local disk with minimal buffering. Posix-compliant. Default.
Buffered permanent file	`H5FD_STDIO`	Permanent file on local disk with additional low-level buffering.
File family	`H5FD_FAMILY`	Several files that, together, constitute a single virtual HDF5 file. Designed for systems that do not support files larger than 2 gigabytes.
Multiple files	`H5FD_MULTI`	Separate files for different types of meta data and for raw data.
Split files	`H5FD_SPLIT`	Two files, one for meta data and one for raw data (limited case of `H5FD_MULTI`).
Parallel files (MPI I/O)	`H5FD_MPI`	Parallel files accessed via the MPI I/O layer. The standard HDF5 file driver for parallel file systems.
Buffered temporary file	`H5FD_CORE`	Temporary file maintained in memory, not written to disk.
Streaming I/O	`H5FD_STREAM`	Streaming I/O over network, no file maintained.
Access logs	`H5FD_LOG`	The SEC2 driver with logging capabilities.

Note that the low-level file drivers manage alternative file storage layouts. Alternative dataset storage layouts, such as chunking, compression, and external dataset storage, are orthogonal to file storage layout and are managed independently.

If an application requires a special-purpose low-level driver, the VFL provides a public API for creating one. But that activity is beyond the scope of this document (see "Virtual File Layer" and "List of VFL Functions" in the *HDF5 Technical Notes*).

8.1 Identifying the previously-used file driver

When creating a new HDF5 file, no history exists, so the file driver must be specified if it is to be other than the default.

When opening existing files, however, the application may need to determine which low-level driver was used to create the file. The function `H5Pget_driver` is used for this purpose.

```
hid_t H5Pget_driver (hid_t fapl_id)
```

`H5Pget_driver` returns a constant identifying the low-level driver for the access property list *fapl_id*. For example, if the file was created with the SEC2 driver, `H5Pget_driver` returns `H5FD_SEC2`.

fapl_id has presumably been previously identified as the access property list for the file being opened.

If the application opens an HDF5 file without both determining the driver used to create the file and setting up the use of that driver, the HDF5 library will examine the Super Block and the Driver Definition Block to identify the driver. See the *HDF5 File Format Specification* for detailed descriptions of the Super Block and the Driver Definition Block.

8.2 Unbuffered permanent files -- SEC2 driver

The SEC2 driver, `H5FD_SEC2`, uses functions from section 2 of the Posix manual to access unbuffered files stored on a local file system. The HDF5 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 library.

The function `H5Pset_fapl_sec2` sets the file access properties to use the SEC2 driver.

```
herr_t H5Pset_fapl_sec2 (hid_t fapl_id)
```

Any previously-defined driver properties are erased from the property list.

Additional parameters may be added to this function in the future. Since there are no additional variable settings associated with the SEC2 driver, there is no `H5Pget_fapl_sec2` function.

8.3 Buffered permanent files -- STDIO driver

The STDIO driver, `H5FD_STDIO` also accesses permanent files in a local file system, but with an additional layer of buffering beneath the HDF5 library.

The function `H5Pset_fapl_stdio` sets the file access properties to use the STDIO driver.

```
herr_t H5Pset_fapl_stdio (hid_t fapl_id)
```

Any previously defined driver properties are erased from the property list.

Additional parameters may be added to this function in the future. Since there are no additional variable settings associated with the STDIO driver, there is no `H5Pget_fapl_stdio` function.

8.4 File families -- FAMILY driver

HDF5 files can become quite large, creating problems on systems that do not support files larger than 2 gigabytes. The HDF5 file family mechanism is designed to solve the problems this creates by simply splitting the HDF5 file address space across several smaller files. This structure does nothing to segregate meta data and raw data; they are mixed in the address space just as they would be in a single contiguous file.

HDF5 applications access such a family of files via the FAMILY driver, `H5FD_FAMILY`. The functions `H5Pset_fapl_family` and `H5Pget_fapl_family` are used to manage file family properties:

```
herr_t H5Pset_fapl_family (hid_t fapl_id, hsize_t memb_size,
         hid_t member_properties)
herr_t H5Pget_fapl_family (hid_t fapl_id, hsize_t *memb_size,
         hid_t *member_properties)
```

Each member of the family is the same logical size, though the size and disk storage reported by file system listing tools (e.g., `'ls -l'` on a UNIX system or the detailed folder listing on a Macintosh or Microsoft Windows system) 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 numbered zero (`0`).

H5Pset_fapl_family sets the access properties to use the FAMILY driver; any previously defined driver properties are erased from the property list. *member_properties* will serve as the file access property list for each member of the file family. *memb_size* specifies the logical size, in bytes, of each family member. *memb_size* is used only when creating a new file or truncating an existing file; otherwise the member size is determined by the size of the first member of the family being opened. Note: If the size of the off_t type is four bytes, the maximum family member size is usually 2^31-1 because the byte at offset 2,147,483,647 is generally inaccessible.

H5Pget_fapl_family is used to retrieve file family properties. If the file access property list is set to use the FAMILY driver, *member_properties* will be returned with a pointer to a copy of the appropriate member access property list. If *memb_size* is non-null, it will contain the logical size, in bytes, of family members.

Additional parameters may be added to these functions in the future.

UNIX tools and an HDF5 utility

It occasionally becomes necessary to repartition a file family. A command-line utility for this purpose, h5repart, is distributed with the HDF5 library.

h5repart [-v] [-b *block_size*[*suffix*]] [-m *member_size*[*suffix*]] *source destination*

h5repart repartitions an HDF5 file by copying the source file or file family to the destination file or file 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). *block_size* and *member_size* may be suffixed with the letters g, m, or k for GB, MB, or kB respectively.

The h5repart utility is fully described on the Tools page of the *HDF5 Reference Manual*.

An existing HDF5 file can be split into a family of files by running the file through split(1) on a UNIX system and numbering the output files. However, the HDF5 library is lazy about extending the size of family members, so a valid file cannot generally be created by concatenation of the family members.

Splitting the file and rejoining the segments by concatenation (split(1) and cat(1) on UNIX systems) does not generate files with holes; holes are preserved only through the use of h5repart.

8.5 Multiple meta data and raw data files -- MULTI driver

In some circumstances, it is useful to separate meta data from raw data and some types of meta data from other types of meta data. Situations that would benefit from use of the MULTI driver include the following:

- In networked situations where the small meta data files can be kept on local disks but larger raw data files must be stored on remote remote media
- In cases where the raw data is extremely large
- In situations requiring frequent access to meta data held in RAM while the raw data can be efficiently held on disk.

In either case, access to the meta data is substantially easier with the smaller, and possibly more localized, meta data files. This often results in improved application performance.

The MULTI driver, `H5FD_MULTI`, provides a mechanism for segregating raw data and different types of meta data into multiple files. The functions `H5Pset_fapl_multi` and `H5Pget_fapl_multi` are used to manage access properties for these multiple files:

```
herr_t H5Pset_fapl_multi (hid_t fapl_id, const H5FD_mem_t *memb_map,
         const hid_t *memb_fapl, const char * const *memb_name,
         const haddr_t *memb_addr, hbool_t relax)
herr_t H5Pget_fapl_multi (hid_t fapl_id, const H5FD_mem_t *memb_map,
         const hid_t *memb_fapl, const char **memb_name,
         const haddr_t *memb_addr, hbool_t *relax)
```

`H5Pset_fapl_multi` sets the file access properties to use the MULTI driver; any previously defined driver properties are erased from the property list. With the MULTI driver invoked, the application will provide a base name to `H5Fopen` or `H5Fcreate`. The files will be named by that base name as modified by the rule indicated in *`memb_name`*. File access will be governed by the file access property list *`memb_properties`*.

See H5Pset_fapl_multi and H5Pget_fapl_multi in the *HDF5 Reference Manual* for complete descriptions of these functions and their usage.

Additional parameters may be added to these functions in the future.

8.6 Split meta data and raw data files -- SPLIT driver

The SPLIT driver, `H5FD_SPLIT`, is a limited case of the MULTI driver, creating exactly two files: one containing all the meta data and another for raw data.

The function `H5Pset_fapl_split` is used to manage SPLIT file access properties:

```
herr_t H5Pset_fapl_split (hid_t access_properties, const char
*meta_extension, hid_t meta_properties, const char *raw_extension,
hid_t raw_properties
```

`H5Pset_fapl_split` sets the file access properties to use the SPLIT driver; any previously defined driver properties are erased from the property list.

With the SPLIT driver invoked, the application will provide a base file name, *file_name* to H5Fcreate or H5Fopen. The meta data and raw data files in storage will then be named *file_name.meta_extension* and *file_name.raw_extension*, respectively. For example, if *meta_extension* is defined as .meta and *raw_extension* is defined as .raw, the final filenames will be *file_name*.meta and *file_name*.raw.

Each file can have its own file access property list. This allows the creative use of other low-level file drivers. For instance, the meta data file can be held in RAM and accessed via the CORE driver while the raw data file is stored on disk and accessed via the SEC2 driver. Meta data file access will be governed by the file access property list in *meta_properties*. Raw data file access will be governed by the file access property list in *raw_properties*.

Additional parameters may be added to these functions in the future. Since there are no additional variable settings associated with the SPLIT driver, there is no H5Pget_fapl_split function.

8.7 Parallel I/O with MPI I/O -- MPI driver

Most of the low-level file drivers described here are for use with serial applications on serial systems. Parallel environments, on the other hand, require a parallel low-level driver. HDF5 relies on MPI I/O in parallel environments and the MPI driver, H5FD_MPI, for parallel file access.

The functions H5Pset_fapl_mpio and H5Pget_fapl_mpio are used to manage parallel file access properties.

```
herr_t H5Pset_fapl_mpio (hid_t fapl_id, MPI_Comm comm,
         MPI_info info)
herr_t H5Pget_fapl_mpio (hid_t fapl_id, MPI_Comm *comm,
         MPI_info *info)
```

The file access properties managed by H5Pset_fapl_mpio and retrieved by H5Pget_fapl_mpio are the MPI communicator, *comm*, and the MPI info object, *info*.

comm is the MPI communicator to be used for file open. info is the MPI info object, an information object much like an HDF5 property list, to be used for file open. Both are defined in MPI_FILE_OPEN of MPI-2.

The communicator and the info object are saved in the file access property list *fapl_id*. *fapl_id* can then be passed to MPI_File_open to create and/or open the file.

This function does not create duplicate *comm* or *info* objects. Any modification to either object after this function call returns may have an undetermined effect on the access property list; users should not modify either of the *comm* or *info* objects while they are defined in a property list.

H5Pset_fapl_mpio and H5Pget_fapl_mpio are available only in the parallel HDF5 library and are not collective functions. The MPI driver is available only in the parallel HDF5 library.

Additional parameters may be added to these functions in the future.

8.8 Buffered temporary files in memory -- CORE driver

There are several situations in which it is it is reasonable, sometimes even required, to maintain a file entirely in system memory. You might want to do so if, for example, either of the following conditions apply:

- Performance requirements are so stringent that disk latency is a limiting factor.
- You are working with small, temporary files that will not be retained and, thus, need not be written to storage media.

The CORE driver, `H5FD_CORE`, provides a mechanism for creating and managing such in-memory files. The functions `H5Pset_fapl_core` and `H5Pget_fapl_core` manage CORE file access properties:

```
herr_t H5Pset_fapl_core (hid_t access_properties,
          size_t block_size, hbool_t backing_store)
herr_t H5Pget_fapl_core (hid_t access_properties,
          size_t *block_size), hbool_t *backing_store)
```

`H5Pset_fapl_core` sets the file access property list to use the CORE driver; 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`*.

`backing_store` is a boolean flag indicating whether to write the file contents to disk when the file is closed. If *`backing_store`* is set to `1` (`TRUE`), the file contents are flushed to a file with the same name as the CORE file when the file is closed or access to the file is terminated in memory. If *`backing_store`* is set to `0` (`FALSE`), the file is not saved.

If the file access property list is set to use the CORE driver, `H5Pget_fapl_core` will return *`block_size`* and *`backing_store`* with the relevant file access property settings.

Note the following important points regarding in-memory files:

- Local temporary files are created and accessed directly from memory without ever being written to disk.
- Total file size must not exceed the available virtual memory.
- Only one HDF5 file identifier can be opened for the file, the identifier returned by `H5Fcreate`. The name assigned in `H5Fcreate` is used only if *`backing_store`* is set and cannot be used with `H5Fopen` to obtain a new file identifier. `H5Fopen` will always fail.
- The file will be discarded when access is terminated unless *`backing_store`* is set to `1`.

Additional parameters may be added to these functions in the future.

8.9 Streaming I/O -- STREAM driver

The STREAM driver is designed for situations where data is to be streamed across the network rather than written to a local file.

The functions `H5Pset_fapl_stream` and `H5Pget_fapl_stream` are used to manage streaming file access properties:

```
herr_t H5Pset_fapl_stream (hid_t fapl_id, H5FD_stream_fapl_t *fapl)
herr_t H5Pget_fapl_stream (hid_t fapl_id, H5FD_stream_fapl_t *fapl)
```

`H5Pset_fapl_stream` sets up the use of the STREAM driver.

`fapl_id` is the identifier for the file access property list currently in use.

`fapl` is the streaming file access property list and is an `H5FD_stream_fapl_t` struct containing the following elements:

```
size_t                    increment
H5FD_STREAM_SOCKET_TYPE   socket
hbool_t                   do_socket_io
unsigned int              backlog
H5FD_stream_broadcast_t   broadcast_fn
void *                    broadcast_arg
```

- *`increment`* specifies how much memory to allocate each time additional memory is required.
- *`socket`* is an external socket descriptor; if a valid socket argument is provided, that socket will be used.
- *`do_socket_io`* is a boolean value specifying whether to perform I/O on *`socket`*.
- *`backlog`* is the argument for the `listen` call.
- *`broadcast_fn`* is the broadcast callback function.
- *`broadcast_arg`* is the user argument to the broadcast callback function.

`H5Pget_fapl_stream` retrieves the values stored in the *`fapl`* struct.

`H5Pset_fapl_stream` and `H5Pget_fapl_stream` are not intended for use in parallel environments.

8.10 Access logging -- LOG driver

The LOG driver, `H5FD_LOG`, is designed for situations where it is necessary to log file access activity.

The function `H5Pset_fapl_log` is used to manage logging properties:

```
herr_t H5Pset_fapl_log (hid_t fapl_id, const char *logfile, unsigned
int flags, size_t buf_size)
```

`H5Pset_fapl_log` sets the file access property list to use the LOG driver. File access characteristices are identical to access via the SEC2 driver. Any previously defined driver properties are erased from the property list.

Log records are written to the file *`logfile`*.

The following values of *`verbosity`* set the indicated logging levels:

`0`	Performs no logging.
`1`	Records where writes and reads occur in the file.
`2`	Records where writes and reads occur in the file and what kind of data is written at each location: raw data or any of several types of metadata (object headers, superblock, B-tree data, local headers, or global headers).

There is no `H5Pget_fapl_log` function.

Additional parameters may be added to this function in the future.

9. Code Examples for Opening and Closing Files

9.1 Example using the H5ACC_TRUNC flag

The following example creates a new file with the default file creation and file access properties. Since H5Fcreate is called with the H5ACC_TRUNC flag, any existing file content is overwritten if the file already exists, i.e., it is truncated. If H5Fcreate should fail if the file already exists, use the flag H5ACC_TRUNC instead of H5ACC_TRUNC.

```
hid_t file;                                  /*  identifier   */

/* 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);

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

9.2 Example with file creation property list

This 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);
```

9.3 Example with file access property list

This 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_fapl_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);
```

Chapter 3
HDF5 Groups

1. Introduction

As suggested by the name Hierarchical Data Format, an HDF5 file is hierarchically structured. The HDF5 group and link objects implement this hierarchy.

In the simple and most common case, the file structure is a tree structure; in the general case, the file structure may be a directed graph with a designated entry point. The tree structure is very similar to the file system structures employed on UNIX systems, directories and files, and on Apple Macintosh and Microsoft Windows systems, folders and files. HDF5 groups are analogous to the directories and folders; HDF5 datasets are analogous to the files.

The one very important difference between the HDF5 file structure and the above-mentioned file system anologs is that HDF5 groups are linked as a directed graph, allowing circular references; the file systems are strictly hierarchical, allowing no circular references. The figures below illustrate the range of possibilities.

In Figure 1, the group structure is strictly hierarchical, identical to the file system analogs.

In Figures 2 and 3, the structure takes advantage of the directed graph's allowance of circular references. In Figure 2, `GroupA` is not only a member of the root group, `/`, but a member of `GroupC`. Since Group C is a member of Group B and Group B is a member of Group A, Dataset1 can be accessed by means of the circular reference `/Group A/Group B/Group C/Group A/Dataset1`. Figure 3 illustrates an extreme case in which `GroupB` is a member of itself, enabling a reference to a member dataset such as `/Group A/Group B/Group B/Group B/Dataset2`.

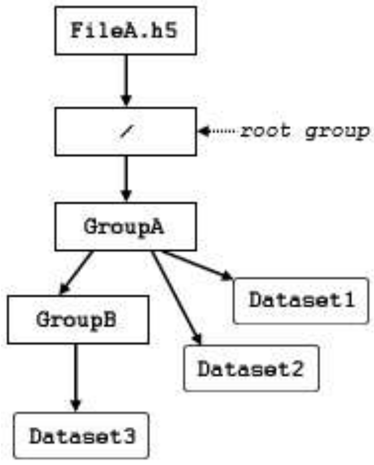


Figure 1: An HDF5 file with a strictly hierarchical group structure

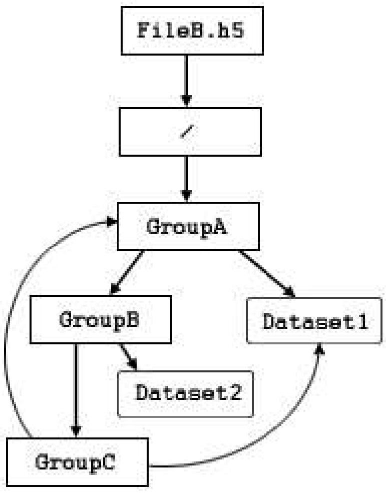


Figure 2: An HDF5 file with a directed graph group structure, including a circular reference

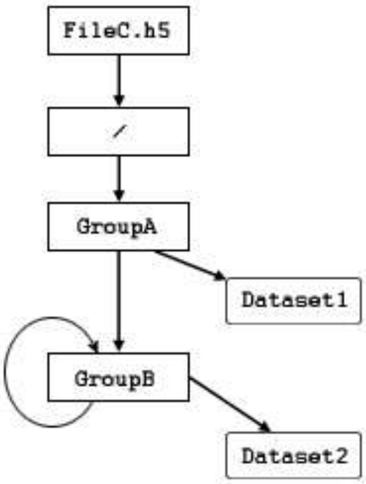


Figure 3: An HDF5 file with a directed graph group structure and one group as a member of itself

As becomes apparent upon reflection, directed graph structures can become quite complex; caution is advised!

The balance of this chapter discusses the following topics:

- The HDF5 group object (or a group) and its structure in more detail
- HDF5 link objects (or links)
- The programming model for working with groups and links
- HDF5 functions provided for working with groups, group members, and links
- Retrieving information about objects in a group
- Discovery of the structure of an HDF5 file and the contained objects
- Examples of file structures

2. Description of the Group Object

2.1 The Group Object

Abstractly, an HDF5 group contains zero or more objects and every object must be a member of at least one group. The root group, the sole exception, may not belong to any group.

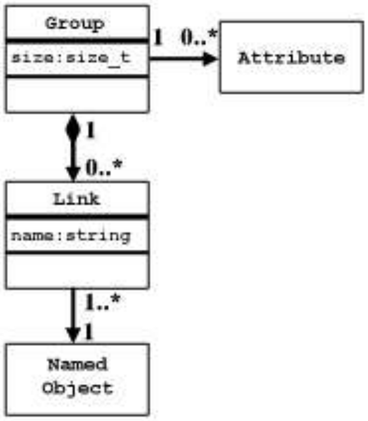


Figure 4: Abstract model of the HDF5 group object

Group membership is actually implemented via *link* objects (see Figure 4). A link object is owned by a group and points to a *named object*. Each link has a *name*, and each link points to exactly one object. Each named object has at least one and possibly many links to it.

There are three classes of named objects: *group*, *dataset*, and *named datatype* (see Figure 5). Each of these objects is the member of at least one group, which means there is at least one link to it.

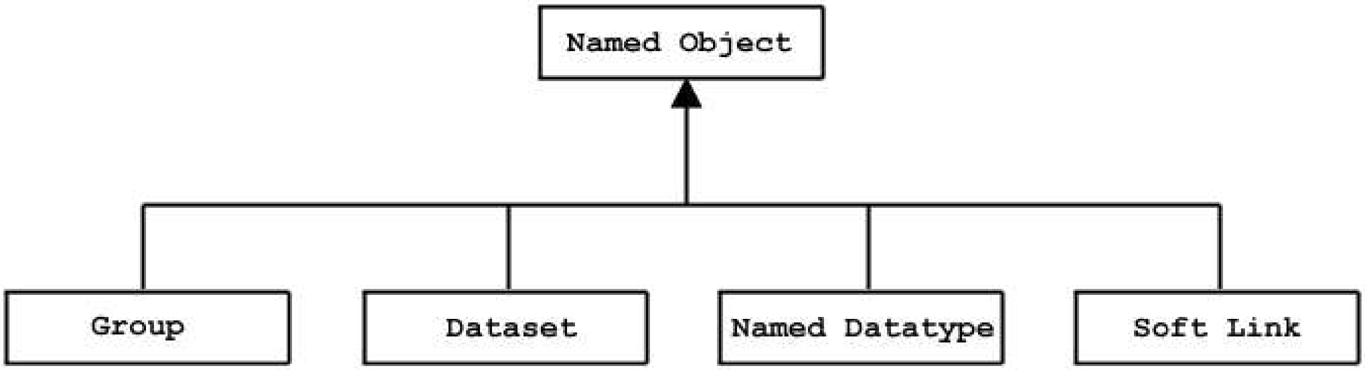


Figure 5:

The primary operations on a group are to add and remove members and to discover member objects. These abstract operations, as listed in Figure 6, are implemented in the H5G APIs, as listed in section 4, “Group Function Summaries.”

To add and delete members of a group, links from the group to *existing* objects in the file are created and deleted with the *`link`* and *`unlink`* operations. When a *new* named object is created, the HDF5 library executes the link operation in the background immediately after creating the object (i.e., a new object is added as a member of the group in which it is created without further user intervention).

Given the name of an object, the *get_object_info* method retrieves a description of the object, including the number of references to it. The *iterate* method iterates through the members of the group, returning the name and type of each object.

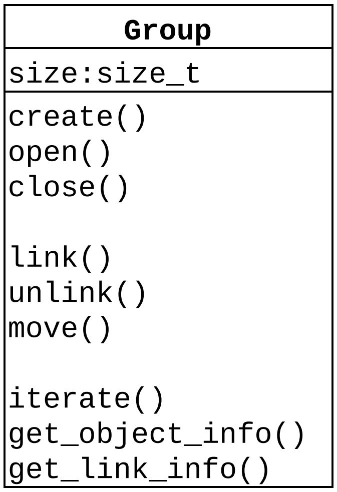


Figure 6:
The group object

Every HDF5 File has a single root group, with the name `/`. The root group is identical to any other HDF5 group, except:

- The root group is automatically created when the HDF5 file is created (`H5Fcreate`).
- The root group has no parent, but, by convention has a reference count of 1.
- The root group cannot be deleted (i.e., unlinked)!

2.2 The Hierarchy of Data Objects

An HDF5 file is organized as a rooted, directed graph using HDF5 group objects. The named data objects are the nodes of the graph, and the links are the directed arcs. Each arc of the graph has a name, with the special name `/` reserved for the root group. New objects are created and then inserted into the graph with a link operation tht is automatically executed by the library; existing objects are inserted into the graph with a link operation explicitly called by the user, which creates a named link from a group to the object. An object can be the target of more than one link.

The names on the links must be unique within each group, but there may be many links with the same name in different groups. These are unambiguous, because some ancestor must have a different name, or else they are the same object. The graph is navigated with path names, analogous to Unix file systems (see section 2.3, “HDF5 Path Names”). An object can be opened with a full path starting at the root group, or with a relative path and a starting point. That starting point is always a group, though it may be the current working group, another specified group, or the root group of the file. Note that all paths are relative to a single HDF5 File. In this sense, an HDF5 file is analogous to a single UNIX file system. [1]

It is important to note that, just like the UNIX file system, HDF5 objects do not have *names*, the names are associated with *paths*. An object has an *object identifier* that is unique within the file, but a single object may have many *names* because there may be many paths to the same object. An object can be renamed, or moved to another group, by adding and deleting links. In this case, the object itself never moves. For that matter, membership in a group has no implication for the physical location of the stored object.

Deleting a link to an object does not necessarily delete the object. The object remains available as long as there is at least one link to it. After all links to an object are deleted, it can no longer be opened, although the storage may or may not be reclaimed. [2]

It is also important to realize that the linking mechanism can be used to construct very complex graphs of objects. For example, it is possible for object to be shared between several groups and even to have more than one name in the same group. It is also possible for a group to be a member of itself, or to create other *cycles* in the graph, such as in the case where a child group is linked to one of its ancestors.

HDF5 also has *soft links* similar to UNIX soft links. A *soft link* is an object that has a name and a path name for the target object. The soft link can be followed to open the target of the link just like a regular or *hard* link. The differences are that the hard link cannot be created if the target object does not exist and it always points to the same object. A soft link can be created with any path name, whether or not the object exists; it may or may not, therefore, be possible to follow a soft link. Furthermore, a soft link's target object may be changed.

2.3 HDF5 Path Names

The structure of the HDF5 file constitutes the name space for the objects in the file. A path name is a string of components separated by slashes (/). Each component is the name of a hard or soft link which points to an object in the file. The slash not only separates the components, but indicates their hierarchical releationship; the component indicated by the link name following a slash is a always a member of the component indicated by the link name preceding that slash.

The first component in the path name may be any of the following:

- the special character dot (., a period), indicating the current group
- the special character slash (/), indicating the root group
- any member of the current group

Component link names may be any string of ASCII characters not containing a slash or a dot (/ and ., which are reserved as noted above). However, users are advised to avoid the use of punctuation and non-printing characters, as they may create problems for other software. Figure 7 provides a BNF grammar for HDF5 path names.

```
PathName ::= AbsolutePathName | RelativePathName
Separator ::= "/" ["/"]*
AbsolutePathName ::= Separator [ RelativePathName ]
RelativePathName ::= Component [ Separator RelativePathName ]*
Component ::=  "." |  Characters
Characters ::= Character+   -  { "." }
Character ::= {c:  c Î { { legal ASCII characters } - {'/'} }
```

Figure 7: A BNF grammar for for HDF5 path names

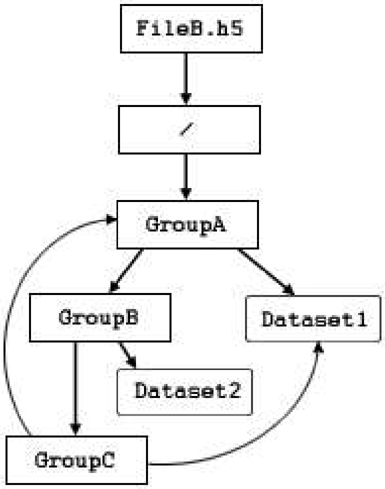


Figure 8: An HDF5 file with a directed graph group structure, including a circular reference

An object can always be addressed by a either a *full or absolute* path name, starting at the root group, or by a *relative* path name, starting in a known location such as the current working group. As noted elsewhere, a given object may have multiple full and relative path names.

Consider, for example, the file illustrated in Figure 8. `Dataset1` can be identified by either of these absolute path names:

```
/GroupA/Dataset1
/GroupA/GroupB/GroupC/Dataset1
```

Since an HDF5 file is a directed graph structure, and is therefore not limited to a strict tree structure, and since this illustrated file includes the sort of circular reference that a directed graph enables, `Dataset1` can also be identified by this absolute path name:

```
/GroupA/GroupB/GroupC/GroupA/Dataset1
```

Alternatively, if the current working location is `GroupB`, `Dataset1` can be identified by either of these relative path names:

```
GroupC/Dataset1
GroupC/GroupA/Dataset1
```

Note that relative path names in HDF5 do not employ the `../` notation, the UNIX notation indicating a parent directory, to indicate a parent group.

3. Using h5dump

You can use h5dump, the command-line utility distributed with HDF5, to examine a file for purposes either of determining where to create an object within an HDF5 file or to verify that you have created an object in the intended place. inspecting the contents of an HDF5 file.

In the case of the new group created in section 5.1, “Creating a group,” the following h5dump command will display the contents of FileA.h5:

h5dump FileA.h5

Assuming that the discussed objects, GroupA and GroupB are the only objects that exist in FileA.h5, the output will look something like the following:

```
HDF5 "FileA.h5" {
GROUP "/" {
GROUP GroupA {
GROUP GroupB {
}
}
}
}
```

h5dump is fully described on the Tools page of the *HDF5 Reference Manual*. The HDF5 DDL grammar is fully described in the document DDL in BNF for HDF5, an element of the *HDF5 User's Guide*.

4. Group Function Summaries (H5G)

C Function F90 Function	Purpose
`H5Gcreate` `h5gcreate_f`	Creates a new empty group and gives it a name.
`H5Gopen` `h5gopen_f`	Opens an existing group for modification and returns a group identifier for that group.
`H5Gclose` `h5gclose_f`	Closes the specified group.
`H5Gset_comment` `h5gset_comment_f`	Sets the comment for the specified object.
`H5Gget_comment` `h5gget_comment_f`	Retrieves the comment for the specified object.
`H5Glink` `h5glink_f`	Creates a link of the specified type from a new name to a current name.
`H5Glink2` `h5glink2_f`	Creates a link of the specified type from a new name to a current name.
`H5Gunlink` `h5gunlink_f`	Removes a link to an object from a group.
`H5Gmove` `h5gmove_f`	Renames an object within an HDF5 file.
`H5Gmove2` `h5gmove2_f`	Renames an object within an HDF5 file.

`H5Giterate`
`(none)`
	Iterates an operation over the entries of a group.

`(none)`
`h5gget_obj_info_idx_f`
	Returns the name and type of a specified group member.

`(none)`
`h5gn_members_f`
	Returns the number of group members.

`H5Gget_objinfo`
`(none)`
	Returns information about an object.

`H5Gget_num_objs`
`(none)`
	Returns number of objects in the specified group.

`H5Gget_objname_by_idx`
`(none)`
	Returns a name of an object specified by its index.

`H5Gget_objtype_by_idx`
`(none)`
	Returns the type of an object specified by its index.

`H5Gget_linkval` `h5gget_linkval_f`	Returns the name of the object that the specified symbolic link points to.

5. Programming Model: Working with Groups

The programming model for working with groups is as follows:

1. Create a new group or open an existing one.
2. Perform the desired operations on the group.
    - Create new objects in the group.
    - Insert existing objects as group members.
    - Delete existing members.
    - Open and close member objects.
    - Access information regarding member objects.
    - Iterate across group members.
    - Manipulate links.
3. Terminate access to the group. (Close the group.)

5.1 Creating a Group

To create a group, use `H5Gcreate`, specifying the location and the path of the new group. The location is the identifier of the file or the group in a file with respect to which the new group is to be identified. The path is a string that provides wither an absolute path or a relative path to the new group (see section 2.3, "HDF5 Path Names"). A path that begins with a slash (`/`) is an absolute path indicating that it locates the new group from the root group of the HDF5 file. A path that begins with any other character is a relative path. When the location is a file, a relative path is a path from that file's root group; when the location is a group, a relative path is a path from that group.

The sample code in Figure 9 creates three groups. The group `Data` is created in the root directory; two groups are then created in `/Data`, one with absolute path, the other with a relative path.

```
hid_t file;
file = H5Fopen(....);

group = H5Gcreate(file, "/Data", 0);
group_new1 = H5Gcreate(file, "/Data/Data_new1", 0);
group_new2 = H5Gcreate(group, "Data_new2", 0);
```

Figure 9: Creating three new groups

The third `H5Gcreate` parameter 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, a default size is chosen.

5.2 Opening a group and accessing an object in that group

Though it is not always necessary, it is often useful to explicitely open a group when working with objects in that group. Using the file created in the example above, Figure 10 illustrates the use of a previously-acquired file identifier and a path relative to that file to open the group `Data`.

Any object in a group can be also accessed by its absolute or relative path. To open an object using a relative path, an application must first open the group or file on which that relative path is based. To open an object using an absolute path, the application can use any location identifier in the same file as the target object; the file identifier is commonly used, but object identifier for any object in that file will work. Both of these approaches are illustrated in Figure 10 offers example code in the first two lines to open a group then open a dataset with the appropriate relative path to open the same dataset with an abslolute path and .

Using the file created in the examples above, Figure 10 provides example code illustrating the use of both relative and absolute paths to access an HDF5 data object. The first sequence (two function calls) uses a previously-acquired file identifier to open the group `Data` then uses the returned group identifier and a relative path to open the dataset `CData`. The second approach (one function call) uses the same previously-acquired file identifier and an absolute path to open the same dataset.

```
group = H5Gopen(file, "Data");
dataset1 = H5Dopen(group, "CData");

dataset2 = H5Dopen(file, "/Data/CData");
```

Figure 10: Open a dataset with relative and absolute paths

5.3 Creating a dataset in a specific group

Any dataset must be created in a particular group. As with groups, a dataset may be created in a particular group by specifying its absolute path or a relative path. Figure 11 illustrates both approaches to creating a dataset in the group `/Data`.

```
dataspace = H5Screate_simple(RANK, dims, NULL);
dataset1 = H5Dcreate(file, "/Data/CData", H5T_NATIVE_INT,
                  dataspace, H5P_DEFAULT);

group = H5Gopen(file, "Data");
dataset2 = H5Dcreate(group, "Cdata2", H5T_NATIVE_INT,
                  dataspace, plist);
```

Figure 11: Create a dataset with absolute and relative paths

5.4 Closing a group

To ensure the integrity of HDF5 objects and to release system resources, an application should always call the appropriate close function when it is through working with an HDF5 object. In the case of groups, `H5Gclose` ends access to the group and releases any resources the HDF5 library has maintained in support of that access, including the group identifier.

As illustrated in Figure 12, all that is required for an `H5Gclose` call is the group identifier acquired when the group was opened; there are no relative versus absolute path considerations.

```
        herr_t status;
        status = H5Gclose(group);
```

Figure 12: Close a group

A non-negative return value indicates that the group was successuflly closed and the resources released; a negative return value indicates that the attempt to close the group or release resources failed.

5.5 Creating Links

As previously mentioned, every object is created in a specific group. Once created, an object can be made a member of additional groups by means of links created with `H5Glink` or `H5Glink2`.

A link is, in effect, is a path by which the target object can be accessed; it therefore has a name which functions as a single path component. A link can be removed with an `H5Gunlink` call, effectively removing the target object from the group that contained the link (assuming, of course, that the removed link was the only link to the target object in the group).

Hard links

There are two kinds of links, hard links and soft links. Hard links are reference counted; soft links are not. When an object is created, a hard link is automatically created. An object can be deleted from the file by removing all the hard links to it.

Working with the file from the previous examples, the code in Figure 13 illustrates the creation of a hard link, named `Data_link`, in the root group, `/`, to the group `Data`. Once that link is created, the dataset `Cdata` can be accessed via either of two absolute paths, `/Data/Cdata` or `/Data_Link/Cdata`.

```
          status = H5Glink(file, H5G_LINK_HARD, "Data", "Data_link");

          dataset1 = H5Dopen(file, "/Data_link/CData");
          dataset2 = H5Dopen(file, "/Data/CData");
```

Figure 13

This and subsequent examples could also use `H5Glink2`, which is used exactly like `H5Glink` except that a second location identifier is specified and the new object name is specified relative to the second location identifier.

Figure 14 shows example code to delete a link, deleting the hard link `Data` from the root group. The group `/Data` and its members are still in the file, but they can no longer be accessed via a path using the component `/Data`.

```
status = H5Gunlink(file, "Data");

dataset1 = H5Dopen(file, "/Data_link/CData");
           /*  This call should succeed; all path component still exist*/
dataset2 = H5Dopen(file, "/Data/CData");
           /*  This call will fail; the path component '/Data' has been deleted*/
```

Figure 14

When the last hard link to an object is deleted, the object is no longer accessible (although space in the file may not be deallocated). Figure 15 shows deletion of the last link, `Data_link`, to the group originally called `Data`. After the unlinking operation, the group is no longer accessible; consequently, the dataset `Cdata` is inaccessible.

```
status = H5Gunlink(file, "Data_link");

dataset = H5Dopen(file, "/Data_link/CData");
          /*  This call will fail; the dataset is no longer accessible */
```

Figure 15

Soft links

Soft links are objects that assign a name in a group to a path. Notably, the target object is determined only when the soft link is accessed, and may, in fact, not exist. Soft links are not reference counted, so there may be one or more soft links to an object.

Like hard links, soft links are also created and deleted with the `H5Glink`, `H5Glink2`, and `H5Gunlink` functions, except that soft links are created as type `H5G_LINK_SOFT` while hard links are created as type `H5G_LINK_HARD`.

Returning to our sample file as it was initially created, Figure 16 shows examples of creating two soft links to the group `/Data`.

```
status = H5Glink(file, H5G_LINK_SOFT, "Data", "Soft2");
status = H5Glink(file, H5G_LINK_SOFT, "Soft2", "Soft3");

dataset = H5Dopen(file, "/Soft2/CData");
```

Figure 16

With the soft links defined in Figure 16, the dataset `CData` in the group `/Data` can now be opened with any of the names `/Data/CData`, `/Soft2/CData`, or `/Soft2/CData`.

Not regarding hard links versus soft links
Note that an object's existence in a file is governed by the presence of at least one hard link to that object. If the last hard link to an object is removed, the object is removed from the file and any remaining soft link becomes a dangling link, a link whose target object does not exist.

Moving or renaming objects, and a warning
An object can be renamed by changing the name of a link to it with either `H5Gmove` or `H5Gmove2`. This has the same effect as creating a new link with the new name and deleting the link with the old name.

Exercise caution in the use of `H5Gmove`, `H5Gmove2` and `H5Gunlink` as these functions each include a step that unlinks a pointer to a dataset or group. If the link that is removed is on the only path leading to an HDF5 object, that object will become permanently inaccessible in the file.

Consider the following example: assume that the group `group2` can only be accessed via the following path, where `top_group` is a member of the file‘s root group:

```
/top_group/group1/group2/
```

Using `H5Gmove` or `H5Gmove2`, `top_group` is renamed to be a member of `group2`. At this point, since `top_group` was the only route from the root group to `group1`, there is no longer a path by which one can access `group1`, `group2`, or any member datasets. And since `top_group` is now a member of `group2`, `top_group` itself and any member datasets have thereby also become inaccessible.

6. Discovering Information about Objects

There is often a need to retrieve information about a particular object. The `H5Gget_objinfo` function fills this niche by returning a description of the specified object in an `H5G_stat_t` structure. The structure contains the following information:

- The file and object identifiers, which together provide unique identification of the object
- The number of references, or hard links, to the object
- The object type: group, dataset, named datatype, or soft link, returned as `H5G_GROUP`, `H5G_DATASET`, `H5G_TYPE`, or `H5G_LINK`, respectively
- The modification time (datasets only)
- A link length value; the length of the path name of a symbolic link's target object (returned for symolic links, or soft links, only)

The `H5G_stat_t` structure specification and the `H5Gget_objinfo` function signature appear in Figure 17. The `H5G_stat_t` structure elements are as listed above. The `H5Gget_objinfo` function parameters are used follows:

- *`loc_id`* specifies the object for which information being sought.
- A path to the object is returned in *`name`*.
- *`follow_link`* is a Boolean value specifying whether to follow a soft link and open the target object (`TRUE`) or not (`FALSE`).
- The `H5G_stat_t` struct is returned in the *`statbuf`* buffer.

```
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

herr_t H5Gget_objinfo(hid_t loc_id, const char *name, hbool_t follow_link, H5G_stat_t *statbuf )
```

Figure 17: The `H5G_stat_t` struct specification and the `H5Gget_objinfo` function signature

Figure 18 provides a code example that prints the local paths to the members of a group, following a soft link when it is found.

```
H5G_stat_t statbuf;

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;
case H5G_LINK:
    lname = (char *)malloc(statbuf.linklen);

     H5Gget_linkval(loc_id, name, statbuf.linklen, lname);
     printf(" Object with name %s is a link to %s \n", name, lname);
     H5Gget_objinfo(loc_id, name, TRUE, &statbuf);
     switch (statbuf.type) {
         case H5G_GROUP:
             printf(" Target of link name %s is a group \n", name);
             break;
        case H5G_DATASET:
             printf(" Target of link name %s is a dataset \n", name);
             break;
        case H5G_TYPE:
             printf(" Target of link name %s is a named datatype \n", name);
             break;
       case H5G_LINK:
            printf(" Target of link name %s is a soft link \n", name);
            break;
       default:
          printf(" Unable to identify target ");
       }
      break;
default:
     printf(" Unable to identify an object ");
}
```

Figure 18: Printing a specified object's name and type and, in the case of a link, opening the target object

7. Discovering Objects in a Group

There are two means of examining all the objects in a group. The first, `H5Giterate`, is discussed below `H5Giterate` is useful both with a single group and in an iterative process that examines an entire file or section of a file (the contents of a group, the contents of all the groups that are members of that group, etc.) and acts on objects as they are encountered.

An alternative approach is to determine the number of objects in a group then approach them one at a time. This is accomplished with the functions `H5Gget_num_objs`, `H5Gget_objname_by_idx`, and `H5Gget_objtype_by_idx`.

`H5Gget_num_objs` retrieves the number of objects, say *`n`*, in the group. The values from `0` through *`n`* `- 1` can then be used as indices to access the members of the group. For example, an index value of `0` identifies the first member, an index value of `1` identifies the second member, and an index value of *`n`* `- 1` identifies the last member. (Note that HDF5 objects do not have permanent indices; these values are strictly transient and may be different each time a group is opened.)

Using the index described above, the name and object type can be retrieved using `H5Gget_objname_by_idx` and `H5Gget_objtype_by_idx`, respectively. With the name and object type, an application can proceed to operate as necessary on all or selected group members.

8. Discovering All the Objects in the File

The structure of an HDF5 file is self-describing, meaning that an application can navigate an HDF5 file to discover and understand all the objects it contains. This is an iterative process wherein the structure is traversed as a graph, starting at one node and recursively visiting linked nodes. To explore the entire file, the traversal should start at the root group.

The function `H5Giterate`, used to discover the members of a group, is the key to the discovery process. An application calls `H5Giterate` with a pointer to a callback function (see Figure 19). The HDF5 library iterates through the group specified by the `loc_id` and `name` parameters, calling the callback function once for each group member. The callback function must have the signature defined by `H5G_iterate_t`. When invoked, the arguments to the callback function are the group being iterated, the group member's name (the object name), and a pointer set by the user program. The callback function is part of the application, so it can execute any actions the program requires to discover and store information about the objects.

```
typedef herr_t (*H5G_iterate_t)(hid_t group_id, const char *member_name,
    void *operator_data);
H5Giterate(hid_t loc_id, const char * name, int *idx, H5G_iterate_t operator,
    void *operator_data );
```

Figure 19

Note that the `H5Giterate` function is following the links from a single group; these links correspond to the components in a path name. To iterate over an entire substructure, `H5Giterate` must be recursively on every member of the original group that turns out to also be a group. To iterate over an entire file, the first call to `H5Giterate` must iterate over the root group; subsequent calls to `H5Giterate` must then iterate over every subsequent group.

Figure 20 illustrates the relationship between the calling module of the application, the callback function (`do_obj`), and calls to the HDF5 library. In this diagram, “Global Variables and Functions” symbolizes the fact that the callback function executes as part of the application, and may therefore call functions and update data structures to describe the file and its objects.

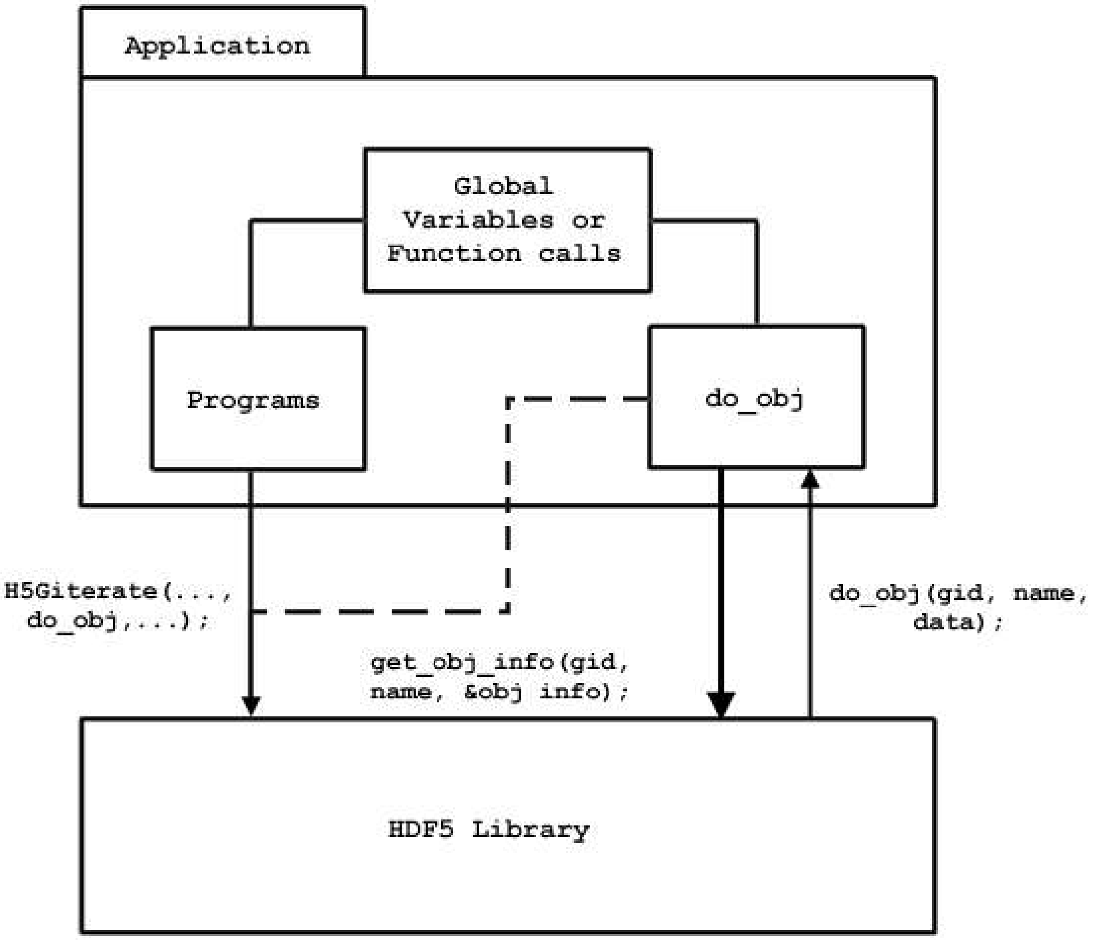

Application
Global Variables or Function calls
Programs
do_obj
H5Giterate(..., do_obj,...);
get_obj_info(gid, name, &obj info);
do_obj(gid, name, data);
HDF5 Library

Figure 20: Relationships between a calling module, the callback function, and the callback function's calls back to the HDF5 library

Figure 21 illustrates the sequence of events precipitated by an `H5Giterate` call.

1. The application first calls `H5Giterate`, passing a pointer to a callback function (`do_obj` in the figure). Note that the callback function is part of the application.
2. The HDF5 library then iterates through the members of the group, calling the callback function in the application once for each group member.
3. When the iteration is complete, the `H5Giterate` call returns to the calling application.

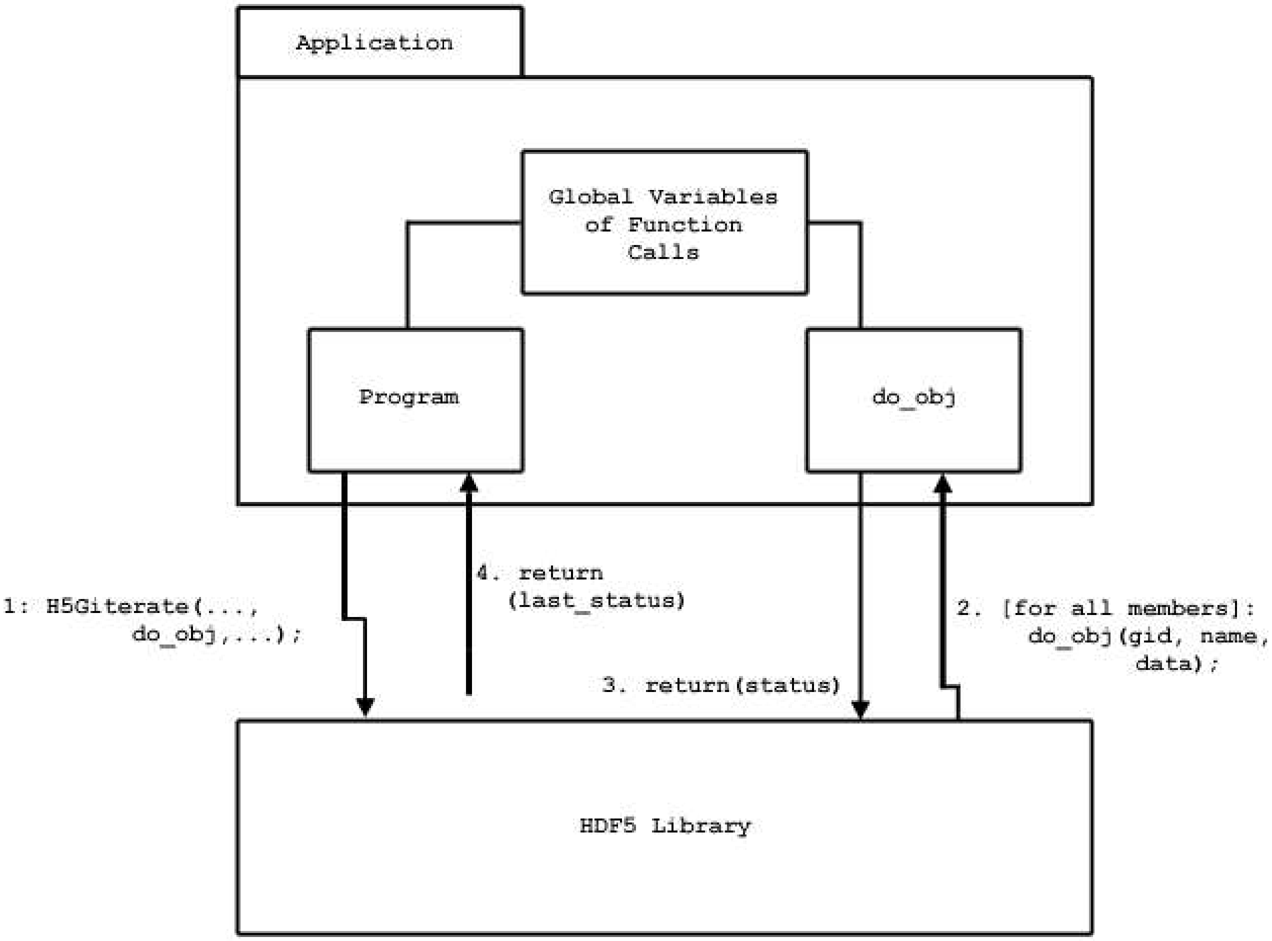

Application
Global Variables of Function Calls
Program
do_obj
1: H5Giterate(..., do_obj,...);
4. return (last_status)
3. return(status)
2. [for all members]: do_obj(gid, name, data);
HDF5 Library

Figure 21

Figure 22 shows the sequence of calls involved in one iteration of a callback function that employs the HDF5 function `H5Gget_objinfo` to discover properties of the object that is the subject of the current step of the iteration (e.g., the object's type and reference count). The HDF5 library then calls the application's callback function `do_obj()`, which in turn calls the HDF5 library to get the object information. The callback function can process the information as needed, accessing any function or data structure of the application program, and it can call the HDF5 library again to, for example, iterate through a group member that is itself a group.

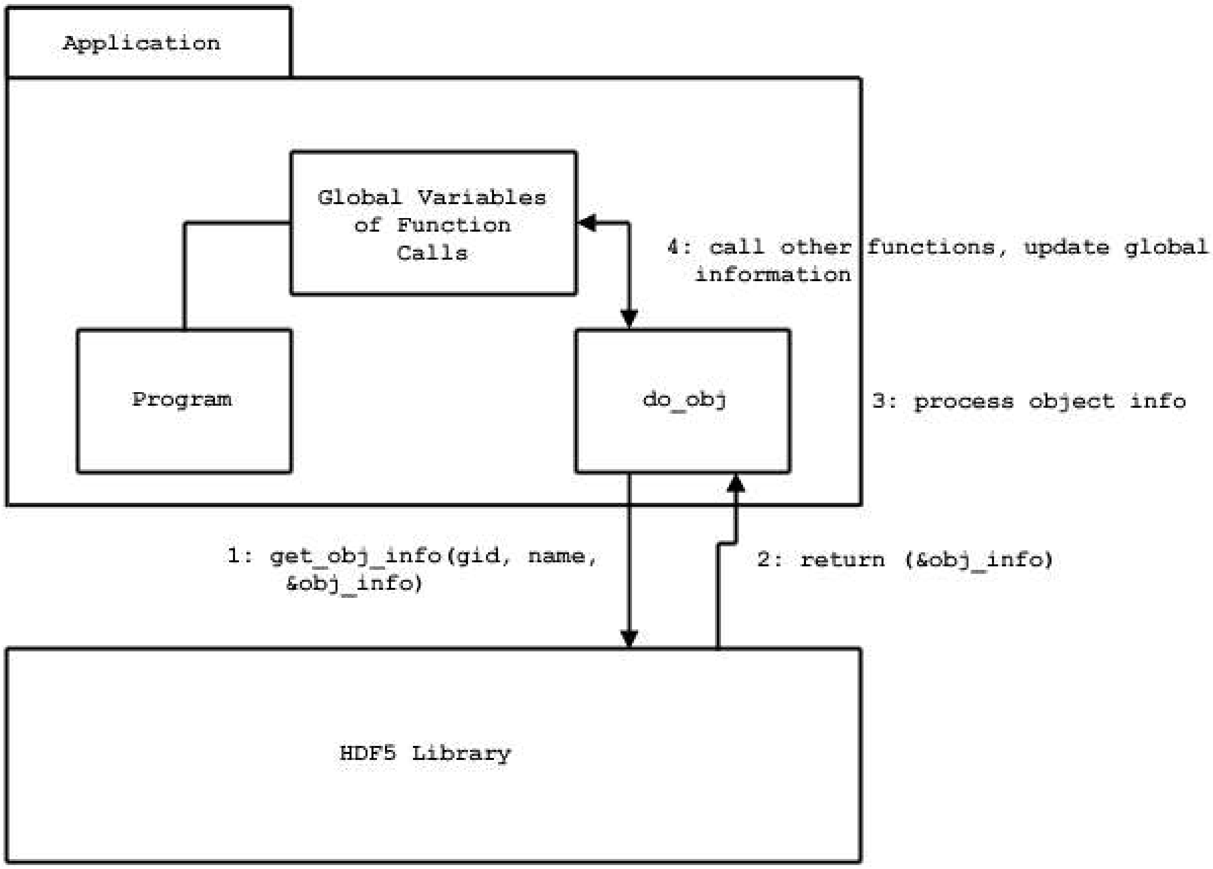

Application
Global Variables of Function Calls
4: call other functions, update global information
Program
do_obj
3: process object info
1: get_obj_info(gid, name, &obj_info)
2: return (&obj_info)
HDF5 Library

Figure 22

Over the course of a successful `H5Giterate` call, the HDF5 library will call the application's callback function once for each member of the group, as illustrated in Figure 23. At each iteration, the callback function must return a status which implies a subsequent course of action:

`1` Continue iterating.
`0` Stop iterating and return to the caller.

Once the iteration has been completed, `H5Giterate` returns to the calling application.

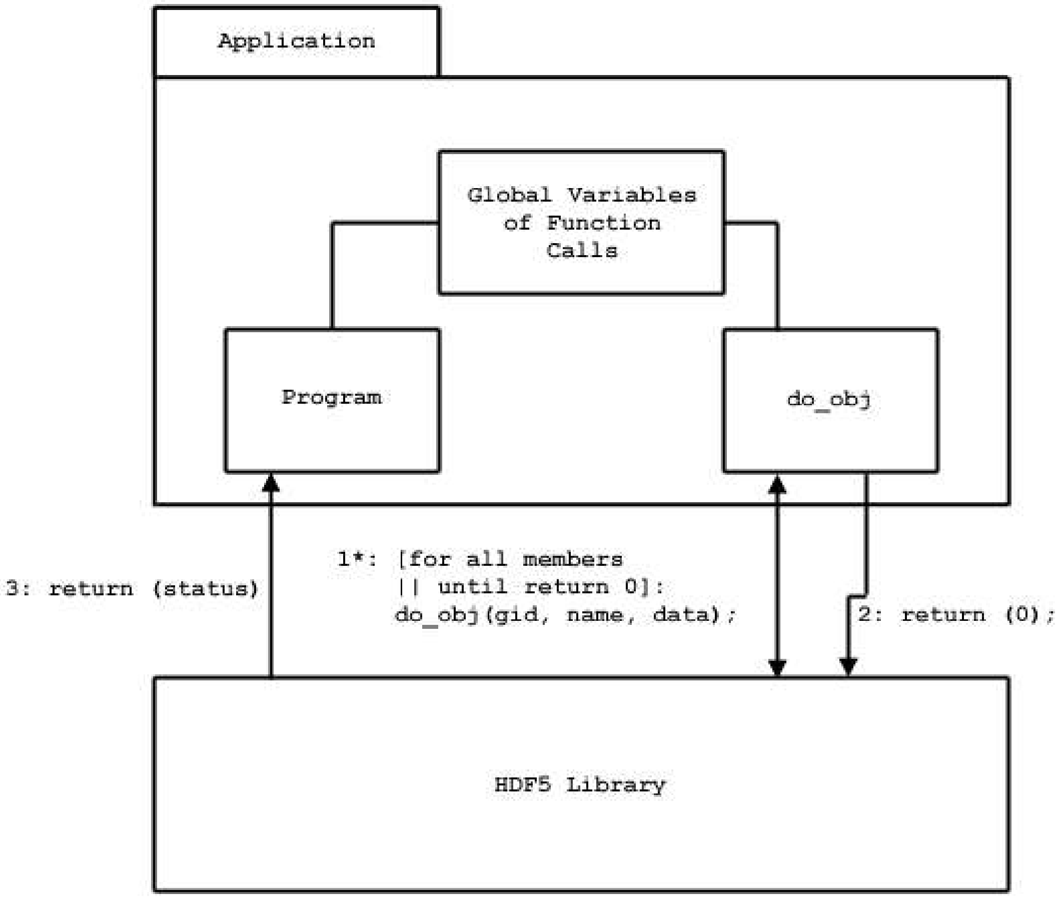

Application
Global Variables of Function Calls
Program
do_obj
3: return (status)
1*: [for all members || until return 0]: do_obj(gid, name, data);
2: return (0);
HDF5 Library

Figure 23

The overall sequence of calls can become quite complex, especially when the callback function in turn calls the HDF5 library. Figure 24 provides a sequence diagram for a case similar to the simple case described above:

1. The calling program invokes `H5Giterate` on a group,
2. which calls `do_obj` once for each group members (three group members in this case).
3. The *do_obj* callback function in turn calls `H5Gget_objinfo` each time it is invoked to discover information about each object.

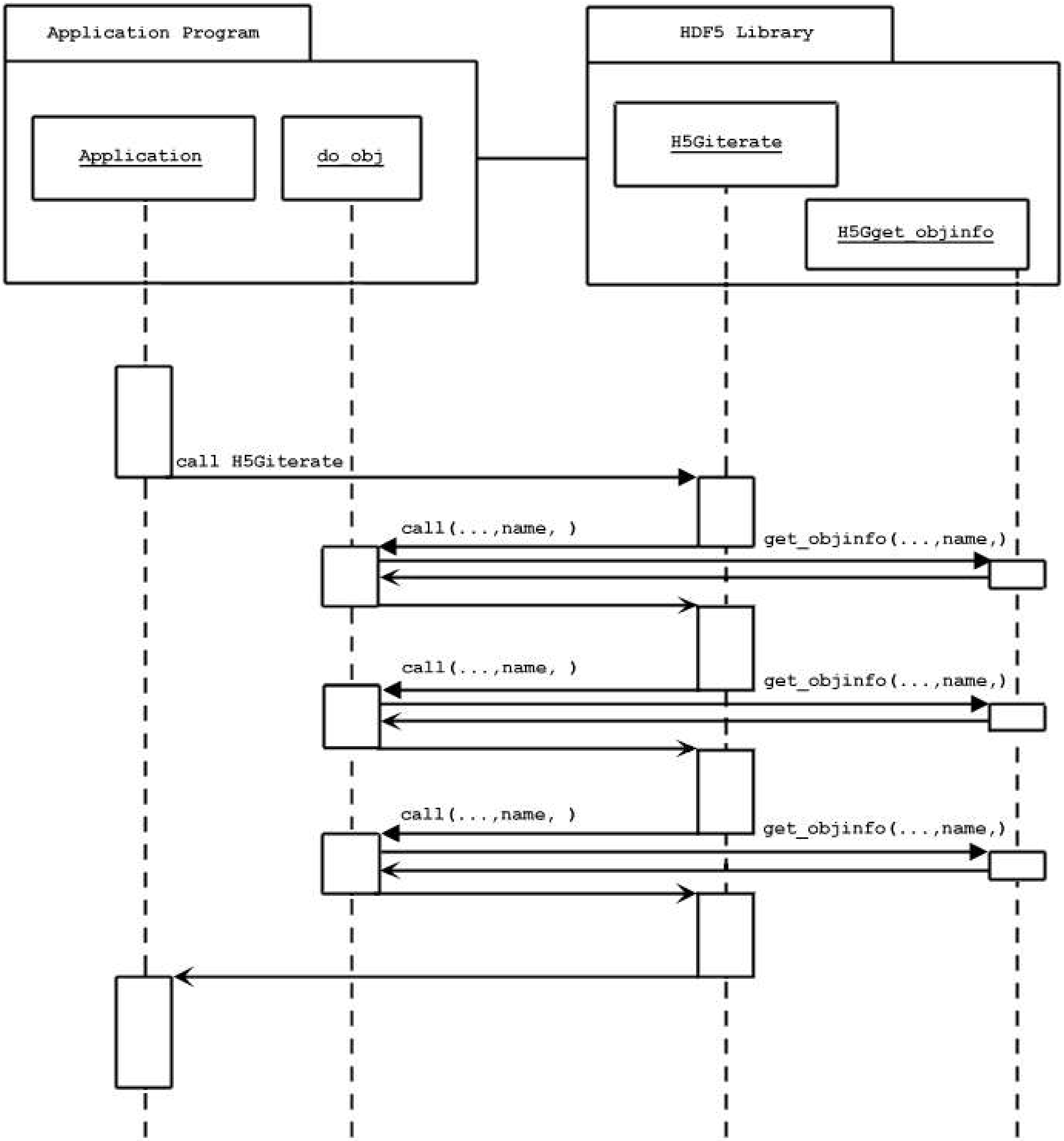
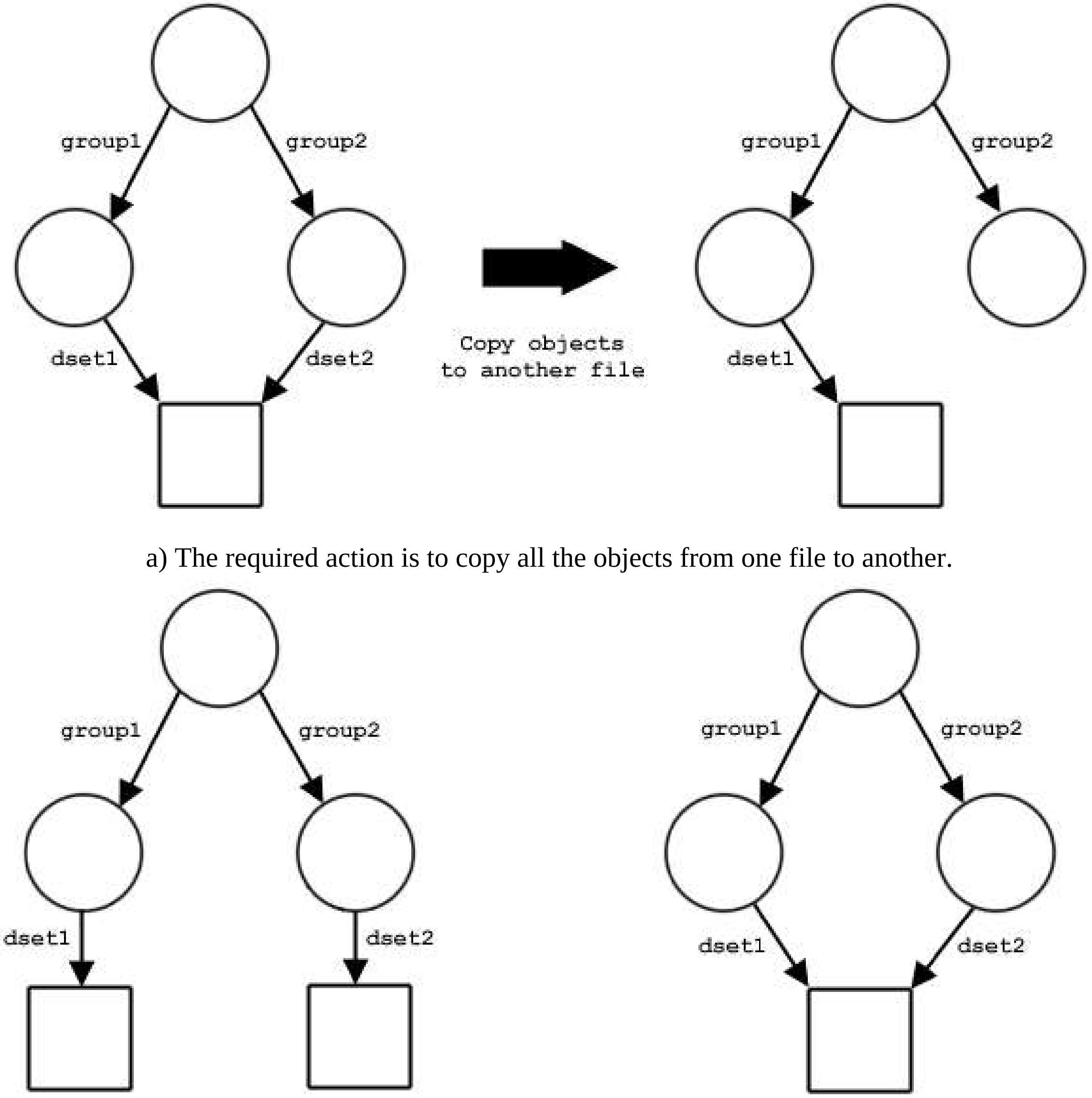

Application Program
HDF5 Library
Application
do_obj
H5Giterate
H5Gget_objinfo
call H5Giterate
call(...,name,)
get_objinfo(...,name,)
call(...,name,)
get_objinfo(...,name,)
call(...,name,)
get_objinfo(...,name,)

Figure 24

Recursively iterating through the members of every group will result in visiting an object once for each link to it. This may result in visiting an object more than once. The calling application must be prepared to recognize this case and handle it appropriately. If an action should be undertaken only once per object, the application must make sure that it does not repeat the action for an object with two links. For example, if the objects are being copied, it is important that an object with two names be copied once, not twice. Figure 25 illustrates this case.



a) The required action is to copy all the objects from one file to another.

b) A shared dataset should not be copied twice.

c) A shared dataset should be copied once and the apppropriate link should be created.

Figure 25

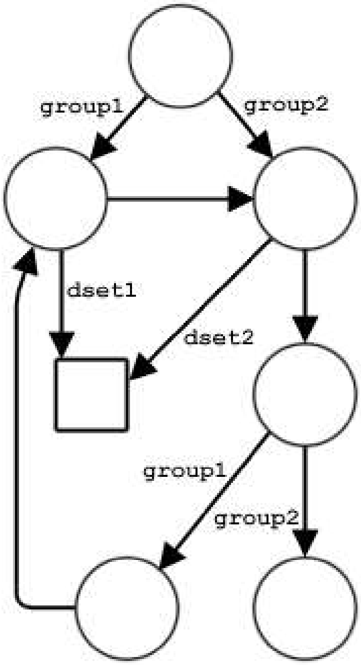


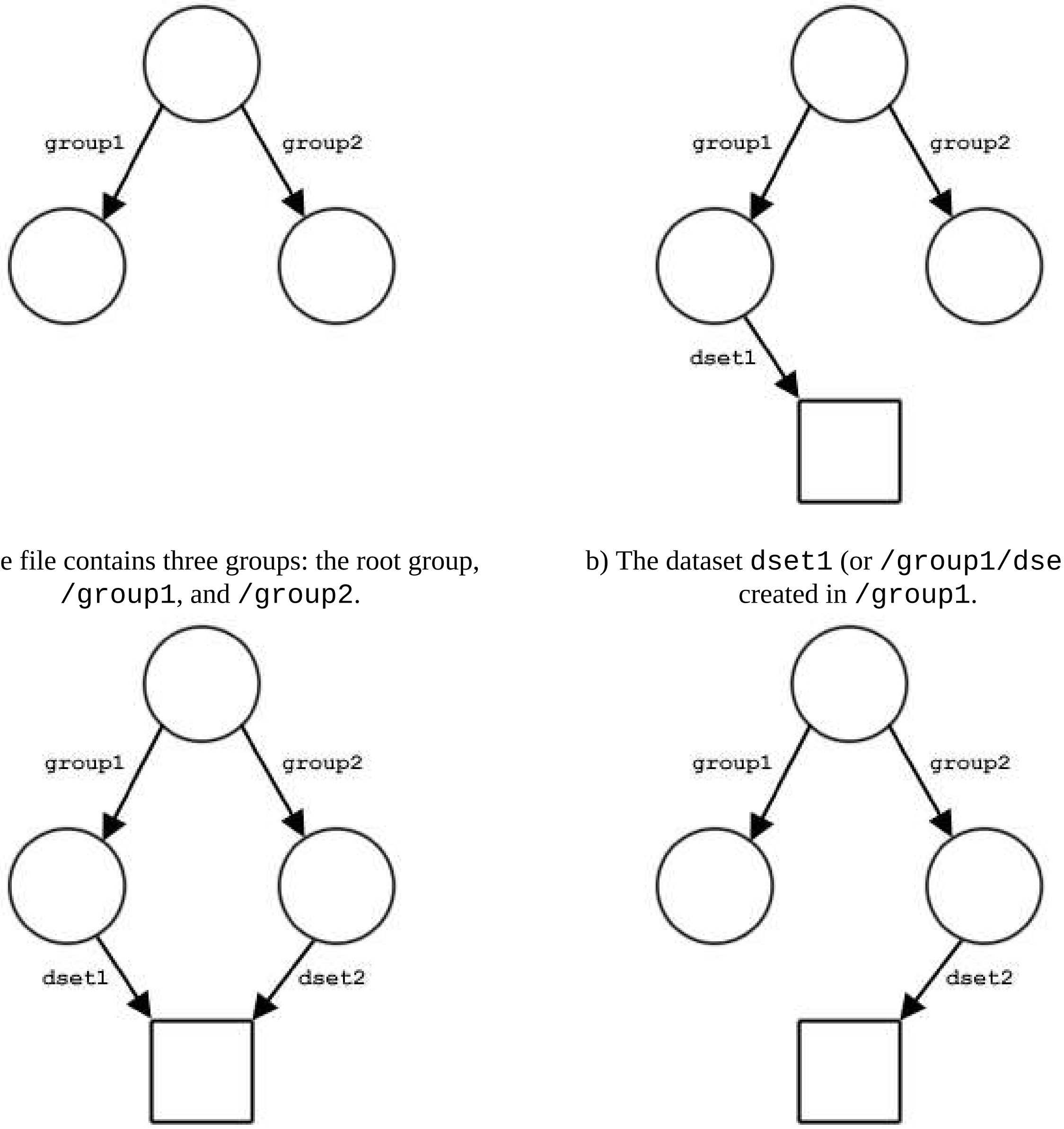
Figure 26

There is a second important case when the twice-visited member is a group. Any Group with more than one link to it can potentially be part of a circular path. I.e., recursively iterating through member groups may eventually bring the the iteration back to the current group and may generate an infinite path within the file’s linked structure. To embark upon the resulting infinite iteration would clearly be unacceptable in the general case. Figure 26 illustrates an HDF5 file with such potential.

In such a case, the callback function should check the reference count in the `H5G_stat_t` buffer as returned by *H5Gget_objinfo*. If the count is greater than one, there is more than one path to the object in question and it may be in a loop; the program should act accordingly. For example, it may be necessary to construct a global table of all the objects visited. Note that the object’s name is not unique, but the full path and the object number (found in the above-mentioned `H5G_stat_t` buffer) are unique within an individual HDF5 file.

9. Examples of File Structures

This section presents several samples of HDF5 file structures.



a) The file contains three groups: the root group, `/group1`, and `/group2`.

b) The dataset `dset1` (or `/group1/dset1`) is created in `/group1`.

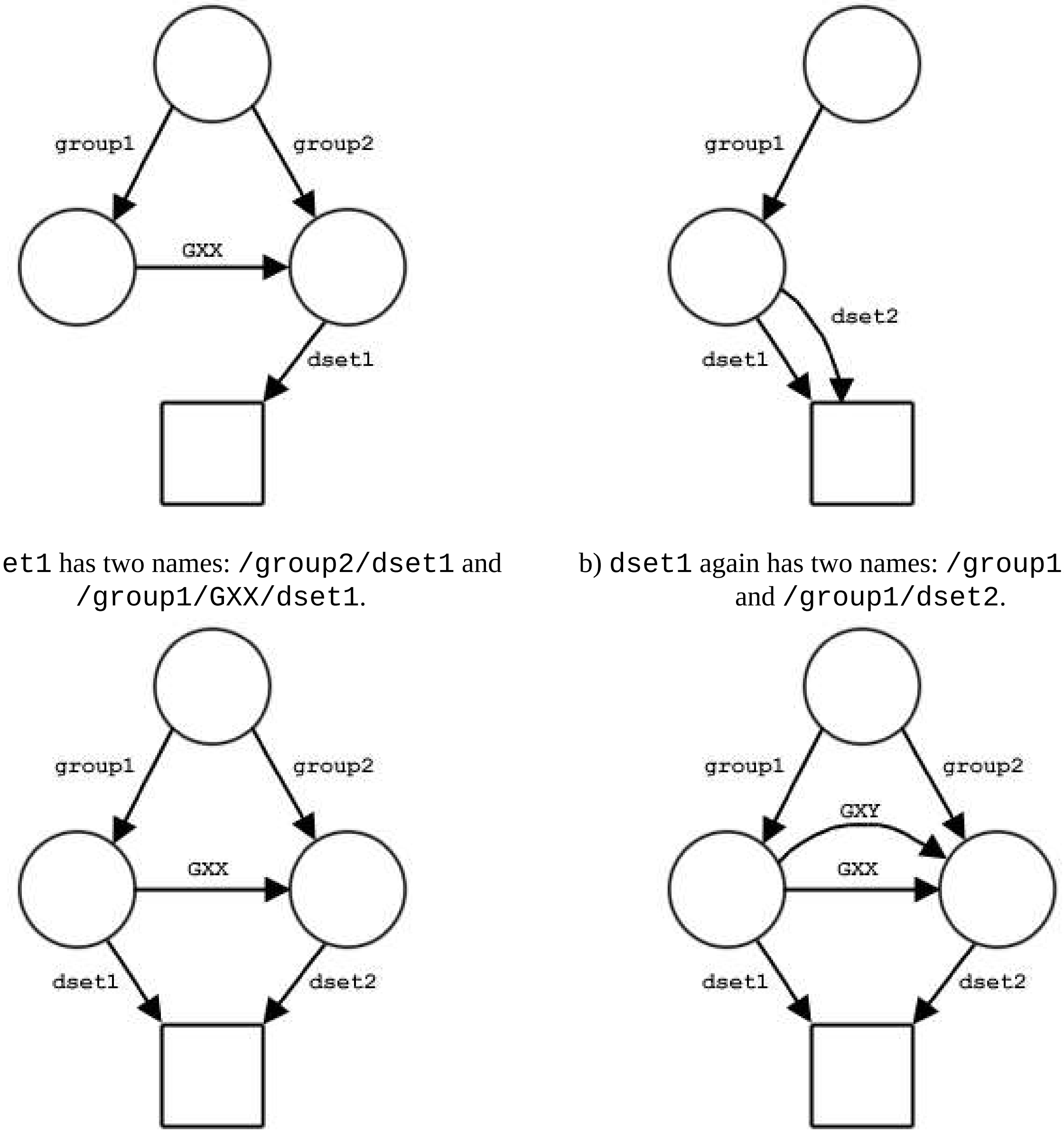
c) A link named `dset2` to the same dataset is created in `/group2`.

d) The link from `/group1` to `dset1` is removed. The dataset is still in the file, but can be accessed only as `/group2/dset2`.

Figure 27

Figure 27 shows examples of the structure of a file with three groups and one dataset. The file in Figure 27a contains three groups: the root group and two member groups. In Figure 27b, the dataset `dset1` has been created in `/group1`. In Figure 27c, a link named `dset2` from `/group2` to the dataset has been added. Note that there is only one copy of the dataset; there are two links to it and it can be accessed either as `/group1/dset1` or as `/group2/dset2`.

Figure 27d illustrates that one of the two links to the dataset can be deleted. In this case, the link from `/group1` has been removed. The dataset itself has not been deleted; it is still in the file but can only be accessed as `/group1/dset2`.



a) `dset1` has two names: `/group2/dset1` and `/group1/GXX/dset1`.

b) `dset1` again has two names: `/group1/dset1` and `/group1/dset2`.

c) `dset1` has three names: `/group1/dset1`, `/group2/dset2`, and `/group1/GXX/dset2`.

d) `dset1` has an infinite number of available path names.

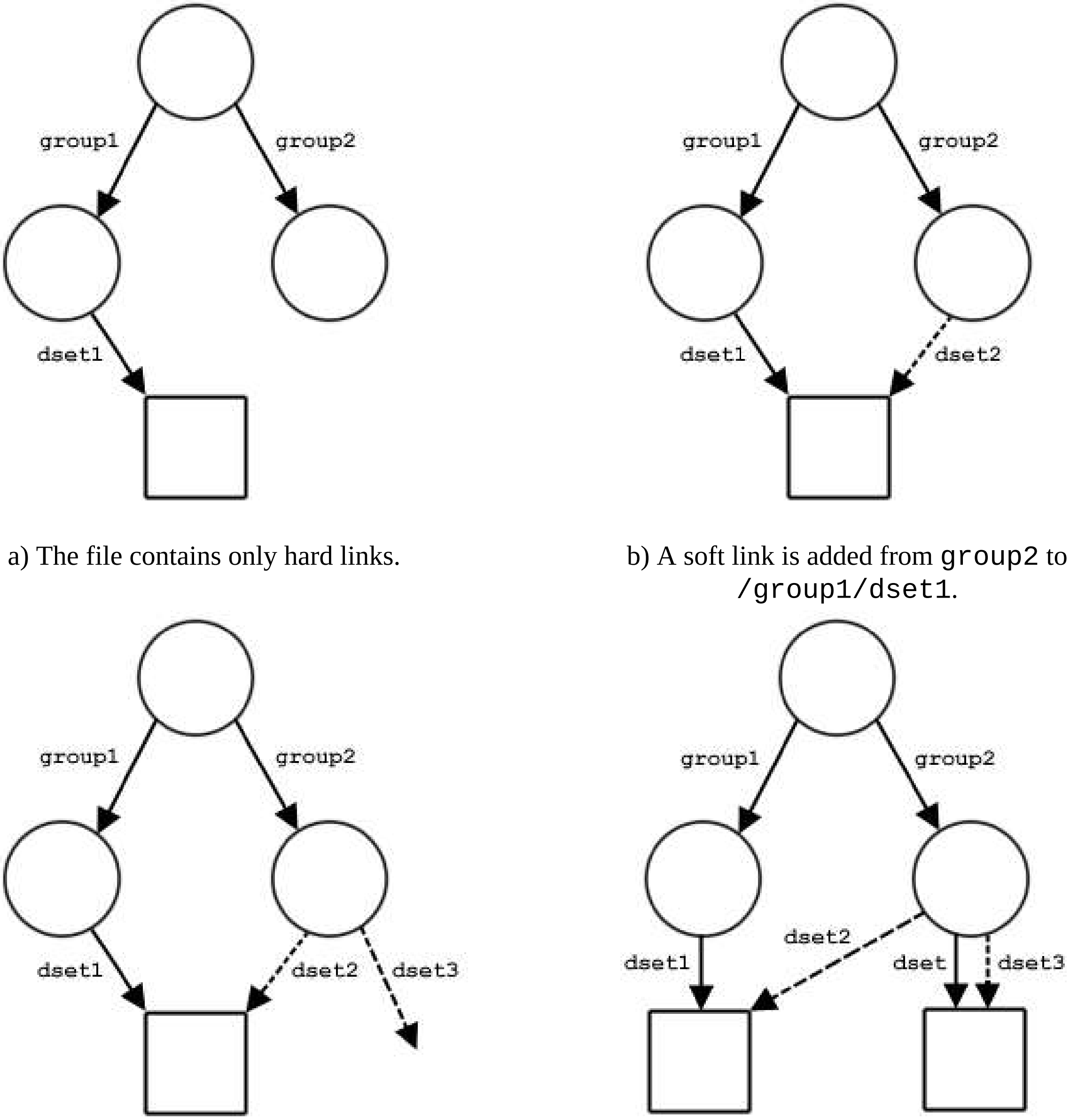
Figure 28

Figure 28 illustrates loops in an HDF5 file structure. The file in Figure 28a contains three groups and a dataset; `group2` is a member of the root group and of the root group's other member group, `group1`. `group2` thus can be accessed by either of two paths: `/group2` or `/group1/GXX`. Similarly, the dataset can be accessed either as `/group2/dset1` or as `/group1/GXX/dset1`.

Figure 28b illustrates a different case: the dataset is a member of a single group but with two links, or names, in that group. In this case, the dataset again has two names, `/group1/dset1` and `/group1/dset2`.

In Figure 28c, the dataset `dset1` is a member of two groups, one of which can be accessed by either of two names. The dataset thus has three path names: `/group1/dset1`, `/group2/dset2`, and `/group1/GXX/dset2`.

And in Figure 28d, two of the groups are members of each other and the dataset is a member of both groups. In this case, there are an infinite number of paths to the dataset because `GXX` and `GYY` can be traversed any number of times on the om the root group, `/`, way frto the dataset. This can yield a path name such as `/group1/GXX/GYY/GXX/GYY/GXX/dset2`.



a) The file contains only hard links.

b) A soft link is added from `group2` to `/group1/dset1`.

c) A soft link named `dset3` is added with a target that does not yet exist.

d) Tht target of soft link is created or linked.

Figure 29

Figure 29 takes us into the realm of soft links. The original file, in Figure 29a, contains only three hard links. In Figure 29b, a soft link named `dset2` from `group2` to `/group1/dset1` has been created, making this dataset accessible as `/group2/dset2`.

In Figure 29c, another soft link has been created in `group2`. But this time the soft link, `dset3`, points to a target object that does not yet exist. That target object, `dset`, has been added in Figure 29d and is now accessible as either `/group2/dset` or `/group2/dset3`.

[1] It could be said that HDF5 extends the organizing concepts of a file system to the internal structure of a single file.

[2] As of HDF5-1.4, the storage used for an object is reclaimed, even if all links are deleted.

Chapter 4
HDF5 Datasets

1. Introduction

An HDF5 dataset is an object composed of a collection of data elements, or raw data, and metadata that stores a description of the data elements, data layout, and all other information necessary to write, read, and interpret the stored data. From the viewpoint of the application the raw data is stored as a one-dimensional or multi-dimensional array of elements (the *raw data*), those elements can be any of several numerical or character types, small arrays, or even compound types similar to C structs. The Dataset object may have Attribute objects.

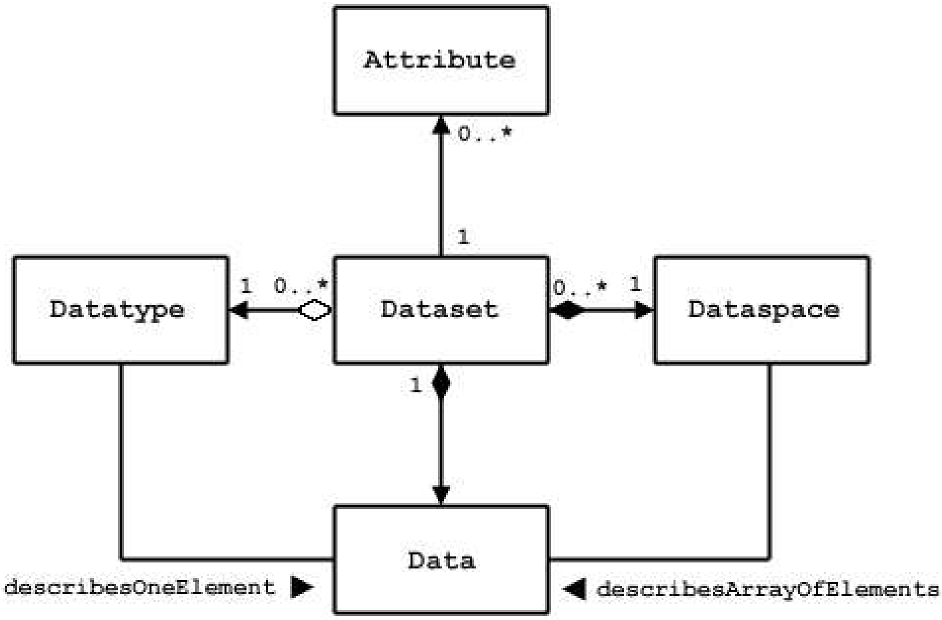


Figure 1

A Dataset objects 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 (the *storage layout*), and other information used by the library to speed up access to the dataset or maintain the file's integrity.

The HDF5 dataset interface, comprising the H5D functions, provides a mechanism for managing HDF5 datasets, including the transfer of data between memory and disk and the description of dataset properties.

A Dataset is used by other HDF5 APIs, either by name or by a handle (e.g., returned by H5Dopen).

Link/Unlink

A Dataset can be added to a Group with the H5Glink call, and deleted from a group with H5Gunlink. The link and unlink operations use the name of an object, which may be a dataset. The dataset does not have to open to be linked or unlinked.

Object reference

A Dataset may be the target of an object reference. The object reference is create by H5Rcreate, with the name of an object which may be a dataset and the reference type H5R_OBJECT. The Dataset does not have to be open to create a reference to it.

An object reference may also refer to a region (selection) of a Dataset. The reference is created with H5Rcreate and a reference type of H5R_DATASET_REGION.

An object reference can be accessed by a call to H5Rdereference. When the reference is to a Dataset or Dataset region, the H5Rdeference call returns a handle to the Dataset, just as if H5open has been called.

Adding attributes

A Dataset may have user defined Attributes, which are created with H5Acreate, and accessed through the H5A API. To create an attribute for a Dataset, the Dataset must be open, and the handle is passed to H5Acreate. The attributes of a Dataset are discovered, and opened using H5Aopen_name, H5Aopen_idx, or H5Aiterate; which use the handle of the Dataset. An Attribute can be deleted with H5Adelete, which uses the handle of the Dataset.

2. File Function Summaries

C Function F90 Function	Purpose
`H5Dcreate` `h5dcreate_f`	Creates a dataset at the specified location.
`H5Dopen` `h5dopen_f`	Opens an existing dataset.
`H5Dclose` `h5dclose_f`	Closes the specified dataset.
`H5Dget_space` `h5dget_space_f`	Returns an identifier for a copy of the dataspace for a dataset.
`H5Dget_space_status` `h5dget_space_status_f`	Determines whether space has been allocated for a dataset.
`H5Dget_type` `h5dget_type_f`	Returns an identifier for a copy of the datatype for a dataset.
`H5Dget_create_plist` `h5dget_create_plist_f`	Returns an identifier for a copy of the dataset creation property list for a dataset.
`H5Dget_offset` `h5dget_offset_f`	Returns dataset address in file.
`H5Dget_storage_size` `h5dget_storage_size_f`	Returns the amount of storage required for a dataset.
`H5Dvlen_get_buf_size` `h5dvlen_get_max_len_f`	Determines the number of bytes required to store VL data.

`H5Dvlen_reclaim`
`(none)`
	Reclaims VL datatype memory buffers.

`H5Dread` `h5dread_f`	Reads raw data from a dataset into a buffer.
`H5Dwrite` `h5dwrite_f`	Writes raw data from a buffer to a dataset.

`H5Diterate`
`(none)`
	Iterates over all selected elements in a dataspace.

`H5Dextend` `h5dextend_f`	Extends a dataset with unlimited dimension.
`H5Dfill` `h5dfill_f`	Fills dataspace elements with a fill value in a memory buffer.

3. Programming Model

This section explains the programming model for a Datasets.

3.1 General Model

The programming model for using a Dataset has three main phases:

obtain access to the dataset operate on the dataset using the Dataset handle returned above release the dataset. A Dataset may be opened several times, and operations performed with several different handles to the same Dataset. All the operations affect the dataset, although the calling program must synchronize if necessary to serialize accesses.

Note that the Dataset remains open until the last handle is closed. Figure 2 shows the basic sequence of operations.

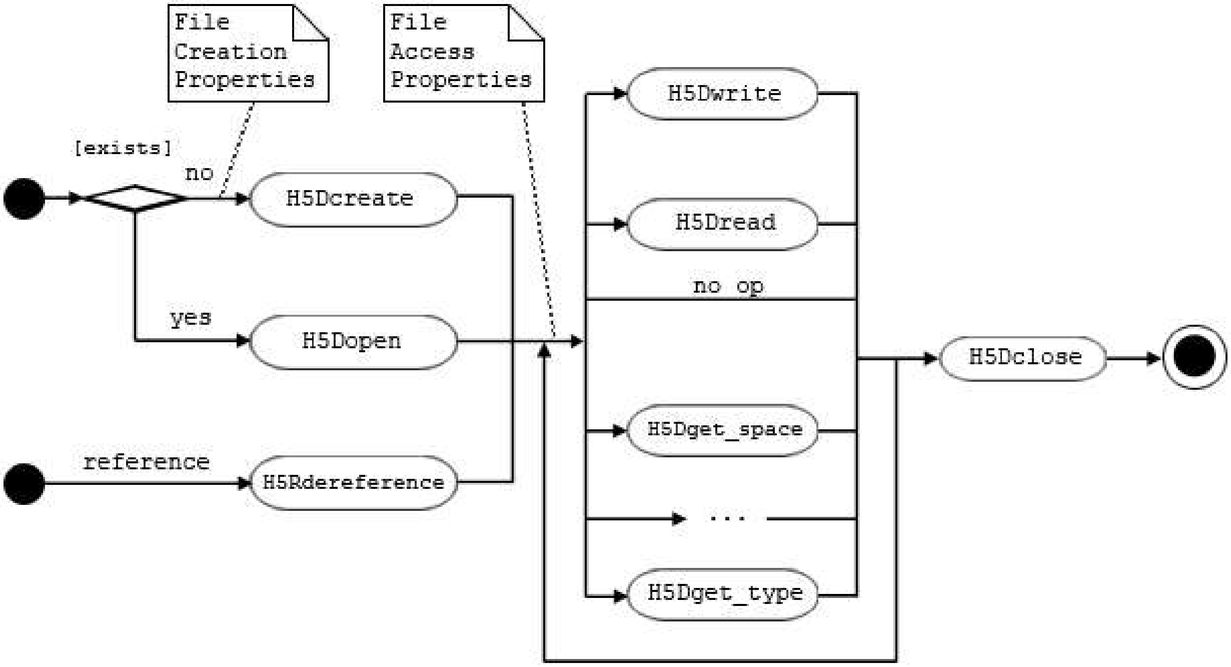


Figure 2

Creation and data access operations may have optional parameters which are set with property lists. The general programming model is:

1. create property list of appropriate class (dataset create, dataset transfer)
2. set properties as needed. Each type of property has its own format and datatype.
3. pass the property list as a parameter of the API call.

Step 1. Obtain Access

A new dataset is created by a call to H5Dcreate. If successful, the call returns a handle for the newly created Dataset.

Access to an existing Dataset is obtained by a call to H5Dopen. This call returns a handle for the existing Dataset.

An Object Reference may be dereferenced to obtain a handle to the dataset it points to.

In each of these cases, the successful call returns a handle to the dataset. The handle is used in subsequent operations until it is closed.

Step 2. Operate on the Dataset

The Dataset handle can be used to write and read data to the Dataset, to query and set properties, and to perform other operations such as adding attributes, linking in groups, creating references, and so on.

The Dataset handle can be used for any number of operations until it is closed.

Step 3. Close the Dataset

When all operations are completed, the Dataset handle should be closed. This releases the dataset.

After the handle is closed, it cannot be used for further operations.

3.2 Create Dataset

A Dataset is created and initialized with a call to H5Dcreate. The Dataset create operation sets permanent properties of the Dataset:

- name
- dataspace
- datatype
- storage properties

These properties cannot be changed for the life of the Dataset, although the dataspace may be expanded up to its maximum dimensions.

Name

A Dataset name is a sequence of alphanumeric ASCII characters. The full name would include a tracing of the group hierarchy from the root group of the file, e.g., /rootGroup/groupA/subgroup23/dataset1. The local name or relative name within the lowest-level group containing the dataset would include none of the group hierarchy. e.g., Dataset1.

Dataspace

The Dataspace of a dataset defines the number of dimensions and the size of each dimension. The Dataspace defines the number of dimensions, and the maximum dimension sizes and current size of each dimension. The maximum dimension size can be a fixed value or the constant H5D_UNLIMITED, in which case the actual dimension size can be incremented with calls to H5Dextend, up to the maximum. The maximum dimension size is set when the Dataset is created and cannot be changed.

Datatype

Raw data has a Datatype, which describes the layout of the raw data stored in the file. The file Datatype is set when the Dataset is created and can never be changed. When data is transferred to and from the Dataset, the HDF5 library will assure that the data is transformed to and from the stored format.

Storage Properties

Storage properties of the Dataset are set when it is created. Table 1 shows the categories of storage properties. The storage properties cannot be changed after the Dataset is created. The storage properties are described below.

Filters

When a Dataset is created, optional filters are specified. The filters are added to the data transfer pipeline when data is read or written. The standard library includes filters to implement compression, data shuffling, and error detection code. Additional user defined filters may also be used.

The required filters are stored as part of the Dataset, and the list may not be changed after the Dataset is created. The HDF5 library automatically applies the filters whenever data is transferred.

Summary

A newly created Dataset has no attributes and no data values. The dimensions, data type (in the file), storage properties, and selected filters are set. Table 1 lists the required inputs, Table 2 lists the optional inputs.

Table 1

Required inputs	Description
Dataspace	The shape of the array
Datatype	The layout of the stored elements
name	The name of the dataset in the group

Table 2

Optional Setting	Description
Storage Layout	How the data is organized in the file, including chunking.
Fill value	The behavior and value for uninitialized data.
External Storage (optional)	Option to store the raw data in an external file.
Folders	Select optional filters to be applied, e.g., compression.

Example

To create a new dataset

Set dataset characteristics. (Optional where default settings are acceptable)

Datatype
Dataspace
Dataset creation property list Create the dataset.
Close the datatype, dataspace, and property list. (As necessary)
Close the dataset.

Figure 3 shows example code to create an empty dataset. The Dataspace is 7 X 8, the Datatype is a Big Endian integer. The dataset is created with the name "dset1", it is a member of the root group, "/".

```
Hid_t    dataset, datatype, dataspace;

 /*
  * Create dataspace: Describe the size of the array and
  * create the data space for fixed size dataset.
 */
 dimsf[0] = 7;
 dimsf[1] = 8;
 dataspace = H5Screate_simple(2, dimsf, NULL);
 /*
  * Define datatype for the data in the file.
  * For this example, 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. No properties are set.
  */
 dataset = H5Dcreate(file, "/dset", datatype, dataspace, H5P_DEFAULT);

 H5Dclose(dataset);
 H5Sclose(dataspace);
 H5Tclose(datatype);
```

Figure 3

Figure 4 shows example code to create a similar Dataset with a fill value of '-1'. This code has the same steps as in Figure 3, but uses a non-default property list. A file creation property list is created, and then the fill value is set to the desired value. Then the property list is passed to the H5Dcreate call.

```
hid_t    dataset, datatype, dataspace;
 hid_t plist;  /* property list */
 int fillval = -1;
 dimsf[0] = 7;
 dimsf[1] = 8;
 dataspace = H5Screate_simple(2, dimsf, NULL);

 datatype = H5Tcopy(H5T_NATIVE_INT);
 status = H5Tset_order(datatype, H5T_ORDER_LE);

 /*
  * Example of Dataset Creation property list: set fill value to '-1'
  */
 plist = H5Pcreate((H5P_DATASET_CREATE);
 status = H5Pset_fill_value(plist,datatype, &fillval);

/* Same as above, but use the property list */
 dataset = H5Dcreate(file, "/dset", datatype, dataspace, plist);

 H5Dclose(dataset);
 H5Sclose(dataspace);
 H5Tclose(datatype);
 H5Pclose(plist);
```

Figure 4

After this code is executed, the dataset has been created and written to the file. The data array is uninitialized. Depending on the storage strategy and fill value options that have been selected, some or all of the space may be allocated in the file, and fill values may be written in the file.

3.3 Data Transfer Operations on a Dataset

Data is transferred between from memory and the raw data array of the Dataset through H5Dwrite and H5Dread operations. A data transfer has the following basic steps:

1. allocate and initialize memory space as needed
2. define the datatype of the memory elements
3. define the elements to be transferred (a selection, or all the elements)
4. set data transfer properties (including parameters for filters or File Drivers) as needed
5. call the H5D API

Note that the location of the data in the file, the Datatype of the data in the file, the storage properties, and the filters do not need to be specified, because these are stored as a permanent part of the Dataset. A selection of elements from the Dataspace is specified, which may be the whole dataspace.

Figure 5 shows a diagram of a write operation, which transfers a data array from memory to a dataset in the file (usually on disk). A read operation has similar parameters, with the data flowing the other direction.

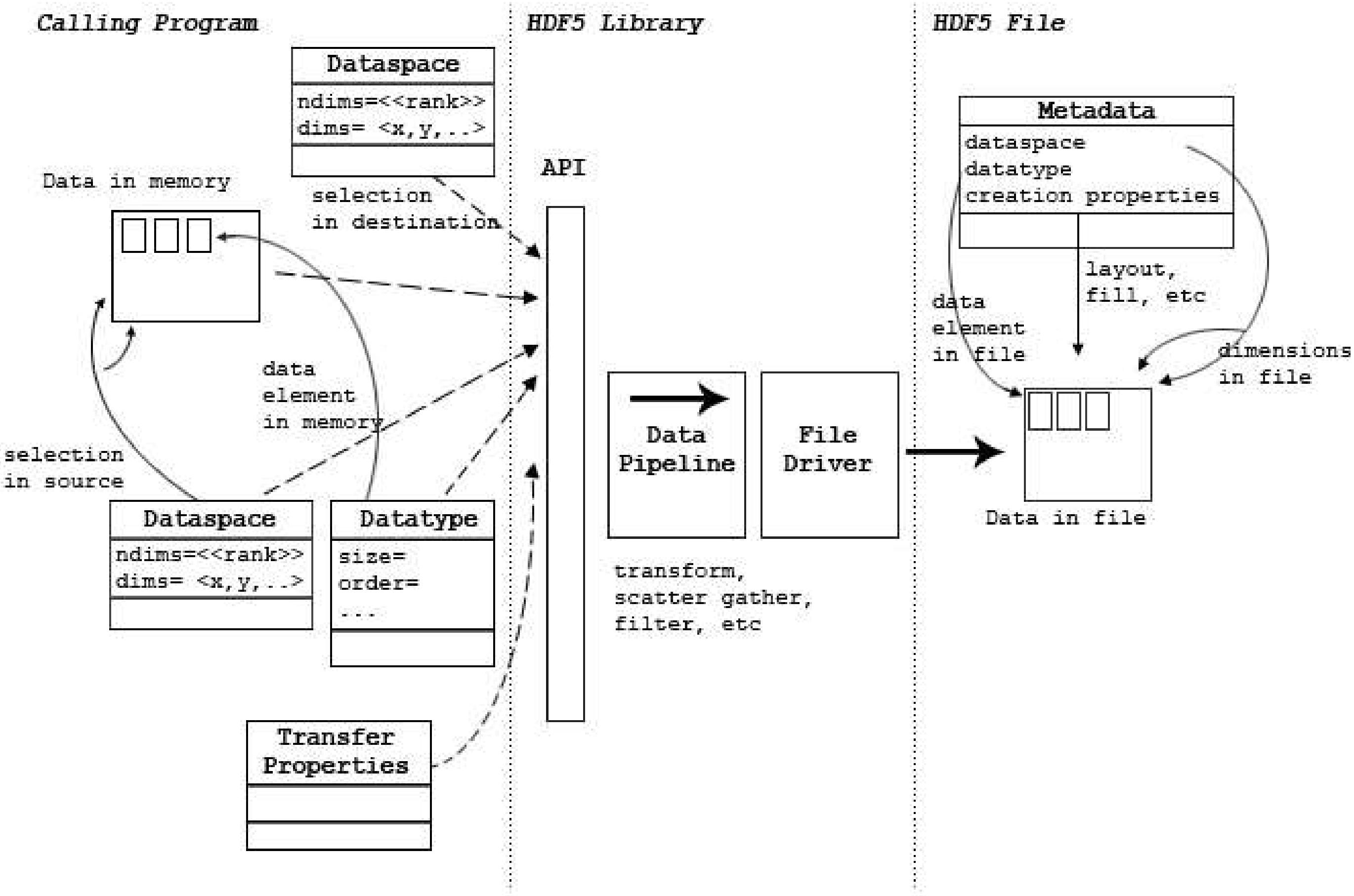


Figure 5

Memory Space

The calling program must allocate sufficient memory to store the data elements to be transferred. For a write (from memory to the file), the memory must be initialized with the data to be written to the file. For a read, the memory must be large enough to store the elements that will be read. The amount of storage needed can be computed from the memory Datatype (which defines the size of each Data element) and the number of elements in the selection.

Memory Datatype

The memory layout of a single data element is specified by the Memory Datatype. This specifies the size, alignment, and byte order of the element, as well as the Datatype Class. Note that the memory data type must be the same Datatype Class as the file, but may have different byte order and other properties. The HDF5 library automatically transforms data elements between the source and destination layouts. See the chapter "HDF5 Datatypes" for more details.

For a write, the memory Datatype defines the layout of the data to be written, e.g., IEEE floating point numbers in native byte order. If the file Datatype (defined when the Dataset is created) is different but compatible, the HDF5 library will transform each data element when it is written. For example, if the file byte order is different than the native byte order, the HDF5 library will swap the bytes.

For a read, the memory Datatype defines the desired layout of the data to be read. This must be compatible with the file Datatype, but should generally use native formats, e.g., byte orders. The HDF5 library will transform each data element as it is read.

Selection

The data transfer will transfer some or all of the elements of the Dataset, depending on the Dataspace selection. The selection is two Dataspace objects (one for the source, and one for the destination) which describe which elements of the Dataspace to be transferred, which may be all of the data, or just some elements (partial I/O). Partial I/O is defined by defining hyperslabs or lists of elements in a Dataspace object.

The Dataspace selection for the source defines the indices of the elements to be read, the Dataspace selection for the destination defines the indices of the elements to be written. The two selections must define the same number of points, but the order and layout may be different. The HDF5 library automatically selects and distributes the elements, according to the selections, e.g., to perform a scatter-gather or sub-set of the data.

Data Transfer Properties

For some data transfers, additional parameters should be set using the transfer property list. Table 2 lists the categories of transfer properties. These properties set parameters for the HDF5 library, and may be used to pass parameters for optional filters and file drivers. For example, transfer properties are used to select independent or collective operation when using MPI-I/O.

Table 3

Properties	Description
Library parameters	Internal caches, buffers, B-Trees, etc.
Memory management	Variable length memory management, data overwrite
File driver management	Parameters for file drivers
Filter management	Parameters for filters

Data Transfer Operation (read or write)

The data transfer is done by calling H5Dread or H5Dwrite with the parameters described above. The HDF5 library constructs the required pipe-line, which will scatter-gather, transform data types, apply the requested filters, and use the correct file driver.

During the data transfer, the transformations and filters are applied to each element of the data, in the required order, until all the data is transferred.

Summary

To perform a data transfer, it is necessary to allocate and initialize memory, describe the source and destination, set required and optional transfer properties, and call the H5D API.

Examples

The basic procedure to write to a dataset

Open the dataset.
Set dataset dataspace of write. (Optional if dataspace is H5S_SELECT_ALL)
Write data.
Close the datatype, dataspace, and property list. (As necessary)
Close the dataset.

Figure 6 shows example code to write a 4 X 6 array of integers. In the example, the data is initialized in the memory array dset_data. The dataset has already been created in the file, so it is opened with H5Dopen.

The Data is written with H5Dwrite. The arguments are the dataset handle, the memory datatype (H5T_NATIVE_INT), the memory and file selections (H5S_ALL in this case: the whole array), and the default (empty) property list. The last argument is the data to be transferred.

```
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("dset.h5", H5F_ACC_RDWR, H5P_DEFAULT);

/* Open an existing dataset. */
dataset_id = H5Dopen(file_id, "/dset");

/* Write the entire dataset, using 'dset_data':
      memory type is 'native int'
      write the entire dataspace to the entire dataspace,
      no transfer properties,
 */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL,
        H5S_ALL, H5P_DEFAULT, dset_data);


status = H5Dclose(dataset_id);
```

Figure 6

Figure 7 shows a similar write, setting a non-default value for the transfer buffer. The code is the same as Figure 6, but a transfer property list is created and the desired buffer size is set. The H5Dwrite has the same arguments, but uses the property list to set the buffer.

```
hid_t       file_id, dataset_id;
hid_t  xferplist;
herr_t      status;
int         i, j, dset_data[4][6];

file_id = H5Fopen("dset.h5", H5F_ACC_RDWR, H5P_DEFAULT);

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

/*
  * Example: set type conversion buffer to 64MB
  */
 xferplist = H5Pcreate(H5P_DTASET_XFER);
 status = H5Pset_buffer( xferplist, 64 * 1024 *1024, NULL, NULL);

/* Write the entire dataset, using 'dset_data':
      memory type is 'native int'
      write the entire dataspace to the entire dataspace,
      set the buffer size with the property list,
 */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL,
        H5S_ALL, plist, dset_data);


status = H5Dclose(dataset_id);
```

Figure 7

To read from a dataset

Define memory dataspace of read. (Optional if dataspace is H5S_SELECT_ALL)
Open the dataset.
Get the dataset dataspace. (If using H5S_SELECT_ALL above)

Else define dataset dataspace of read. Define the memory datatype. (Optional)
Define the memory buffer.
Open the dataset.
Read data.
Close the datatype, dataspace, and property list. (As necessary)
Close the dataset.

Figure 8 shows example code that reads a 4 X 6 array of integers from a dataset called "dset1". First, the dataset is opened. The H5Dread call has parameters:

- the dataset handle (from H5Dopen)
- The memory datatype (H5T_NATVE_INT)
- The memory and file dataspace (H5S_ALL, the whole array)
- A default (empty) property list
- The memory to be filled.

```
Hid_t       file_id, dataset_id;
herr_t      status;
int         i, j, dset_data[4][6];


/* Open an existing file. */
file_id = H5Fopen("dset.h5", H5F_ACC_RDWR, H5P_DEFAULT);

/* Open an existing dataset. */
dataset_id = H5Dopen(file_id, "/dset");

/* read the entire dataset, into 'dset_data':
      memory type is 'native int'
      read the entire dataspace to the entire dataspace,
      no transfer properties,
 */
status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL,
        H5S_ALL, H5P_DEFAULT, dset_data);

status = H5Dclose(dataset_id);
```

Figure 8

3.4 Retrieve properties of a Dataset

The Dataset can be queried to discover its properties (Table 4). The Dataspace and Datatype, and Dataset creation properties can be retrieved. These calls return a handle to an object.

The total size of the data can be retrieved. This is the amount of stored data for the raw data or for variable length data. This is the amount of memory that must be allocated to read the whole array.

Query Function	Description
H5Dget_space	Retrieve the file dataspace
H5Dget_type	Retrieve the file datatype
H5Dget_create_plist	Retrieve the creation properties
H5Dget_storage_size	Retrieve the total bytes for all the data of the dataset
H5Dvlen_get_buf_size	Retrieve the total bytes for all the variable length data of the dataset

Example

```
hid_t       file_id, dataset_id;
hid_t    dspace_id, dtype_id, plist_id;
herr_t      status;

/* Open an existing file. */
file_id = H5Fopen("dset.h5", H5F_ACC_RDWR, H5P_DEFAULT);

/* Open an existing dataset. */
dataset_id = H5Dopen(file_id, "/dset");

dspace_id = H5Dget_space(dataset_id);
dtype_id = H5Dget_type(dataset_id);
plist_id = H5Dget_create_plist(dataset_id);

/* use the objects to discover the properties of the dataset */

status = H5Dclose(dataset_id);
```

3.5 Other Operations

The Dataset is used for other miscellaneous operations.

Table 4

Operation	Description
H5Dextend	See below
H5Diterate	
H5Dvlen_reclaim	See below

4. Data Transfer: Raw Data I/O

The HDF5 library implements data transfers through a pipeline which implements data transformations (according to the Datatype and selections), chunking (as requested), and I/O operations using different mechanisms (file drivers). The pipeline is automatically configured by the HDF5 library. Metadata is stored in the file so that the correct pipeline can be constructed to retrieve the data. In addition, optional filters, such as compression, may be added to the standard pipeline.

Figure 9 illustrates data layouts for different layers of an application using HDF5. The application data is organized as a multidimensional array of elements. The HDF5 format specification defines the stored layout of the data and metadata. The storage layout properties define the organization of the abstract data. This data is written and read to and from some storage medium.

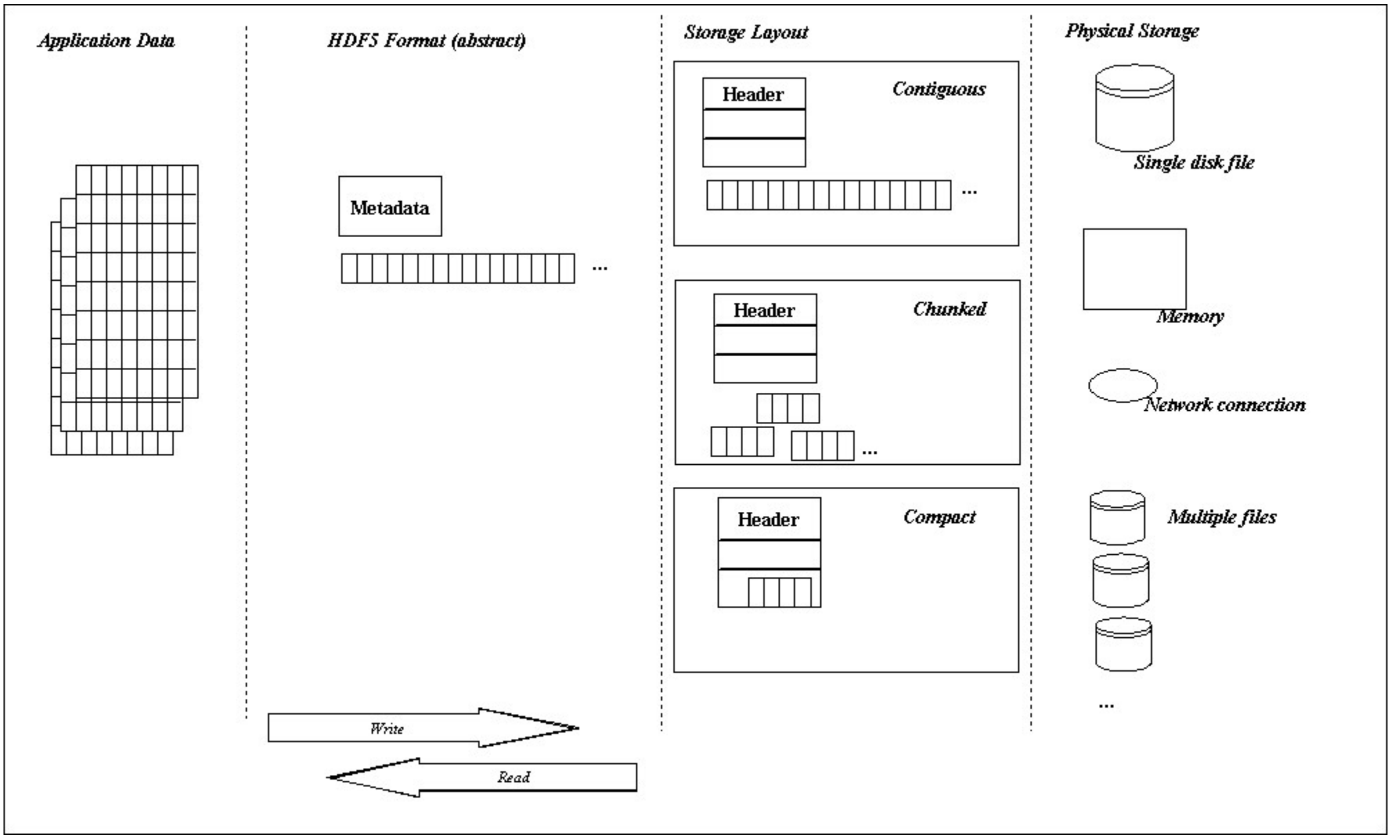


Figure 9

The last stage of a write (and first stage of a read) is managed by an HDF5 File Driver module. The Virtual File Layer of the HDF5 library implements a standard interface to alternative I/O methods, including memory (AKA "core") files, single serial file I/O, multiple file I/O, and parallel I/O. The File Driver maps a simple abstract HDF5 file to the specific access methods.

The raw data of an HDF5 Dataset is conceived to be a multi-dimensional array of data elements. This array may be stored in the file according to several storage strategies:

- COMPACT
- CONTIGUOUS
- CHUNKED

The storage strategy does not affect data access methods, except that certain operations may be more or less efficient depending on the storage strategy and the access patterns.

Overall, the data transfer operations (H5Dread and H5Dwrite) work identically for any storage method, for any file driver, and for any filters and transformations. The HDF5 library automatically manages the data transfer process. In some cases, transfer properties should or must be used to pass additional parameters, such as MPI/IO directives when used the parallel file driver.

4.1 Data pipeline

When data is written or read to or from and HDF5 file, the HDF5 library passes the data through a sequence of processing steps, the HDF5 data pipeline. This data pipeline performs operations on the data in memory, including byte swapping, alignment, scatter-gather, and hyperslab selections. The HDF5 library automatically determines which operations are needed and manages the organization of memory operations, such as extracting selected elements from a data block. The data pipeline modules operate on data buffers, each processes the buffer and passes the transformed buffer to the next stage.

Table 5 lists the stages of the data pipeline Figure 10 shows the order of processing during a read or write.

Table 5

Layers	Description
I/O initiation	Initiation of HDF5 I/O activities in user's application program, i.e. H5Dwrite and H5Dread.
Memory hyperslab operation	Data is scattered to (for read), or gathered from (for write) application's memory buffer (bypassed if no datatype conversion is needed).
Datatype conversion	Datatype is converted if it is different between memory and storage (bypassed if no datatype conversion is needed).
File hyperslab operation	Data is gathered from (for read), or scattered to (for write) to file space in memory (bypassed if no datatype conversion is needed).
Filter pipeline	Data is processed by filters when it passes. Data can be modified and restored here (bypassed if no datatype conversion is needed, no filter is enabled, or dataset is not chunked).
Virtual File Layer	Facilitate easy plug-in file drivers, like MPIO, POSIX I/O.
Actual I/O	Actual file driver used by the library, like MPIO or STDIO.

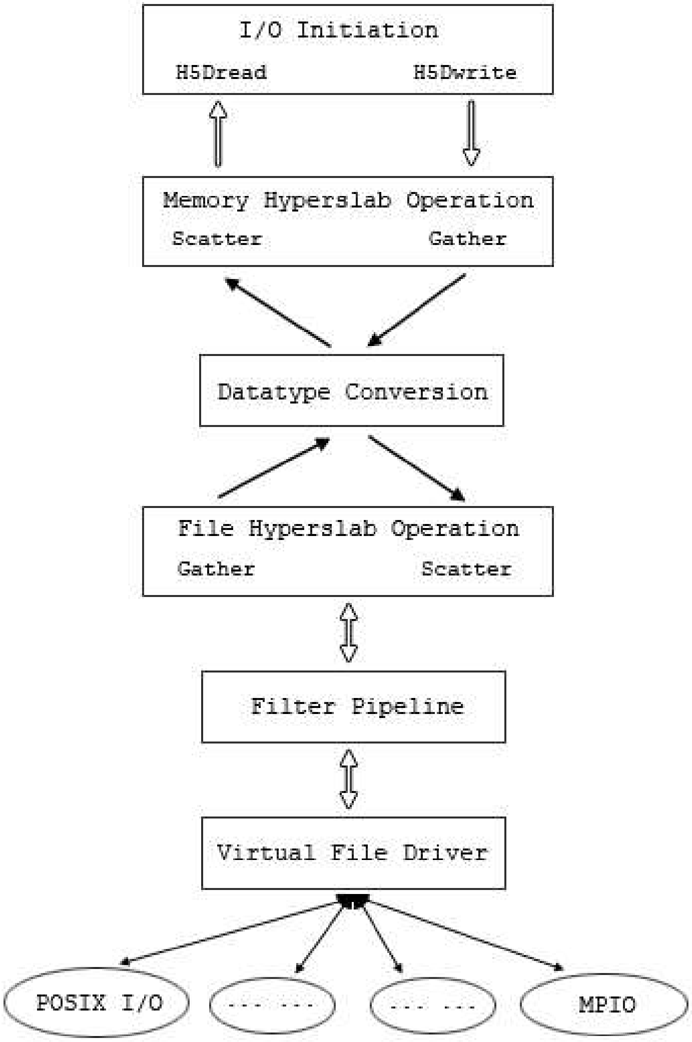


Figure 10

The HDF5 Library automatically applies the stages as needed.

When the memory Dataspace selection is other than the whole Dataspace, the Memory Hyperslab stage scatters/gathers the data elements between the appliation memory (described by the selection) and contiguous memory buffer for the pipeline. On a write, this is a gather operation, on a read, this is a scatter operation.

When the Memory Datatype is different from the File Datatype, the Datatype Conversion stage transforms each data element. For example, if data is written from 32-bit Big Endian memory, and the File Datatype is 32-bit Little Endian, the Datatype Conversion stage will swap the bytes of every elements. Similarly, when data is read from the file to native memory, byte swapping will be applied automatically when needed.

The File Hyperslab stage is similar to the Memory Hyperslab stage, but is managing the arrangement of the elements according to the File Dataspace selection. When data is read, data elements are gathered from the data blocks from the file to fill the contiguous buffers, which are processed by the pipeline. When data is read, the elements from a buffer are scattered to the data blocks of the file.

4.2 Filters

In addition to the standard pipeline, optional stages, called filters, can be inserted in the pipeline. The standard distribution includes optional filters to implement compression and error checking. User applications may add custom filters as well.

The HDF5 Library distribution includes or employs several optional filters, as listed in Table 6. The filters are applied in the pipeline between the Virtual File Layer and the File Hyperslab Operation (Figure 10). The application can use any number of filters, in any order.

Table 6

Filters	Description
gzip compression	Data compression using `zlib`.
Szip compression	Data compression using the Szip library.
Shuffling	To improve compression performance, data is regrouped by its byte position in the data unit. I.e. 1st, 2nd, 3rd, 4th bytes of integers are stored together respectively.
Fletcher32	Fletcher32 checksum for error-detection.

Filters may be used only for chunked data and are applied to chunks of data between the file hyperslab stage and the virtual file layer. At this stage in the pipeline, the data is organized as fixed-size blocks of elements, and the filter stage processes each chunk separately.

Filters are selected by dataset creation properties, and some behavior may be controlled by data transfer properties. The library determines what filters must be applied and in what order.

See The HDF Group website for further information regarding the Szip filter.

4.3 File drivers

I/O is performed by the HDF5 Virtual File layer. The File Driver interface writes and reads blocks of data, each driver module implements the interface using different I/O mechanisms. Table 7 lists the File Drivers currently supported. Note that the I/O mechanisms are separated from the pipeline processing: the pipeline and filter operations are identical no matter what data access mechanism is used.

Table 7

File Driver	Description
H5FD_CORE	Store in memory (optional backing store to disk file)
H5FD_DPSS	
H5FD_FAMILY	Store in a set of files
H5FD_GASS	Store using Globus Access to Secondary Storage
H5FD_LOG	Store in logging file.
H5FD_MPIO	Store using MPI/IO
H5FD_MULTI	Store in multiple files, several options to control layout.
H5FD_SEC2	Serial I/O to file using Unix "section 2" functions.
H5FD_STDIO	Serial I/O to file using Unix "stdio" functions
H5FD_STREAM	I/O to socket.

Each File Driver writes/reads contiguous blocks of bytes from a logically contiguous address space. The File Driver is responsible for managing the details of the different physical storage methods.

In general, everything above the Virtual File Layer works identically no matter what storage method is used. However, some combinations of storage strategies and file drivers are not allowed. Also, some options may have substantially different performance depending on the file driver that is used. In particular, multi-file and parallel I/O may perform considerably differently from serial drivers, depending on chunking and other settings.

4.4 Data Transfer Properties to manage the pipeline

Data Transfer properties set optional parameters that control parts of the data pipeline. Table 8 lists three transfer properties that control the behavior of the library.

Table 8

Property	Description
H5Pset_buffer	Maximum size for the type conversion buffer and background buffer and optionally supplies pointers to application-allocated buffers
H5Pset_hyper_cache	Whether to cache hyperslab blocks during I/O.
H5Pset_btree_ratios	Set 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.

Some filters and File Drivers require or use additional parameters from the application program. These can be passed in the transfer property list. Table 9 lists the four File Driver property lists.

Table 9

Property	Description
H5Pset_dxpl_mpio	Control the MPI I/O transfer mode (independent or collective) during data I/O operations.
H5Pset_dxpl_multi	
H5Pset_small_data_block_size	Reserves blocks of size bytes for the contiguous storage of the raw data portion of small datasets. The HDF5 library then writes the raw data from small datasets to this reserved space, thus reducing unnecessary discontinuities within blocks of meta data and improving IO performance.
H5Pset_edc_check	Disable/enable EDC checking for read. (When selected, EDC is always written)

The Transfer Properties are set in a property list, which is passed as a parameter of the H5Dread or H5Dwrite call. The transfer properties are passed to each pipeline stage, which may use or ignore any property in the list. In short, there is one property list, which contains all the properties.

4.5 Storage strategies

The raw data is conceptually a multi-dimensional array of elements, stored as a contiguous array of bytes. The data may be physically stored in the file in several ways. Table 6 lists the storage strategies for a dataset.

Table 10

Storage Strategy	Description
CONTIGUOUS	The dataset is stored as one continuous array of bytes
CHUNKED	The dataset is stored as fixed sized chunks.
COMPACT	A small dataset is stored in the metadata header.

The different storage strategies do not affect the data transfer operations of the dataset: reads and writes work the same for any storage strategy.

These strategies are described in the following sections.

Contiguous

A contiguous dataset is stored in the file as a header and a single continuous array of bytes. (Figure 12) The data elements are arranged in row major order, with according to the datatype. By default, data is stored contiguously.

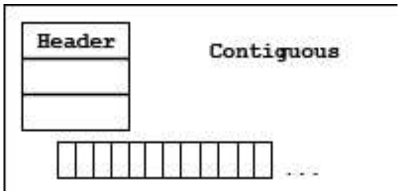


Figure 12

Contiguous storage is the simplest model. It has several limitations. First, the dataset must be a fixed size: it is not possible to extend the limit of dataset, or to have unlimited dimensions. Second, because data is passed through the pipeline as fixed sized blocks, compression and other filters cannot be used with contiguous data.

Chunked

The data of a dataset may be stored as fixed sized chunks (Figure 13). A chunk is a hyper-rectangle of any shape (less than or equal to the rank of the dataset). When a dataset is chunked, each chunk is read or written as a single I/O operation, and passed from stage to stage of the pipeline and filters.

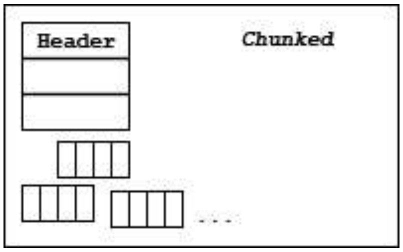


Figure 13

Chunks may be any size and shape that fits in the Dataspace of the dataset. For example, a three dimensional Dataspace can be chunked as 3-D cubes, 2-D planes, or 1-D lines. The chunks may extend beyond the size of the Dataspace, for example a 3 X 3 dataset might by chunked in 2 X 2 chunks. Sufficient chunks will be allocated to store the array, any extra space will not be accessible. So, to store the 3X3 array, four 2X2 chunks would be allocated, with 5 unused elements stored.

Chunked datasets can be unlimited (in any direction), and can be compressed or filtered.

Since the data is read or written by chunks, chunking can have a dramatic effect on performance by optimizing what is read and written. Note, too, that for specific access patterns (e.g., parallel I./O) decomposition into chunks can have a large impact on performance.

Compact

For Contiguous and Chunked storage, the Dataset header information and data are stored in two (or more) blocks (Figure 14). Therefore, at least two I/O operations are required to access the data, one to access the header, and one (or more) to access data. For a small dataset, this is considerable overhead.

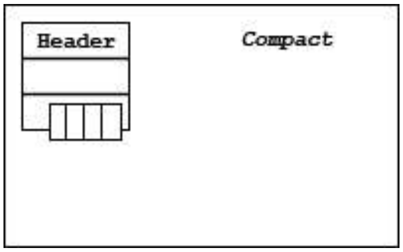


Figure 14

A small dataset may be stored in a continuous array of bytes in the header block, using the COMPACT storage option. This dataset can be read entirely in one operation, which retrieves the header and data. The dataset must fit in the header, which varies depending on what metadata may be stored. In general, a compact dataset should be approximately 30 KB or less total size.

4.6 Partial I/O-Subseting and Hyperslabs

Data transfers can write or read some of the data elements of the dataset. This is controlled by specifying two selections, one for the source and one for the destination. Selections are specified by creating a Dataspace with selections.

Selections may be a union of hyperslabs, where a hyperslab is a contiguous hyper-rectangle from the Dataspace. A second form of selection is a list of points. Selected fields of compound data type may be read or written. In this case, the selection is controlled by the memory and file Datatypes.

Summary of Procedure:

1. open the Dataset
2. define the memory Datatype
3. define the memory Dataspace selection and file Dataspace selection
4. transfer data (H5Dread or H5Dwrite)

For a detailed explanation of selections, see the chapter "HDF5 Dataspaces and Partial I/O."

5. Allocation of Space in the File

When a dataset is created, space is allocated in the file for its header and initial data. The amount of space allocated when the dataset is created depends on the storage properties. When the Dataset is modified (data is written, attributes added, or other changes), additional storage may be allocated if necessary.

Table 11

Object	Size (bytes)
Header	Variable, but typically around 256 bytes at the creation of a simple dataset with a simple datatype
Data	Size of the data array (number of elements X size of element). Space allocated in the file depends on storage strategy and allocation strategy.

Header

A dataset header consists of one or more header messages containing persistent metadata describing various aspects of the dataset. These records are defined in the *HDF5 File Format Specification*. The amount of storage required for the metadata depends on the metadata to be stored. Table 12 summarizes the metadata.

Table 12

Header Information	Approximate Storage Size
Datatype (required)	bytes or more, depends on type
Dataspace (required)	bytes or more, depends on number of dimensions and hsize_t
Layout (required) - points to the stored data	bytes or more, depends on hsize_t and number of dimensions
Filters	Depends on the number of filters, size of filter message depends on name and data that will be passed.

The header blocks also store the name and values of attributes, so the total storage depends on the number and size of the attributes.

In addition, the data set must have at least one link, including a name, which is stored in the file and in the Group it is a linked from.

The different storage strategies determine when and how much space is allocated for the data array. See the discussion of fill values below for a detailed explanation of the storage allocation.

Contiguous Storage

For a continuous storage option, the data is stored in a single, contiguous block in the file. The data is nominally a fixed size, (number of elements X size of element). Figure 15 shows and example of a two dimensional array, stored as a contiguous dataset.

Depending on the fill value properties, the space may be allocated when the dataset is created or when first written (default), and filled with fill values if specified. For parallel I/O, by default the space is allocated when the dataset if created.

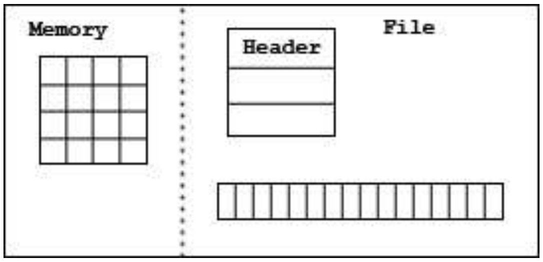


Figure 15

Chunked

For chunked storage, the data is stored in one or more chunks. Each chunk is a continuous block in the file, but chunks are not necessarily stored contiguously. Each chunk has the same size. The data array has the same nominal size as a contiguous array (number of elements X size of element), but the storage is allocated in chunks, so the total size in the file can be larger that the nominal size of the array.

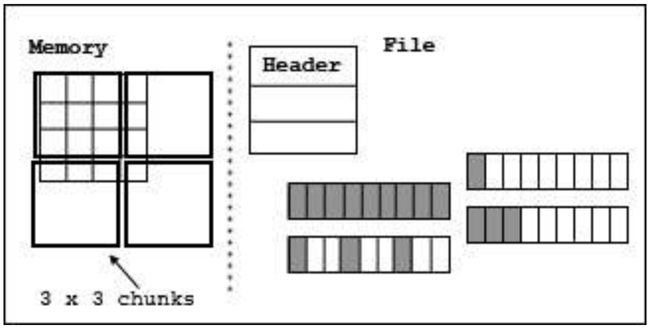


Figure 16

If a fill value is defined, each chunk will be filled with the fill value. Chunks must be allocated when data is written, but they may be allocated when the file is created, as the file expands, or when data is written.

For serial I/O, by default chunks are allocated incrementally, as data is written to the chunk. For a sparse dataset, chunks are allocated only for the parts of the dataset that are written. In this case, if the dataset is extended, no storage is allocated.

For parallel I/O, by default the chunks are allocate when the dataset is created or extended., with fill values written to the chunk.

In either case, the default can be changed using fill value properties. For example, using serial I/O, the properties can select to can allocate chunks when the dataset is created

H5Dextend is used to change the current dimensions of the Dataset, within the limits of the Dataspace. Each dimension can be extended up to its maximum, or unlimited. Extending the Dataspace may or may not allocate space in the file, and may or may not write fill values, if they are defined. See the next section for an explanation.

```
hid_t       file_id, dataset_id;
Herr_t      status;
size_t        newdims[2];


/* Open an existing file. */
file_id = H5Fopen("dset.h5", H5F_ACC_RDWR, H5P_DEFAULT);

/* Open an existing dataset. */
```

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

/* Example:  dataset is 2 X 3, each dimension is UNLIMITED */
/* extend to 2 X 7 */
newdims[0] = 2;
newdims[1] = 7;

status = H5Dextend(dataset_id, newdims);

/* dataset is now 2 X 7 */

status = H5Dclose(dataset_id);
```

Figure 17

5.1 Storage Allocation in the File: Early, Incremental, Late

The HDF5 Library implements several strategies for when storage is allocated if and when it is filled with fill values for elements not yet written by the user. Different strategies are recommended for different storage layouts and file drivers. In particular, a parallel program needs storage allocated during a collective call (e.g., create or extend), while serial programs may benefit from delaying the allocation until the data is written.

Two File Creation properties control "when to allocate space", "when to write the fill-value" and the actual fill-value to write.

Table 13 shows the three options for when data is allocated in the file. "Early" allocation is done during dataset create call. Certain File Drivers (especially MPI-I/O and MPI-posix) require space to be allocated when a dataset is created, so all processors will have the correct view of the data.

Table 13

Strategy	Description
Early	Allocate storage for the dataset immediately when the dataset is created.
Late	Defer allocating space for storing the dataset until the dataset is written.
Incremental	Defer allocating space for storing each chunk until the chunk is written.
Default	Use the strategy (Early, Late, Incremental) for the storage method and access method. (Recommended)

"Late" allocation is done at the time of the first write to dataset. Space for the whole Dataset is allocated at the first write.

"Incremental" allocation (chunks only) is done at the time of the first write to the chunk. Chunks that have never been written are not allocated in the file. In a sparsely populated Dataset, this option allocates chunks only where data is actually written.

The "Default" property selects the option recommended as appropriate for the storage method and access method. The defaults are shown in Table 14. Note that "Early" allocation is recommended for all Parallel I/O, while other options are recommended as the default for serial I/O cases.

Table 14

	Serial I/O	Parallel I/O
Contiguous Storage	Late	Early
Chunked Storage	Incremental	Early
Compact Storage	Early	Early

The second property is when to write fill value, "Never" and "Allocation". Table 15 shows these options.

Table 15

When	Description
Never	Fill value will never be written.
Allocation	Fill value is written when space is allocated. (Default for chunked and contiguous data storage.)

The third property is the fill value to write. Table 16 shows the values. By default, the data is filled with zeroes. The application may choose no fill value (Undefined), in which case uninitialized data may have any random values. The application may define a fill-value of the appropriate type. See the chapter "HDF5 Datatypes" for more information regarding fill values.

What to Write	Description
Undefined	No value stored, do not fill with zeroes (the default)
Default	By default, the library defines a fill-value of all zero bytes
User-defined	The applications specifies the fill value.

Together these three properties control the library's behavior. Table 16 summarizes the possible behavior of during the dataset create-write-close cycle.

Table 16

When to allocate space	When to write fill value	What fill value to write	Library create-write-close behavior
Early	Never	-	Library allocates space when dataset is created, but never writes fill value to dataset. (Read of unwritten data returns undefined values.)
Late	Never	-	Library allocates space when dataset is written to, but never writes fill value to dataset. (Read of unwritten data returns undefined values.)
Incremental	Never	-	Library allocates space when dataset or chunk (whichever is smallest unit of space) is written to, but never writes fill value to dataset or chunk. (Read of unwritten data returns undefined values.)
-	Allocation	undefined	**Error** on creating dataset, dataset not created.
Early	Allocation	default or user-defined	Allocate space for dataset when dataset is created. Write fill value (default or user-defined) to entire dataset when dataset is created.
Late	Allocation	default or user-defined	Allocate space for dataset when application first writes data values to the dataset. Write fill value to entire dataset before writing application data value.
Incremental	Allocation	default or user-defined	Allocate space for dataset when application first writes data values to the dataset or chunk (whichever is smallest unit of space). Write fill value to entire dataset or chunk before writing user's data value.

During the H5Dread function call, the library behavior depends on whether space has been allocated, whether fill value has been written to storage, how fill value is defined, and when to write fill value. Table 17 summarizes the different behaviors.

Table 17

Is space allocated in the file?	What is the fill value?	When to write fill value?	Library read behavior
No	undefined	<<any>>	**Error**. Cannot create this dataset.
	default or user-defined	<<any>>	Fill memory buffer with the fill value.
Yes	undefined	<<any>>	Return data from storage (dataset), trash is possible if user has not written data to portion of dataset being read.
	default or user-defined	Never	Return data from storage (dataset), trash is possible if user has not written data to portion of dataset being read.
	default or user-defined	Allocation	Return data from storage (dataset).

There are two cases to consider, depending on whether the space in the file has been allocated before the read or not. When space has not yet been allocated, if a fill value is defined the memory buffer will be filled with the values and returned (no read from disk).

If the space has been allocated, the values are returned from the stored data. The unwritten elements will be filled according to the fill value, or undefined.

5.2 Deleting a dataset from a file, reclaiming space

The size of the dataset cannot be reduced after it is created. The dataset can be expanded by extending one or more dimensions, with H5Dextend. It is not possible to contract a Dataspace, or to reclaim allocated space.

HDF5 does not at this time provide a mechanism to remove a Dataset from a file, or to reclaim the storage from deleted objects. Through the H5Gunlink function one can remove links to a dataset from the file structure. Once all links to a dataset have been removed, that dataset becomes inaccessible to any application and is effectively removed from the file. But this does not recover the space the dataset occupies.

The only way to recover the space is to write all the objects of the file into a new file. Any unlinked object is inaccessible to the application and will not be included in the new file.

See the chapter "HDF5 Groups" for further discussion of HDF5 file structures and the use of links.

5.3 Releasing memory resources (handles) when no longer needed

The system resources required for HDF5 objects, including Datasets, Datatypes, and Dataspaces, should be released once access to the object is no longer needed. This is accomplished via the appropriate close function. This is not particular to datasets but a general requirement when working with the HDF5 library; failure to close objects will result in resource leaks.

In the case where a Dataset is created or data has been transferred, there are several objects that must be closed, including the Dataset, the Datatype, Dataspace, and property lists.

The application program must free any memory variables and buffers it allocates. When accessing data from the file, the amount of memory required can be determined by determining the size of the memory Datatype and the number of elements in the memory selection.

Variable length data are organized in two or more areas of memory (see "HDF5 Datatypes"). When writing data, the application creates an array of vl_info_t, which contains pointers to the elements, e.g., strings. In the file, the Variable Length data is stored in two parts: a heap with the variable length values of the data elements, and an array vlinfo_t elements. When the data is read, the amount of memory required for the heap can be determined with the H5Dget_vlen_buf_size call.

The a Data Transfer property may be used to set a custom memory manager for allocating variable length data for a H5Dread. This is set with the H5Pset_vlen_mem_manager call.

To free the memory for Variable Length data, it is necessary to visit each element, free the variable length data, and reset the element. The application must free the memory it has allocated. For memory allocated by the HDF5 Library during a read, the H5Dvlen_reclaim function can be used to perform this operation.

5.4 External Storage Properties

The External storage format allows data to be stored across a set of non-HDF5 files. 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. Currently, only the H5D_CONTIGUOUS storage format allows external storage. External storage is enabled by a Dataset Creation Property. Table 18 shows the API.

Table 18

Function	Description
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, unsigned 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()).

Figure 19 shows an example of 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.

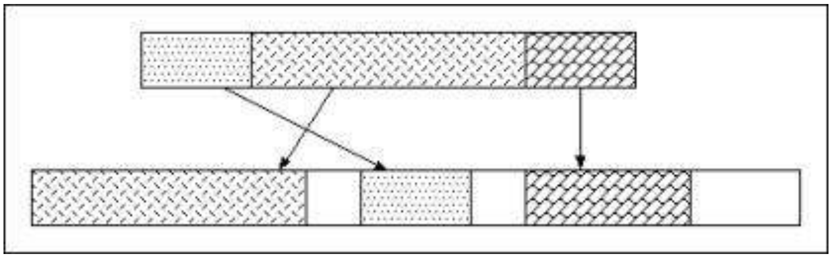


Figure 19

Figure 19a shows code that defines the external storage for the example. 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.

```
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);
```

Figure 19a

Figure 20 shows an example of 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.

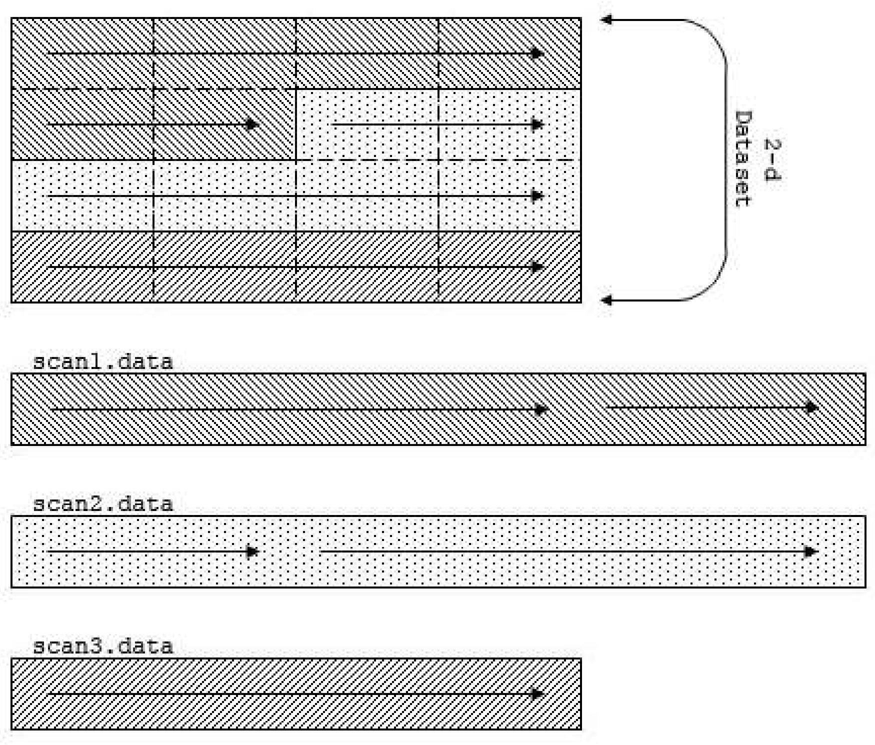


Figure 20

Figure 21 shows code for this example.

In this example, the library maps the multi-dimensional array onto a linear address space as defined by the HDF5 Format Specification, and then maps that address space into the segments defined in the external file list.

```
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);
```

Figure 21

The segments of an external file can exist beyond the end of the (external) 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 external 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. However, if the file is transferred from its original context, care must be taken to assure that all the external files are accessible in the new location.

Chapter 6
HDF5 Datatypes

1. Introduction

1.1 Introduction and Definitions

An HDF5 dataset is an array of data elements, arranged according to the specifications of the dataspace. In general, a data element is the smallest addressable unit of storage in the HDF5 file. (Compound datatypes are the exception to this rule.) The HDF5 datatype defines the storage format for a single data element (Figure 1).

The model for HDF5 attributes is extremely similar to datasets: an attribute has a dataspace and a datatype, as shown in Figure 1. The information in this chapter applies to both datasets and attributes.

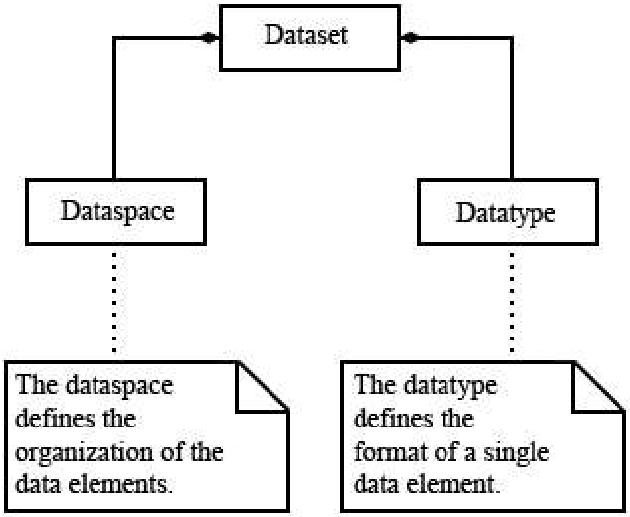

Dataset
Dataspace
Datatype
The dataspace defines the organization of the data elements.
The datatype defines the format of a single data element.

Figure 1

Abstractly, each data element within the dataset is a sequence of bits, interpreted as a single value from a set of values (e.g., a number or a character). For a given data type, there is a standard or convention for representing the values as bits, and when the bits are represented in a particular storage the bits are laid out in a specific storage scheme, e.g., as 8-bit bytes, with a specific ordering and alignment of bytes within the storage array.

HDF5 datatypes implement a flexible, extensible, and portable mechanism for specifying and discovering the storage layout of the data elements, determining how to interpret the elements (e.g., as floating point numbers), and for transferring data from different compatible layouts.

An HDF5 datatype describes one specific layout of bits, a dataset has a single datatype which applies to every data element. When a dataset is created, the storage datatype is defined, the datatype cannot be changed.

- The datatype describes the storage layout of a single data element.
- All elements of the dataset must have the same type.
- The datatype of a dataset is immutable.

When data is transferred (e.g., a read or write), each end point of the transfer has a datatype, which describes the correct storage for the elements. The source and destination may have different (but compatible) layouts, in which case the data elements are automatically transformed during the transfer.

HDF5 datatypes describe commonly used binary formats for numbers (integers and floating point) and characters (ASCII). A given computing architecture and programming language supports certain number and character representations. For example, a computer may support 8-, 16-, 32-, and 64-bit signed integers, stored in memory in little-endian byte order. These would presumably correspond to the C programming language types 'char', 'short', 'int', and 'long'.

When reading and writing from memory, the HDF5 library must know the appropriate datatype that describes the architecture specific layout. The HDF5 library provides the platform independent 'NATIVE' types, which are mapped to an appropriate datatype for each platform. So the type 'H5T_NATIVE_INT' is an alias for the appropriate descriptor for each platform.

Data in memory has a datatype

- The storage layout in memory is architecture-specific.
- The HDF5 'NATIVE' types are predefined aliases for the architecture-specific memory layout.
- The memory datatype need not be the same as the stored datatype of the dataset.

In addition to numbers and characters, an HDF5 datatype can describe more abstract classes of types, including date-times, enumerations, strings, bit strings, and references (pointers to objects in the HDF5 file). HDF5 supports several classes of composite datatypes, which are compose one or more other datatypes. In addition to the standard predefined datatypes, users can define new datatypes within the datatype classes.

The HDF5 datatype model is very general and flexible

- For common simple purposes, only predefined types will be needed
- Datatypes can be composed to create complex structured datatypes.
- If needed, users can define custom atomic datatypes.

1.2 HDF5 Datatype Model

The HDF5 Library implements an object-oriented model of datatypes. HDF5 datatypes are organized as a logical set of base types, or datatype classes. Each datatype class defines a format for representing logical values as a sequence of bits. For example the H5T_CLASS_INT is a format for representing twos complement integers of various sizes.

A datatype class is defined as a set of one or more datatype properties. A datatype property is a property of the bit string. The datatype properties are defined by the logical model of the datatype class. For example, the integer class (twos complement integers) has properties such as "signed or unsigned", "length", and "byte-order". The float class (IEEE floating point numbers) has these properties, plus "exponent bits", "exponent sign", etc.

A datatype is derived from one datatype class: a given datatype has a specific value for the datatype properties defined by the class. For example, for 32-bit signed integers, stored big-endian, the HDF5 datatype is a sub-type of integer, with the properties set to: signed=1, size=4 (bytes), byte-order=BE.

The HDF5 datatype API provides methods to create datatypes of different datatype classes, to set the datatype properties of a new datatype, and to discover the datatype properties of an existing datatype.

The datatype for a dataset is stored in the HDF5 file as part of the metadata for the dataset. A datatype can be shared by more than one dataset in the file. A datatype can optionally be stored as a named object in the file.

When transferring data (e.g., a read or write), the data elements of the source and destination storage must have compatible types. As a general rule, data elements with the same datatype class are compatible, while elements from different datatype classes are not compatible. When transferring data of one datatype to another compatible datatype, the HDF5 Library uses the datatype properties of the source and destination to automatically transform each data element. For example, when reading from data stored as 32-bit, signed integers, big-endian, into 32-bit signed integers, little-endian, the HDF5 Library will automatically swap the bytes.

Thus, data transfer operations (H5Dread, H5Dwrite, H5Aread, H5Awrite) require a datatype for both the source and the destination.

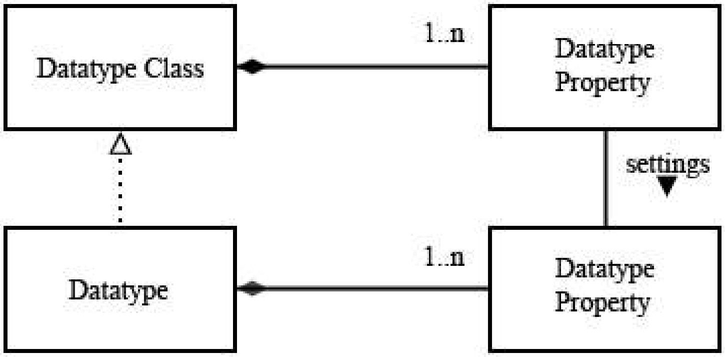


Figure 2

The HDF5 Library defines a set of predefined datatypes, corresponding to commonly used storage formats, such as twos complement integers, IEEE Floating point numbers, etc., 4- and 8-byte sizes, big endian and little endian byte orders. In addition, a user can derive types with custom values for the properties. For example, a user program may create a datatype to describe a 6-bit integer, or a 600-bit floating point number.

In addition to atomic datatypes, the HDF5 Library supports composite datatypes. A composite datatype is an aggregation of one or more datatypes. Each class of composite datatypes has properties that describe the organization of the composite datatype (Figure 3). Composite datatypes include:

- Compound datatypes: structured records
- Array: a multidimensional array of a datatype
- Variable length: a one-dimensional array of a datatype

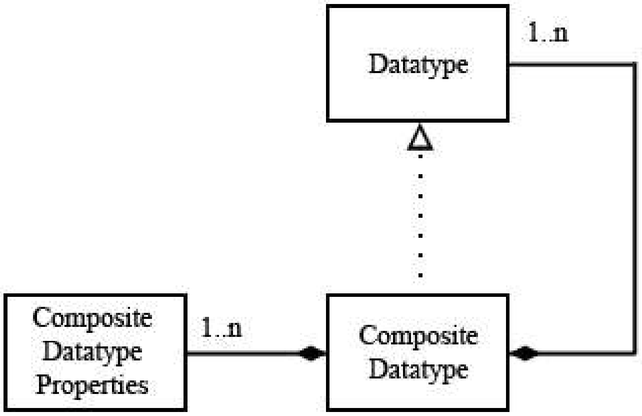


Figure 3

1.2.1 Datatype Classes and Properties

Figure 4 shows the HDF5 datatype classes. Each class is defined to have a set of properties which describe layout of the data element and the interpretation of the bits. Table 1 lists the properties for the datatype classes.

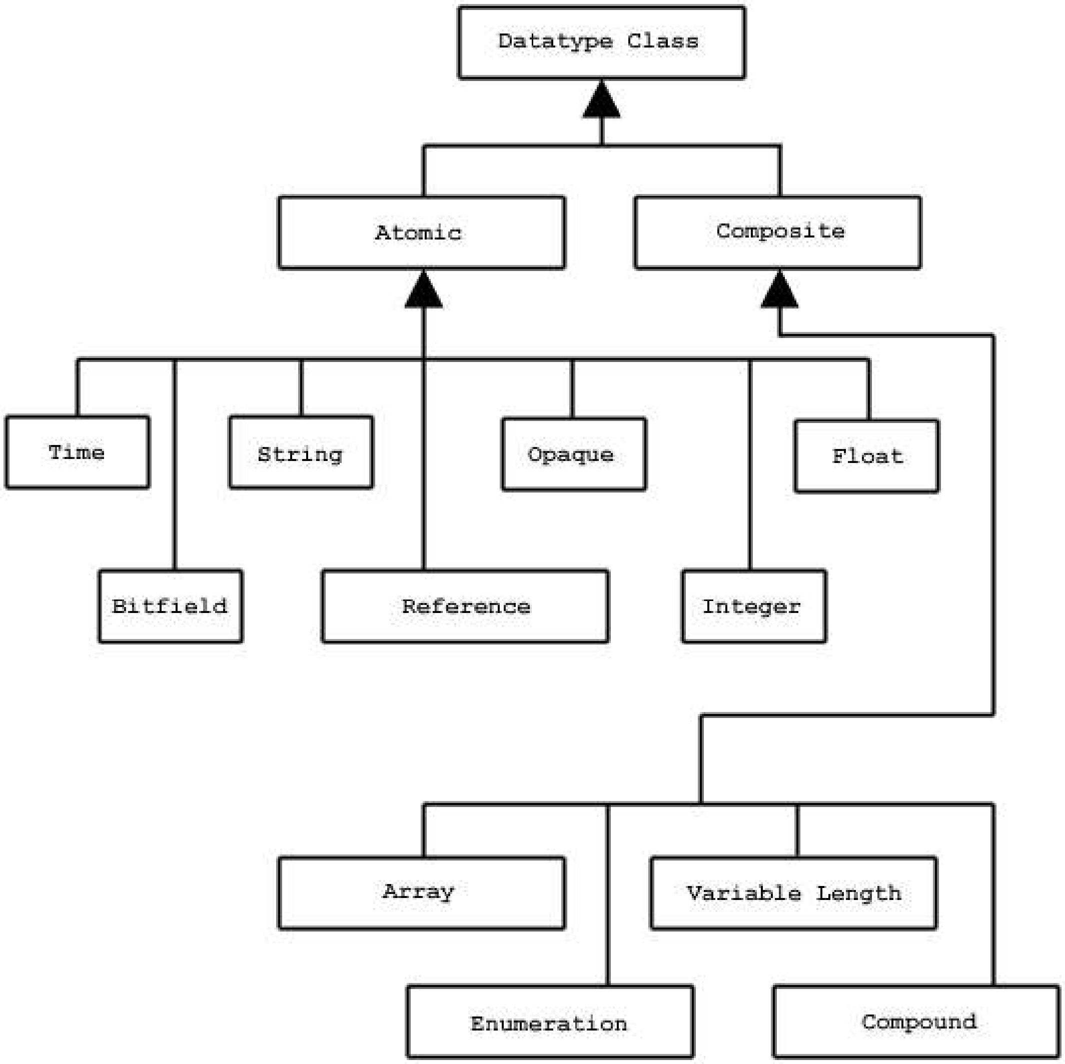


Figure 4

Table 1. Datatype Classes and their properties.

Class	Description	Properties	Notes
Integer	Twos complement integers	Size (bytes), precision (bits), offset (bits), pad, byte order, signed/unsigned	
Float	Floating Point numbers	Size (bytes), precision (bits), offset (bits), pad, byte order, sign position, exponent position, exponent size (bits), exponent sign, exponent bias, mantissa position, mantissa (size) bits, mantissa sign, mantissa normalization, internal padding	See IEEE 754 for a definition of these properties. These properties describe non-IEEE 754 floating point formats as well.
Character	Array of 1-byte character encoding	Size (characters), Character set, byte order, pad/no pad, pad character	Currently, only ASCII is supported.
Date and Time	Date/time string	Size (bytes), precision (bits), offset (bits), pad, byte order,	ISO-8601 Date/time string
Bitfield	String of bits	Size (bytes), precision (bits), offset (bits), pad, byte order	When stored, are packed into bytes
Opaque	Uninterpreted data	Size (bytes), precision (bits), offset (bits), pad, byte order, tag	A sequence of bytes, stored and retrieved as a block. The ‘tag’ is a string that can be used to label the value.
Enumeration	A list of discrete values, with symbolic names in the form of strings.	Number of elements, element names, element values	Enumeration is a list of pairs, (name, value). The name is a string, the value is an unsigned integer.
Reference	Reference to object or region within the HDF5 file		See the Reference API, H5R
Array	Array (1-4 dimensions) of data elements	Number of dimensions, dimension sizes, base datatype	The array is accessed atomically: no selection or subsetting.
Variable length	A variable length 1-dimensional array of data data elements	Current size, base type	
Compound	A Datatype composed of a sequence of Datatypes	Number of members, member names, member types, member offset, member class, member size, byte order	

1.2.2 Predefined Datatypes

The HDF5 library predefines a modest number of commonly used datatypes. These types have standard symbolic names of the form `H5T_`*`arch`*`_`*`base`* where *arch* is an architecture name and *base* is a programming type name (Table 2). New types can be derived from the predefined types by copying the predefined type (see `H5Tcopy()`) and then modifying the result.

The base name of most types consists of a letter to indicate the class (Table 3), a precision in bits, and an indication of the byte order (Table 4).

Table 5 shows examples of predefined datatypes. The full list can be found in the "HDF5 Predefined Datatypes" section of the *HDF5 Reference Manual*.

Table 2

Architecture Name	Description
`IEEE`	IEEE-754 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.

Table 3

	Bitfield
D	Date and time
F	Floating point
I	Signed integer
R	References
S	Character string
U	Unsigned integer

Table 4

BE	Big endian
LE	Little endian
VX	Vax order

Table 5

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 HDF5 Library predefines a set of NATIVE datatypes which are similar to C type names. The native types are set to be an alias for the appropriate HDF5 datatype for each platform. For example, H5T_NATIVE_INT corresponds to a C int type. On an Intel based PC, this type is the same as H5T_STD_32LE, while on a MIPS system this would be equivalent to H5T_STD_32BE. Table 6 shows examples of NATIVE types and corresponding C types for a common 32-bit workstation.

Table 6

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

2. How Datatypes Are Used

2.1 The Datatype object and the HDF5 Datatype API

The HDF5 Library manages datatypes as objects. The HDF5 datatype API manipulates the datatype objects through C function calls. 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()`.

The datatype object is used in several roles in the HDF5 model and library. Essentially, a datatype is used whenever the format of data elements is needed. There are four major uses of datatypes in the HDF5 library: at dataset creation, during data transfers, when discovering the contents of a file, and for specifying user defined data types (Table 7).

Table 7

Use	Description
Dataset creation	The datatype of the data elements must be declared when the dataset is created.
Data transfer	The datatype (format) of the data elements must be defined for both the source and destination.
Discovery	The datatype of a dataset can be interrogated to retrieve a complete description of the storage layout.
Creating User defined Datatypes	Users can define their own datatypes by creating datatype objects and setting its properties.

2.2 Dataset creation

All the data elements of a dataset have the same datatype. When a dataset is created (`H5Tcreate`), the datatype for the data elements must be specified. The datatype of a dataset can never be changed. Figure 5 shows the use of a datatype to create a dataset called "/dset". In this example, the dataset will be stored as 32-bit signed integers, in big endian order.

```
hid_t dt;
dt = H5Tcopy(H5T_STD_I32BE);
dataset_id = H5Dcreate(file_id, "/dset", dt, dataspace_id,
    H5P_DEFAULT);
```

Figure 5

2.3 Data transfer (Read and Write)

Probably the most common use of datatypes is to write or read data from a dataset or attribute. In these operations, each data element is transferred from the source to the destination (possibly rearranging the order of the elements). Since the source and destination do not need to be identical (i.e., one is disk and the other is memory) the transfer requires both the format of the source element and the destination element. Therefore, data transfers use two datatype objects, for the source and destination.

When data is written, the source is memory and the destination is disk (file). The memory datatype describes the format of the data element in the machine memory, and the file datatype describes the desired format of the data element on disk. Similarly, when reading, the source datatype describes the format of the data element on disk, and the destination datatype describes the format in memory.

In the most common cases, the file datatype is the datatype specified when the dataset was created, and the memory datatype should be the appropriate NATIVE type.

Figures 5 and 6, respectively, show examples of writing data to and reading data from a dataset. The data in memory is declared C type 'int', the datatype H5T_NATIVE_INT corresponds to this type. The datatype of the dataset should be of datatype class INTEGER.

```
int  dset_data[DATA_SIZE];

status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
      H5P_DEFAULT, dset_data);
```

Figure 6

```
int dset_data[DATA_SIZE];

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

Figure 7

2.4 Discovery of data format

The HDF5 Library enables a program to determine the datatype class and properties for any data type. In order to discover the storage format of data in a dataset, the datatype is obtained, and the properties determined by queries to the datatype object. Figure 8 shows an example of code that analyzes the datatype for an integer, and prints out a description of its storage properties (byte Order, signed, size.)

```
switch (H5Tget_class(type)) {
case H5T_INTEGER:
   ord = H5Tget_order(type);
   sgn = H5Tget_sign(type);
   printf("Integer ByteOrder= ");
   switch (ord) {
   case H5T_ORDER_LE:
       printf("LE");
       break;
   case H5T_ORDER_BE:
       printf("BE");
       break;
   }
   printf(" Sign= ");
   switch (sgn) {
   case H5T_SGN_NONE:
       printf("false");
       break;
   case H5T_SGN_2:
       printf("true");
       break;
   }
   printf(" Size= ");
   sz = H5Tget_size(type);
   printf("%d", sz);
   printf("\n");
   break;
```

Figure 8

2.5 Creating and using user defined datatypes

Most programs will primarily use the predefined datatypes described above, possibly in composite datatypes such as compound or array datatypes. However, the HDF5 datatype model is extremely general; a user program can define a great variety of atomic datatypes (storage layouts). In particular, the datatype properties can define signed and unsigned integers of any size and byte order, and floating point numbers with different formats, size, and byte order. The HDF5 datatype API provides methods to set these properties.

User defined types can be used to define the layout of data in memory, e.g., to match some platform specific number format or application defined bit-field. The user defined type can also describe data in the file, e.g., some application-defined format. The user defined types can be translated to and from standard types of the same class, as described above.

3. Datatype (H5T) Function Summaries

3.1 General Datatype Operations

C Function F90 Function	Purpose
`H5Tcreate` `h5tcreate_f`	Creates a new datatype.
`H5Topen` `h5topen_f`	Opens a named datatype.
`H5Tcommit` `h5tcommit_f`	Commits a transient datatype to a file, creating a new named datatype.
`H5Tcommitted` `h5tcommitted_f`	Determines whether a datatype is a named type or a transient type.
`H5Tcopy` `h5tcopy_f`	Copies an existing datatype.
`H5Tequal` `h5tequal_f`	Determines whether two datatype identifiers refer to the same datatype.

`H5Tlock`
`(none)`
	Locks a datatype.

`H5Tget_class` `h5tget_class_f`	Returns the datatype class identifier.
`H5Tget_size` `h5tget_size_f`	Returns the size of a datatype.
`H5Tget_super` `h5tget_super_f`	Returns the base datatype from which a datatype is derived.

`H5Tget_native_type`
`(none)`
	Returns the native datatype of a specified datatype.

`H5Tdetect_class`
`(none)`
	Determines whether a datatype is of the given datatype class.

`H5Tclose` `h5tclose_f`	Releases a datatype.

3.2 Conversion Functions

C Function F90 Function	Purpose

`H5Tconvert`
`(none)`
	Converts data from between specified datatypes.

`H5Tfind`
`(none)`
	Finds a conversion function.

`H5Tset_overflow`
`(none)`
	Sets the overflow handler to a specified function.

`H5Tget_overflow`
`(none)`
	Returns a pointer to the current global overflow function.

`H5Tregister`
`(none)`
	Registers a conversion function.

`H5Tunregister`
`(none)`
	Removes a conversion function from all conversion paths.

3.3 Atomic Datatype Properties

C Function F90 Function	Purpose
`H5Tset_size` `h5tset_size_f`	Sets the total size for an atomic datatype.
`H5Tget_order` `h5tget_order_f`	Returns the byte order of an atomic datatype.
`H5Tset_order` `h5tset_order_f`	Sets the byte ordering of an atomic datatype.
`H5Tget_precision` `h5tget_precision_f`	Returns the precision of an atomic datatype.
`H5Tset_precision` `h5tset_precision_f`	Sets the precision of an atomic datatype.
`H5Tget_offset` `h5tget_offset_f`	Retrieves the bit offset of the first significant bit.
`H5Tset_offset` `h5tset_offset_f`	Sets the bit offset of the first significant bit.
`H5Tget_pad` `h5tget_pad_f`	Retrieves the padding type of the least and most-significant bit padding.
`H5Tset_pad` `h5tset_pad_f`	Sets the least and most-significant bits padding types.
`H5Tget_sign` `h5tget_sign_f`	Retrieves the sign type for an integer type.
`H5Tset_sign` `h5tset_sign_f`	Sets the sign property for an integer type.
`H5Tget_fields` `h5tget_fields_f`	Retrieves floating point datatype bit field information.

`H5Tset_fields` `h5tset_fields_f`	Sets locations and sizes of floating point bit fields.
`H5Tget_ebias` `h5tget_ebias_f`	Retrieves the exponent bias of a floating-point type.
`H5Tset_ebias` `h5tset_ebias_f`	Sets the exponent bias of a floating-point type.
`H5Tget_norm` `h5tget_norm_f`	Retrieves mantissa normalization of a floating-point datatype.
`H5Tset_norm` `h5tset_norm_f`	Sets the mantissa normalization of a floating-point datatype.
`H5Tget_inpad` `h5tget_inpad_f`	Retrieves the internal padding type for unused bits in floating-point datatypes.
`H5Tset_inpad` `h5tset_inpad_f`	Fills unused internal floating point bits.
`H5Tget_cset` `h5tget_cset_f`	Retrieves the character set type of a string datatype.
`H5Tset_cset` `h5tset_cset_f`	Sets character set to be used.
`H5Tget_strpad` `h5tget_strpad_f`	Retrieves the storage mechanism for a string datatype.
`H5Tset_strpad` `h5tset_strpad_f`	Defines the storage mechanism for character strings.

3.4 Enumeration Datatypes

C Function F90 Function	Purpose
`H5Tenum_create` `h5tenum_create_f`	Creates a new enumeration datatype.
`H5Tenum_insert` `h5tenum_insert_f`	Inserts a new enumeration datatype member.
`H5Tenum_nameof` `h5tenum_nameof_f`	Returns the symbol name corresponding to a specified member of an enumeration datatype.
`H5Tenum_valueof` `h5tenum_valueof_f`	Returns the value corresponding to a specified member of an enumeration datatype.
`H5Tget_member_value` `h5tget_member_value_f`	Returns the value of an enumeration datatype member.
`H5Tget_nmembers` `h5tget_nmembers_f`	Retrieves the number of elements in a compound or enumeration datatype.
`H5Tget_member_name` `h5tget_member_name_f`	Retrieves the name of a compound or enumeration datatype member.

`H5Tget_member_index`
`(none)`
	Retrieves the index of a compound or enumeration datatype member.

3.5 Compound Datatype Properties

C Function F90 Function	Purpose
`H5Tget_nmembers` `h5tget_nmembers_f`	Retrieves the number of elements in a compound or enumeration datatype.

`H5Tget_member_class`
`(none)`
	Returns datatype class of compound datatype member.

`H5Tget_member_name` `h5tget_member_name_f`	Retrieves the name of a compound or enumeration datatype member.

`H5Tget_member_index`
`(none)`
	Retrieves the index of a compound or enumeration datatype member.

`H5Tget_member_offset` `h5tget_member_offset_f`	Retrieves the offset of a field of a compound datatype.
`H5Tget_member_type` `h5tget_member_type_f`	Returns the datatype of the specified member.
`H5Tinsert` `h5tinsert_f`	Adds a new member to a compound datatype.
`H5Tpack` `h5tpack_f`	Recursively removes padding from within a compound datatype.

3.6 Array Datatypes

C Function F90 Function	Purpose

`H5Tarray_create`
`(none)`
	Creates an array datatype object.

`H5Tget_array_ndims`
`(none)`
	Returns the rank of an array datatype.

`H5Tget_array_dims`
`(none)`
	Returns sizes of array dimensions and dimension permutations.

3.7 Variable-length Datatypes

C Function F90 Function	Purpose
`H5Tvlen_create` `h5tvlen_create_f`	Creates a new variable-length datatype.
`H5Tis_variable_str` `h5tis_variable_str_f`	Determines whether datatype is a variable-length string.

3.8 Opaque Datatypes

C Function F90 Function	Purpose
`H5Tset_tag` `h5tset_tag_f`	Tags an opaque datatype.
`H5Tget_tag` `h5tget_tag_f`	Gets the tag associated with an opaque datatype.

4. The Programming Model

4.1 Introduction

The HDF5 Library implements an object-oriented model of datatypes. HDF5 datatypes are organized as a logical set of base types, or datatype classes. The HDF5 Library manages datatypes as objects. The HDF5 datatype API manipulates the datatype objects through C function calls. Figure 9 shows the abstract view of the datatype object. Table 8 shows the methods (C functions) that operate on datatype object as a whole. New datatypes can be created from scratch or copied from existing datatypes.

`Datatype`
`size:int?` `byteOrder:BOtype`

`open(hid_t loc, char *, name):return hid_t`
`copy(hid_t tid) return hid_t`
`create(hid_class_t clss, size_t size) return hid_t`

Figure 9. The datatype object

Table 8. General operations on datatype objects

API function	Description
`hid_t H5Tcreate (H5T_class_t` *`class`*`, size_t` *`size`*`)`	Create a new datatype object of datatype class *class*. The following datatype classes are supported with this function: • H5T_COMPOUND • H5T_OPAQUE • H5T_ENUM Other datatypes are created with `H5Tcopy()`.
`hid_t H5Tcopy (hid_t` *`type`*`)`	Obtain a modifiable transient datatype which is a copy of *type*. If *type* is a dataset identifier then the type returned is a modifiable transient copy of the datatype of the specified dataset.
`hid_t H5Topen (hid_t` *`location`*`, const char *`*`name`*`)`	Open a named datatype. The named datatype returned by this function is read-only.
`htri_t H5Tequal (hid_t` *`type1`*`, hid_t` *`type2`*`)`	Determines if two types are equal.

`herr_t H5Tclose (hid_t type)`	Releases resources associated with a datatype obtained from H5Tcopy, H5Topen, or H5Tcreate. It is illegal to close an immutable transient datatype (e.g., predefined types).
`herr_t H5Tcommit (hid_t location, const char *name, hid_t type)`	Commit a transient datatype (not immutable) a file to become a named datatype. Named datatypes can be shared.
`htri_t H5Tcommitted (hid_t type)`	Test whether the datatype is transient or commited (named).
`herr_t H5Tlock (hid_t type)`	Make a transient datatype immutable (read-only and not closable). Predefined types are locked.

In order to use a datatype, the object must be created (`H5Tcreate`), or a reference obtained by cloning from an existing type (`H5Tcopy`), or opened (`H5Topen`). In addition, a reference to the datatype of a dataset or attribute can be obtained with `H5Dget_type` or `H5Aget_type`. For composite datatypes a reference to the datatype for members or base types can be obtained (`H5Tget_membertype`, `H5Tget_super`). When the datatype object is no longer needed, the reference is discarded with `H5Tclose`.

Two datatype objects can be tested to see if they are the same with `H5Tequal`. This function returns true if the two datatype references refer to the same datatype object. However, if two datatype objects define equivalent datatypes (the same datatype class and datatype properties), they will not be considered 'equal'.

A datatype can be written to the file as a first class object (`H5Tcommit`). Named datatypes can be used in the same way as any other dataype. Named datatypes are explained below.

4.2 Discovery of Datatype Properties

Any HDF5 datatype object can be queried to discover all of its datatype properties. For each datatype class, there are a set of API functions to retrieve the datatype properties for this class.

4.2.1 Properties of Atomic Datatypes

Table 9 lists the functions to discover the properties of atomic datatypes. Table 10 lists the queries relevant to specific numeric types. Table 11 gives the properties for atomic string datatype, and Table 12 gives the property of the opaque datatype.

Table 9

Functions to Discover Properties of Atomic DataTypes	Description
`H5T_class_t H5Tget_class (hid_t` *`type`*`)`	The datatype class: `H5T_INTEGER, H5T_FLOAT, H5T_TIME, H5T_STRING, or H5T_BITFIELD, H5T_OPAQUE, H5T_COMPOUND, H5T_REFERENCE, H5T_ENUM, H5T_VLEN, H5T_ARRAY`
`size_t H5Tget_size (hid_t` *`type`*`)`	The total size of the element in bytes, including padding which may appear on either side of the actual value.
`H5T_order_t H5Tget_order (hid_t` *`type`*`)`	The byte order describes how the bytes of the datatype are laid 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 opposite order then they are said to be *big-endian* or `H5T_ORDER_BE.`
`size_t H5Tget_precision (hid_t` *`type`*`)`	The `precision` property identifies the number of significant bits of a datatype and the `offset` property (defined below) identifies its location. 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.
`size_t H5Tget_offset (hid_t` *`type`*`)`	The `offset` property defines the bit location of the least significant bit of a bit field whose length is `precision`.
`herr_t H5Tget_pad (hid_t` *`type`*`, H5T_pad_t *`*`lsb`*`, H5T_pad_t *`*`msb`*`)`	Padding is the bits of a data element which are not significant as defined by the `precision` and `offset` properties. Padding in the low-numbered bits is *lsb* padding and padding in the high-numbered bits is *msb* padding. Padding bits can be set to zero (`H5T_PAD_ZERO`) or one (`H5T_PAD_ONE`).

Table 10

Properties of Atomic Numeric Types	Description
`H5T_sign_t H5Tget_sign (hid_t type)`	**(INTEGER)** Integer data can be signed two's complement (`H5T_SGN_2`) or unsigned (`H5T_SGN_NONE`).
`herr_t H5Tget_fields (hid_t type, size_t *spos, size_t *epos, size_t *esize, size_t *mpos, size_t *msize)`	**(FLOAT)** A floating-point data element 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 sign bit is always of length one and none of the fields are allowed to overlap.
`size_t H5Tget_ebias (hid_t type)`	**(FLOAT)** The exponent is stored as a non-negative value which is `ebias` larger than the true exponent.
`H5T_norm_t H5Tget_norm (hid_t type)`	**(FLOAT)** This property describes the normalization method of the mantissa. • `H5T_NORM_MSBSET`: 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. • `H5T_NORM_IMPLIED`: 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. • `H5T_NORM_NONE`: the fractional part of the mantissa is stored without normalizing it.
`H5T_pad_t H5Tget_inpad (hid_t type)`	**(FLOAT)** 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 padding can be: H5T_PAD_NONE, `H5T_PAD_ZERO` or `H5T_PAD_ONE`.

Table 11

Properties of Atomic String Datatypes	Description
`H5T_cset_t H5Tget_cset (hid_t` *`type`*`)`	The only character set currently supported is `H5T_CSET_ASCII`.
`H5T_str_t H5Tget_strpad (hid_t` *`type`*`)`	The string datatype has a fixed length, but the String may be shorter than the length. This property defines the storage mechanism for the left over bytes. The options are: `H5T_STR_NULLTERM`, `H5T_STR_NULLPAD`, or `H5T_STR_SPACEPAD`.

Table 12

Properties of Opaque Atomic Datatypes	Description
char *H5Tget_tag(hid_t type_id)	A user defined string.

4.2.2 Properties of Composite Datatypes

The composite datatype classes can also be analyzed to discover their datatype properties and the datatypes that are members or base types of the composite datatype. The member or base type can, in turn, be analyzed. Table 13 lists the functions that can access the datatype properties of the different composite datatypes.

Table 13

Properties of Composite Datatype	Description
`int H5Tget_nmembers(hid_t type_id )`	**(COMPOUND)** The number of fields in the compound datatype.
`H5T_class_t H5Tget_member_class( hid_t cdtype_id, int member_no )`	**(COMPOUND)** The datatype class of compound datatype member `member_no`.
`char * H5Tget_member_name(hid_t type_id, int field_idx )`	**(COMPOUND)** The name of field `field_idx` of a compound datatype.
`size_t H5Tget_member_offset(hid_t type_id, int memb_no )`	**(COMPOUND)** The byte offset of the beginning of a field within a compound datatype.
`hid_t H5Tget_member_type(hid_t type_id, int field_idx )`	**(COMPOUND)** The datatype of the specified member.
`int H5Tget_array_ndims( hid_t adtype_id )`	**(ARRAY)** The number of dimensions (rank) of the array datatype object.
`herr_t H5Tget_array_dims( hid_t adtype_id, hsize_t *dims[], int *perm[] )`	**(ARRAY)** The sizes of the dimensions and the dimension permutations of the array datatype object.
`hid_t H5Tget_super(hid_t type )`	**(ARRAY, VL, ENUM)** The base datatype from which the datatype type is derived.
`herr_t H5Tenum_nameof(hid_t type void *value, char *name, size_t size )`	**(ENUM)** The symbol name that corresponds to the specified value of the enumeration datatype
`herr_t H5Tenum_valueof(hid_t type char *name, void *value )`	**(ENUM)** The value that corresponds to the specified name of the enumeration datatype
`hid_t H5Tget_member_value(hid_t type int memb_no, void *value )`	**(ENUM)** The value of the enumeration datatype member `memb_no`

4.3 Definition of Datatypes

The HDF5 Library enables user programs to create and modify datatypes. The essential steps are:

1. a) Create a new datatype object of a specific composite datatype class, or
   b) Copy an existing atomic datatype object.
2. Set properties of the datatype object.
3. Use the datatype object.
4. Close the datatype object.

To create a user defined atomic datatype, the procedure is to clone a predefined datatype of the appropriate datatype class (`H5Tcopy`). Then set the datatype properties appropriate to the datatype class. For example, Table 14 shows how to create a datatype to describe a 1024-bit unsigned integer.

Table 14

```
hid_t new_type = H5Tcopy (H5T_NATIVE_INT);
H5Tset_precision(new_type, 1024);
H5Tset_sign(new_type, H5T_SGN_NONE);
```

Composite datatypes are created with a specific API call for each datatype class. Table 15 shows the creation method for each datatype class. A newly created datatype cannot be used until the datatype properties are set. For example, a newly created compound datatype has no members and cannot be used.

Table 15

Datatype Class	Function to Create
COMPOUND	`H5Tcreate`
OPAQUE	`H5Tcreate`
COMPOUND	`H5Tcreate`
ENUM	`H5Tenum_create`
ARRAY	`H5Tarray_create`
VL	`H5Tvlen_create`

Once the datatype is created and the datatype properties set, the datatype object can be used.

Predefined datatypes are defined by the library during initialization using the same mechanisms as described here. Each predefined datatype is locked (`H5Tlock`), so that it cannot be changed or destroyed. User defined datatypes may also be locked using `H5Tlock`.

4.3.1 User Defined Atomic Datatypes

Table 16 summarizes the API methods that set properties of atomic types. Table 17 shows properties specific to numeric types, Table 18 shows properties specific to the string datatype class. Note that offset, pad, etc. don't apply to strings. Table 19 shows the specific property of the OPAQUE datatype class.

Table 16

Functions to set Properties of Atomic DataTypes	Description
`herr_t H5Tset_size (hid_t` *`type`*`, size_t` *`size`*`)`	Set the total size of the element 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.
`herr_t H5Tset_order (hid_t` *`type`*`, H5T_order_t` *`order`*`)`	Set the byte order to little-endian (`H5T_ORDER_LE`)or big endian (`H5T_ORDER_BE`).
`herr_t H5Tset_precision (hid_t` *`type`*`, size_t` *`precision`*`)`	Set the number of significant bits of a datatype. 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.

`herr_t H5Tset_offset (hid_t` *`type`*`, size_t` *`offset`*`)`	Set the bit location of the least significant bit of a bit field whose length is `precision`. 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 significant 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.
`herr_t H5Tset_pad (hid_t` *`type`*`, H5T_pad_t` *`lsb`*`, H5T_pad_t` *`msb`*`)`	Set the padding to zeros (`H5T_PAD_ZERO`) or ones (`H5T_PAD_ONE`). Padding is the bits of a data element which are not significant as defined by the `precision` and `offset` properties. Padding in the low-numbered bits is *`lsb`* padding and padding in the high-numbered bits is *`msb`* padding.

Table 17

Properties of Numeric Types	Description
`herr_t H5Tset_sign (hid_t` *`type`*`, H5T_sign_t` *`sign`*`)`	**(INTEGER)** Integer data can be signed two's complement (`H5T_SGN_2`) or unsigned (`H5T_SGN_NONE`).
`herr_t H5Tset_fields (hid_t` *`type`*`, size_t` *`spos`*`, size_t` *`epos`*`, size_t` *`esize`*`, size_t` *`mpos`*`, size_t` *`msize`*`)`	**(FLOAT)** Set the properties define the location (bit position of least significant bit of the field) and size (in bits) of each field. The sign bit is always of length one and none of the fields are allowed to overlap.
`Herr_t H5Tset_ebias (hid_t` *`type`*`, size_t` *`ebias`*`)`	**(FLOAT)** The exponent is stored as a non-negative value which is `ebias` larger than the true exponent.
`herr_t H5Tset_norm (hid_t` *`type`*`, H5T_norm_t` *`norm`*`)`	**(FLOAT)** This property describes the normalization method of the mantissa. • `H5T_NORM_MSBSET`: 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. • `H5T_NORM_IMPLIED`: 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. • `H5T_NORM_NONE`: the fractional part of the mantissa is stored without normalizing it.
`herr_t H5Tset_inpad (hid_t` *`type`*`, H5T_pad_t` *`inpad`*`)`	**(FLOAT)** 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 padding can be: H5T_PAD_NONE, `H5T_PAD_ZERO` or `H5T_PAD_ONE`.

Table 18

Properties of Atomic String Datatypes	Description
H5Tset_size (hid_t *type*, size_t *size*)	Set the length of the string, in bytes. The precision is automatically set to 8*size.
H5Tset_precision (hid_t *type*, size_t *precision*)	The precision must be a multiple of 8.
H5Tset_cset(hid_t type_id, H5T_cset_t cset)	The only character set currently supported is H5T_CSET_ASCII.
H5Tset_strpad(hid_t type_id, H5T_str_t strpad)	The string datatype has a fixed length, but the string may be shorter than the length. This property defines the storage mechanism for the left over bytes. 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. Valid string padding values, as passed in the parameter strpad, are as follows: H5T_STR_NULLTERM (0) Null terminate (as C does) H5T_STR_NULLPAD (1) Pad with zeros H5T_STR_SPACEPAD (2) Pad with spaces (as FORTRAN does).

Table 19

Properties of Opaque Atomic Datatypes	Description
H5Tset_tag(hid_t type_id const char *tag)	Tags the opaque datatype type_id with an ASCII identifier tag.

Examples

Figure 10 shows an example of how 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). Note that the proper procedure is to begin from a type of the intended datatype class, in this case, a NATIVE INT.

```
hid_t new_type = H5Tcopy (H5T_NATIVE_INT);
H5Tset_precision (new_type, 128);
H5Tset_order (new_type, H5T_ORDER_LE);
```

Figure 10

Figure 11 shows the storage layout as the type is defined. The H5Tcopy creates a datatype that is the same as H5T_NATIVE_INT. In this example, suppose this is a 32-bit big endian number (Figure 11a). The precision is set to 128 bits, which automatically extends the size to 8 bytes (Figure 11b). Finally, the byte order is set to little-endian (Figure 11c).

Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678901

a) The H5T_NATIVE_INT

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
01234567	89012345	67890123	45678901	23456789	01234567	89012345	67890123

b) Precision extended to 128-bits, the size is automatically adjusted.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
01234567	89012345	67890123	45678901	23456789	01234567	89012345	67890123

c) The Byte Order is switched.

Figure 11

The significant bits of a data element can be offset from the beginning of the memory for that element by an amount of padding. The `offset` property specifies the number of bits of padding that appear to the "right of" the value. Table 20 and Figure 12 shows how a 32-bit unsigned integer with 16-bits of precision having the value `0x1122` will be laid out in memory.

Table 20

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]

Big-Endian: Offset = 0

Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
PPPPPPPP	*PPPPPPPP*	00010001	00100010

Big-Endian: Offset = 16

Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
00010001	00100010	*PPPPPPPP*	*PPPPPPPP*

Little-Endian: Offset = 0

Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
00010001	00100010	*PPPPPPPP*	*PPPPPPPP*

Little-Endian: Offset = 16

Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
PPPPPPPP	*PPPPPPPP*	00010001	00100010

Figure 12

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.

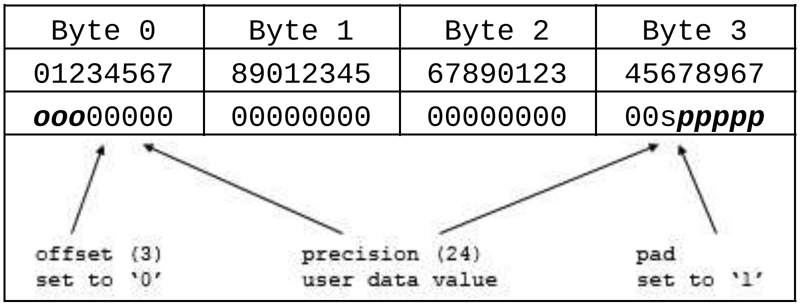
The bits of the entire data are numbered beginning at zero at the least significant bit of the least significant byte (the byte at the lowest memory address for a little-endian type or the byte at the highest address for a big-endian type). The `offset` property defines the bit location of the least 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.

To illustrate the properties of the integer datatype class, Figure 13 shows how to create a user defined datatype that describes a 24-bit signed integer that starts on the third bit of a 32-bit word. The datatype is specialized from a 32-bit integer, the *precision* is set to 24 bits, and the *offset* is set to 3.

```
hid_t dt;

dt = H5Tcopy(H5T_SDT_32LE);

H5Tset_precision(dt, 24);
H5Tset_offset(dt,3);
H5Tset_pad(dt, H5T_PAD_ZERO,H5T_PAD_ONE);
```

Figure 13

Figure 14 shows the storage layout for a data element. Note that the unused bits in the offset will be set to zero and the unused bits at the end will be set to one, as specified in the `H5Tset_pad` call.



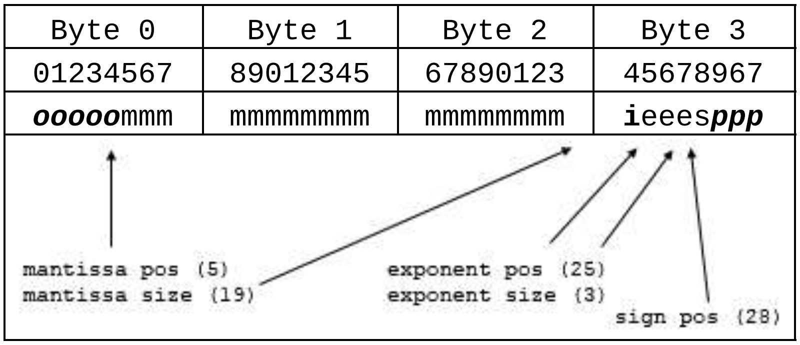
Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
ooo00000	00000000	00000000	00s***ppppp***

Figure 14. A User defined integer Datatype: range -1,048,583 to 1,048,584

To illustrate a user defined floating point number, Figure 13 shows how to create a 24-bit floating point number, that starts 5 bits into a 4 byte word. The floating point number is defined to have a mantissa of 19 bits (bits 5-23), and exponent of 3 bits (25-27) and the sign bit is bit 28. (Note that this is an illustration of what can be done, not necessarily a floating point format that a user would require.)

```
hid_t dt;

dt = H5Tcopy(H5T_IEEE_32LE);

H5Tset_precision(dt, 24);
H5Tset_fields (dt, 28, 25, 3, 5, 19);
H5Tset_pad(dt, H5T_PAD_ZERO,H5T_PAD_ONE);
H5Tset_inpad(dt, H5T_PAD_ZERO);
```

Figure 15



Byte 0	Byte 1	Byte 2	Byte 3
01234567	89012345	67890123	45678967
ooooommm	mmmmmmmm	mmmmmmmm	ieees***ppp***

Figure 16. A User defined Floating Point Datatype.

Figure 16 shows the storage layout of a data element for this datatype. Note that there is an unused bit (24) between the mantissa and the exponent. This bit is filled with the *inpad* value, in this case 0.

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.

4.3.2 Composite Datatypes

All composite datatypes must be user defined, there are no predefined composite datatypes.

4.3.2.1 Compound Datatypes

The subsections below describe how compound datatypes are created and how to write and read data of compound datatype.

4.3.2.1.1 Defining Compound Datatypes

Compound datatypes are conceptually similar to a C struct or Fortran 95 derived types. The compound datatype defines a contiguous sequence of bytes, which are formatted using one up to 2^16 datatypes (members). A compound datatype may have any number of members, in any order, and the members may have any datatype, including compound. Thus, complex nested compound datatypes can be created. The total size of the compound datatype is greater than or equal to the sum of the size of its members, up to a maximum of 2^32 bytes. HDF5 does not support datatypes with distinguished records or the equivalent of C unions or Fortran 95 EQUIVALENCE statement.

Usually a C struct or Fortran derived type 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 HDF5 C libary provides a macro `HOFFSET(s,m)` to calculate the member's ofset. The HDF5 Fortran applications have to calculate offsets by using sizes of members datatypes and by taking in consideration the order of members in the Fortran derived type.

`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.

Note for Fortran users: Offsets of Fortran structure members correspond to the offsets within a packed datatype (see explanation below) stored in an HDF5 file.

Each member of a compound datatype 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 of Fortran derived type, although this is often the case. Nor does one need to define all members of the C struct of Fortran derived type in the HDF5 compound datatype (or vice versa).

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 its total size. Then members are added to the compound datatype in any order. Each member type is inserted at a designated offset. Each member has a name which is the key used to uniquely identify the member within the compound datatype.

Figure 17a shows an example of creating an HDF5 C compound datatype to describe a complex number, which is a structure with two components, "real" and "imagenery", each double. An equivalent C `struct` is whose type is defined by the `complex_t struct`, is shown.

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

  hid_t complex_id = H5Tcreate (H5T_COMPOUND, sizeof(complex_t));
  H5Tinsert (complex_id, "real", HOFFSET(complex_t,re),
             H5T_NATIVE_DOUBLE);
  H5Tinsert (complex_id, "imaginary", HOFFSET(complex_t,im),
             H5T_NATIVE_DOUBLE);
```

Figure 17a

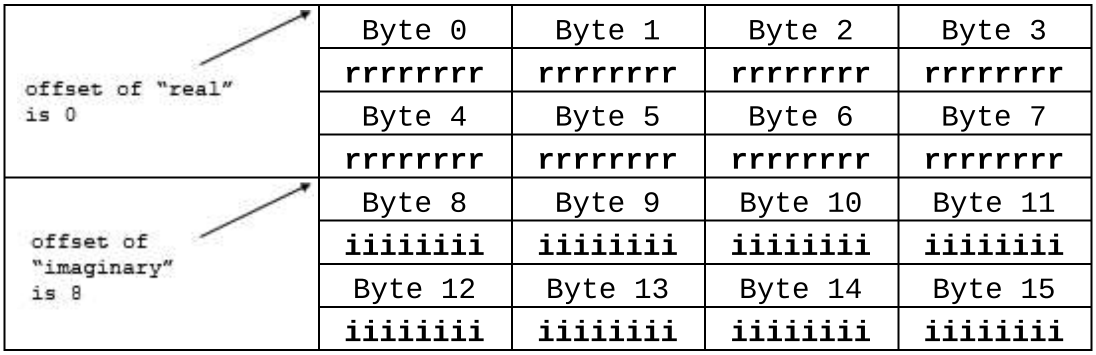
Figure 17b shows an example of creating an HDF5 Fortran compound datatype to describe a complex number, which is a Fortran derived type with two components, "real" and "imagenary", each DOUBLE PRECISION. An equivalent Fortran `TYPE` is whose type is defined by the `TYPE complex_t`, is shown.

```
TYPE complex_t
   DOUBLE PRECISION re   ! real part
   DOUBLE PRECISION im;  ! imaginary part
END TYPE complex_t

CALL h5tget_size_f(H5T_NATIVE_DOUBLE, re_size, error)
CALL h5tget_size_f(H5T_NATIVE_DOUBLE, im_size, error)
complex_t_size = re_size + im_size
CALL h5tcreaet_f(H5T_COMPOUND_F, complex_t_size, type_id)
offset = 0
CALL h5tinsert_f(type_id, "real", offset, H5T_NATIVE_DOUBLE, error)
offset = offset + re_size
CALL h5tinsert_f(type_id, "imaginary", offset, H5T_NATIVE_DOUBLE, error)
```

Figure 17b

Important Note: The compound datatype is created with a size sufficient to hold all its members. In the C example above, the size of the C `struct` and the HOFFSET macro are used as a convenient mechanism to determine the appropriate size and offset. Alternatively, the size and offset could be manually determined, e.g., the size can be set to 16 with "real" at offset 0 and "imaginary" at offset 8. However, different platforms and compilers have different sizes for "double", and may have alignment restrictions which require additional padding within the structure. It is much more portable to use the HOFFSET macro, which assures that the values will be correct for any platform.

Figure 18 shows how the compound datatype would be laid out, assuming that NATIVE_DOUBLE are 64-bit numbers, and there are no alignment requirements. The total size of the compound datatype will be 16 bytes, the "real" component will start at byte 0, and "imaginary" will start at byte 8.

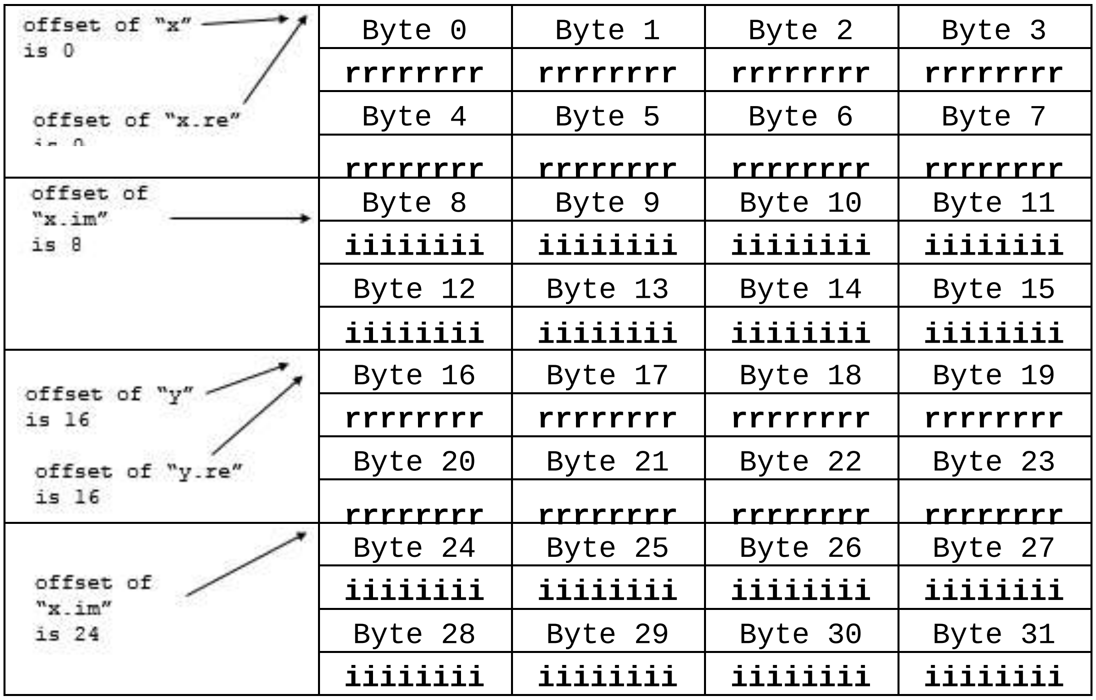


offset of “real” is 0	Byte 0	Byte 1	Byte 2	Byte 3
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
	Byte 4	Byte 5	Byte 6	Byte 7
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
offset of “imaginary” is 8	Byte 8	Byte 9	Byte 10	Byte 11
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii
	Byte 12	Byte 13	Byte 14	Byte 15
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii

Total size of Compound Datatype is 16 bytes

Figure 18

The members of a compound datatype may be any HDF5 datatype, including compound, array, and VL. Figures 19 and 20 show an example which creates a compound datatype composed of two complex values, each of which is a compound datatype as in Figure 18 above.



offset of “x” is 0 offset of “x.re” is 0	Byte 0	Byte 1	Byte 2	Byte 3
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
	Byte 4	Byte 5	Byte 6	Byte 7
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
offset of “x.im” is 8	Byte 8	Byte 9	Byte 10	Byte 11
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii
	Byte 12	Byte 13	Byte 14	Byte 15
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii
offset of “y” is 16 offset of “y.re” is 16	Byte 16	Byte 17	Byte 18	Byte 19
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
	Byte 20	Byte 21	Byte 22	Byte 23
	rrrrrrrr	rrrrrrrr	rrrrrrrr	rrrrrrrr
offset of “x.im” is 24	Byte 24	Byte 25	Byte 26	Byte 27
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii
	Byte 28	Byte 29	Byte 30	Byte 31
	iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii

Total size of Compound Datatype is 32 bytes.

Figure 19

```
 typedef struct {
    complex_t x;
    complex_t y;
 } surf_t;

hid_t complex_id, surf_id; /*hdf5 datatypes*/

 complex_id = H5Tcreate (H5T_COMPOUND, sizeof(complex_t));
 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(surf_t));
 H5Tinsert (surf_id, "x", HOFFSET(surf_t,x), complex_id);
 H5Tinsert (surf_id, "y", HOFFSET(surf_t,y), complex_id);
```

Figure 20

Note that a similar result could be accomplished by creating a compound datatype and inserting four fields (Figure 21). This results in the same layout as above (Figure 19). The difference would be how the fields are addressed. In the first case, the real part of 'y' is called 'y.re'; in the second case it is 'y-re'.

```
typedef struct {
    complex_t x;
    complex_t y;
 } surf_t;

 hid_t surf_id = H5Tcreate (H5T_COMPOUND, sizeof(surf_t));
 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);
```

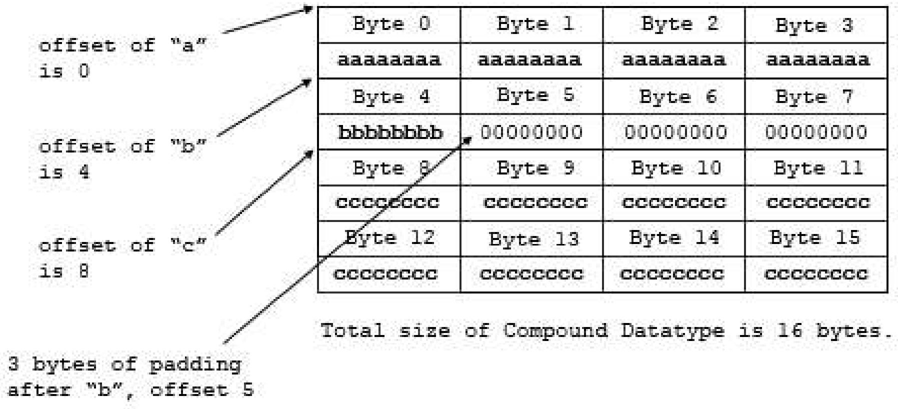
Figure 21

The members of a compound datatype do not always fill all the bytes. The HOFFSET macro assures that the members will be laid out according to the requirements of the platform and language. Figure 22 shows an example of a C struct which requires extra bytes of padding on many platforms. The second element, 'b', is a 1-byte character, followed by an 8 byte double, 'c'. On many systems, the 8-byte value must be stored on a 4- or 8-byte boundary, requiring the struct to be larger than the sum of the size of its elements.

In Figure 22 , the `sizeof` and `HOFFSET` macro is used to assure that the members are inserted at the correct offset to match the memory conventions of the platform. Figure 23 shows how this data element would be stored in memory, assuming the double must start on a 4-byte boundary. Notice the extra bytes between 'b' and 'c'.

```
typedef struct s1_t {
   int    a;
   char  b;
   double c;
} s1_t;

s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_CHAR);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
```

Figure 22



Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
Byte 4	Byte 5	Byte 6	Byte 7
bbbbbbbb	00000000	00000000	00000000
Byte 8	Byte 9	Byte 10	Byte 11
cccccccc	cccccccc	cccccccc	cccccccc
Byte 12	Byte 13	Byte 14	Byte 15
cccccccc	cccccccc	cccccccc	cccccccc

offset of "a" is 0

offset of "b" is 4

offset of "c" is 8

3 bytes of padding after "b", offset 5

Total size of Compound Datatype is 16 bytes.

Figure 23

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`):

Figure 24a shows how to create a disk version of the compound datatype from Figure 22 above in order to store data on disk in as compact a form as possible. Figure 25 shows the layout of the bytes in the packed data structure. 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 s2_tid = H5Tcopy (s1_tid);
               H5Tpack (s2_tid);
```

Figure 24a

Figure 24b shows the sequence of Fortran calls to create a packed compound datatype. An HDF5 Fortran compound datatype never describes a compound datatype in memory and compound data is *ALWAYS* written by fields as described in the next section. Therefore packing is not needed unless the the offset of each consecutive member is not equal to the sum of the sizes of the previous members.

```
CALL h5tcopy_f(s1_id, s2_id, error)
CALL h5tpack_f(s2_id, error)
```

Figure 24b

4.3.2.1.2 Creating, writing and reading datasets with compound datatypes

Creating datasets with compound datatypes is similar to creating datasets with any other HDF5 datatypes. But writing and reading may be different since datsets that have compound datatypes can be written or read by a field (member) or subsets of fields (members). The compound datatype is the only compoiste datatype that supports "sub-setting" by the elements the datatype is built from.

Figure 25a shows C example of creating and writing a dataset with a compound datatype.

```
     typedef struct s1_t {
        int    a;
        float  b;
        double c;
     } s1_t;

     s1_t data[LENGTH];

     /* Initialize data */
     for (i = 0; i <LENGTH; i++) {
          data[i].a = i;
          data[i].b = i*i;
          data[i].c = 1./(i+1);
     ...
     s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
     H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
     H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_FLOAT);
     H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
     ...
     dataset_id = H5Dcreate(file_id, "SDScompound.h5", s1_t, space_id,
H5PDEFAULT);
     H5Dwrite (dataset_id,s1_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);
```

Figure 25a

Figure 25b shows the content of the file written on the liitle-endian machine.

```
HDF5 "SDScompound.h5" {
GROUP "/" {
    DATASET "ArrayOfStructures" {
       DATATYPE  H5T_COMPOUND {
          H5T_STD_I32LE "a_name";
          H5T_IEEE_F32LE "b_name";
          H5T_IEEE_F64LE "c_name";
       }
       DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
       DATA {
       (0): {
             0,
             0,
             1
          },
       (1): {
             1,
             1,
             0.5
          },
       (2): {
             2,
             4,
             0.333333
          }
       }
    }
}
}
```

Figure 25b

It is not necessary to write the whole data at once. Datasets with compound datatypes can be written by field. In order to do this one has to remember to set transfer property of the dataset using `H5Pset_preserve` call and to define memory datatype that corresponds to a field. Figure 25c shows how field b is written to the dataset.

```
typedef struct sb_t {
   float  b;
   double c;
} sb_t;

typedef struct sc_t {
   float  b;
   double c;
} sc_t;
sb_t data1[LENGTH];
sc_t data2[LENGTH];

/* Initialize data */
for (i = 0; i <LENGTH; i++) {
     data1.b = i*i;
     data2.c = 1./(i+1);
}
...
/* Create dataset as in example 25a */
```

```
     ...
     /* Create memory datatypes corresponding to float and double
datatype fileds */

     sb_tid = H5Tcreate (H5T_COMPOUND, sizeof(sb_t));
     H5Tinsert(sb_tid, "b_name", HOFFSET(sb_t, b), H5T_NATIVE_FLOAT);
     sc_tid = H5Tcreate (H5T_COMPOUND, sizeof(sc_t));
     H5Tinsert(sc_tid, "c_name", HOFFSET(sc_t, c), H5T_NATIVE_DOUBLE);
     ...
     /* Set transfer property */
     xfer_id = H5Pcreate(H5P_DATASET_XFER);
     H5Pset_preserve(xfer_id, 1);
     H5Dwrite (dataset_id,sb_tid, H5S_ALL, H5S_ALL, xfer_id, data1);
     H5Dwrite (dataset_id,sc_tid, H5S_ALL, H5S_ALL, xfer_id, data2);
```

Figure 25c

Figure 25d shows the content of the file written on the little-endian machine. Only float and double fileds are written. Default fill value is used to initialize unwritten integer filed.

```
HDF5 "SDScompound.h5" {
GROUP "/" {
    DATASET "ArrayOfStructures" {
       DATATYPE  H5T_COMPOUND {
          H5T_STD_I32LE "a_name";
          H5T_IEEE_F32LE "b_name";
          H5T_IEEE_F64LE "c_name";
       }
       DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
       DATA {
       (0): {
             0,
             0,
             1
          },
       (1): {
             0,
             1,
             0.5
          },
       (2): {
             0,
             4,
             0.333333
          }
       }
    }
}
}
```

Figure 25d

Figure 25e contains a Fortran example that creates and writes a dataset with a compound datatype. As this example illustrates, writing and reading compound datatypes in Fortran is *always* done by fields. The content of the written file is the same as shown in the Figure 25b.

```
! One cannot write an array of a derived datatype in Fortran.
TYPE s1_t
   INTEGER          a
   REAL             b
   DOUBLE PRECISION c
END TYPE s1_t
TYPE(s1_t) d(LENGTH)

! Therefore, the following code initializes an array corresponding
! to each field in the derived datatype and writes those arrays
! to the dataset

INTEGER, DIMENSION(LENGTH)          :: a
REAL, DIMENSION(LENGTH)             :: b
DOUBLE PRECISION, DIMENSION(LENGTH) :: c

! Initialize data
 do i = 1, LENGTH
    a(i) = i-1
    b(i) = (i-1) * (i-1)
    c(i) = 1./i
 enddo

...

 ! Set dataset transfer property to preserve partially initialized fields
 ! during write/read to/from dataset with compound datatype.
 !
 CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
 CALL h5pset_preserve_f(plist_id, .TRUE., error)
...
 !
 ! Create compound datatype.
 !
 ! First calculate total size by calculating sizes of each member
 !
 CALL h5tget_size_f(H5T_NATIVE_INTEGER, type_sizei, error)
 CALL h5tget_size_f(H5T_NATIVE_REAL, type_sizer, error)
 CALL h5tget_size_f(H5T_NATIVE_DOUBLE, type_sized, error)
 type_size = type_sizei + type_sizer + type_sized
 CALL h5tcreate_f(H5T_COMPOUND_F, type_size, dtype_id, error)
 !
 ! Insert memebers
 !
 !
 ! INTEGER member
 !
 offset = 0
 CALL h5tinsert_f(dtype_id, "a_name", offset, H5T_NATIVE_INTEGER, error)
 !
 ! REAL member
 !
 offset = offset + type_sizei
 CALL h5tinsert_f(dtype_id, "b_name", offset, H5T_NATIVE_REAL, error)
 !
 ! DOUBLE PRECISION member
 !
 offset = offset + type_sizer
 CALL h5tinsert_f(dtype_id, "c_name", offset, H5T_NATIVE_DOUBLE, error)

 !
 ! Create the dataset with compound datatype.
 !
 CALL h5dcreate_f(file_id, dsetname, dtype_id, dspace_id, &
                  dset_id, error)
 !
...
 ! Create memory types. We have to create a compound datatype
 ! for each member we want to write.
 !
 !
 CALL h5tcreate_f(H5T_COMPOUND_F, type_sizei, dt1_id, error)
 offset = 0
 CALL h5tinsert_f(dt1_id, "a_name", offset, H5T_NATIVE_INTEGER, error)
 !
 CALL h5tcreate_f(H5T_COMPOUND_F, type_sizer, dt2_id, error)
 offset = 0
 CALL h5tinsert_f(dt2_id, "b_name", offset, H5T_NATIVE_REAL, error)
 !
 CALL h5tcreate_f(H5T_COMPOUND_F, type_sized, dt3_id, error)
 offset = 0
 CALL h5tinsert_f(dt3_id, "c_name", offset, H5T_NATIVE_DOUBLE, error)
 !
 ! Write data by fields in the datatype. Fields order is not important.
 !
 CALL h5dwrite_f(dset_id, dt3_id, c, data_dims, error, xfer_prp = plist_id)
 CALL h5dwrite_f(dset_id, dt2_id, b, data_dims, error, xfer_prp = plist_id)
 CALL h5dwrite_f(dset_id, dt1_id, a, data_dims, error, xfer_prp = plist_id)
```

Figure 25e

4.3.2.2 Array

Many scientific datasets have multiple measurements for each point in a space. There are several natural ways to represent this data, depending on the variables and how they are used in computation (Table 21).

Table 21

Storage Strategy	Stored as	Remarks
Mulitple planes	Several datasets with identical dataspaces	This is optimal when variables are accessed individually, or when often uses only selected variables.
Additional dimension	One dataset, the last "dimension" is a vector of variables	This can give good performance, although selecting only a few variables may be slow. This may not reflect the science.
Record with multiple values	One dataset with compound datatype	This enables the variables to be read all together or selected. Also handles "vectors" of heterogenous data.
Vector or Tensor value	One dataset, each data element is a small array of values.	This uses the same amount of space as the previous two, and may represent the science model better.

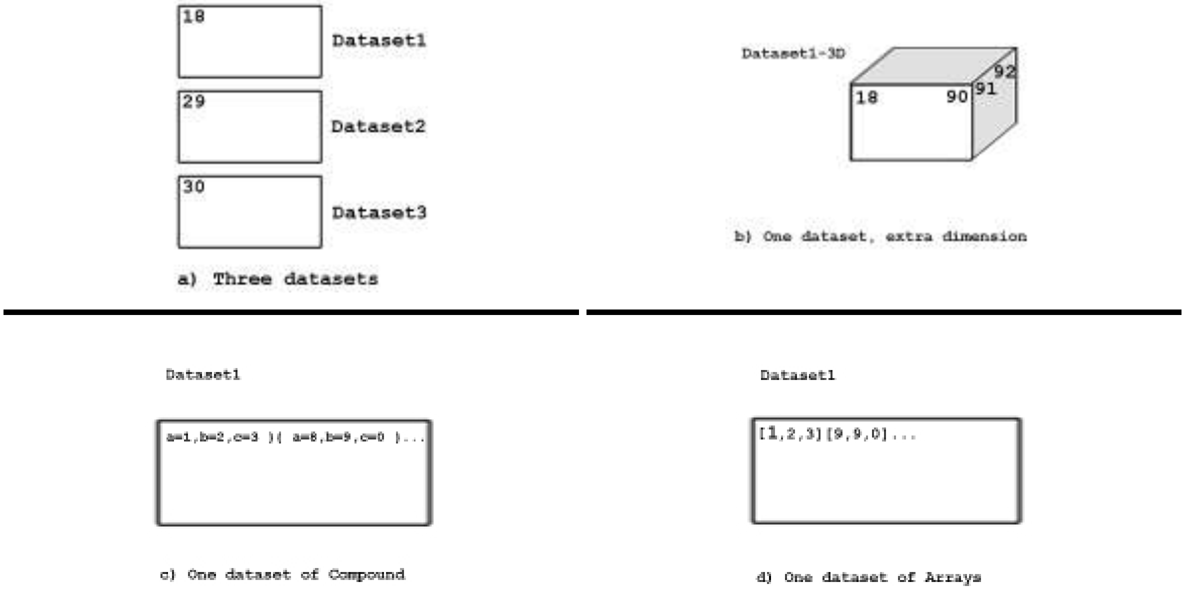


Figure 26

The HDF5 H5T_ARRAY datatype defines the data element to be a homogeneous, multi-dimensional array, as in Figure 25d, above. The elements of the array can be any HDF5 datatype (including compound and array), the size of the datatype is the total size of the array. A dataset of array datatype cannot be subdivided for I/O within the data element, the entire array of the data element must be transferred. If the data elements need to be accessed separated, e.g., by plane, then the array datatype should not be used. Table 22 gives advantages and disadvantages of the storage methods.

Table 22

Method	Advantages	Disadvantages
a) Multiple Datasets	• Easy to access each plane, can select any plane(s).	• Less efficient to access a 'column' through the planes
b) N+1 Dimension	• All access patterns supported.	• Must be homogeneous datatype. • The added dimension may not make sense in the scientific model.
c) Compound Datatype	• Can be heterogenous datatype.	• Planes must be named, selection is by plane. • Not a natural representation for a matrix.
d) Array	• Cannot access elements separately (no access by plane)	• A natural representation for vector or tensor data.

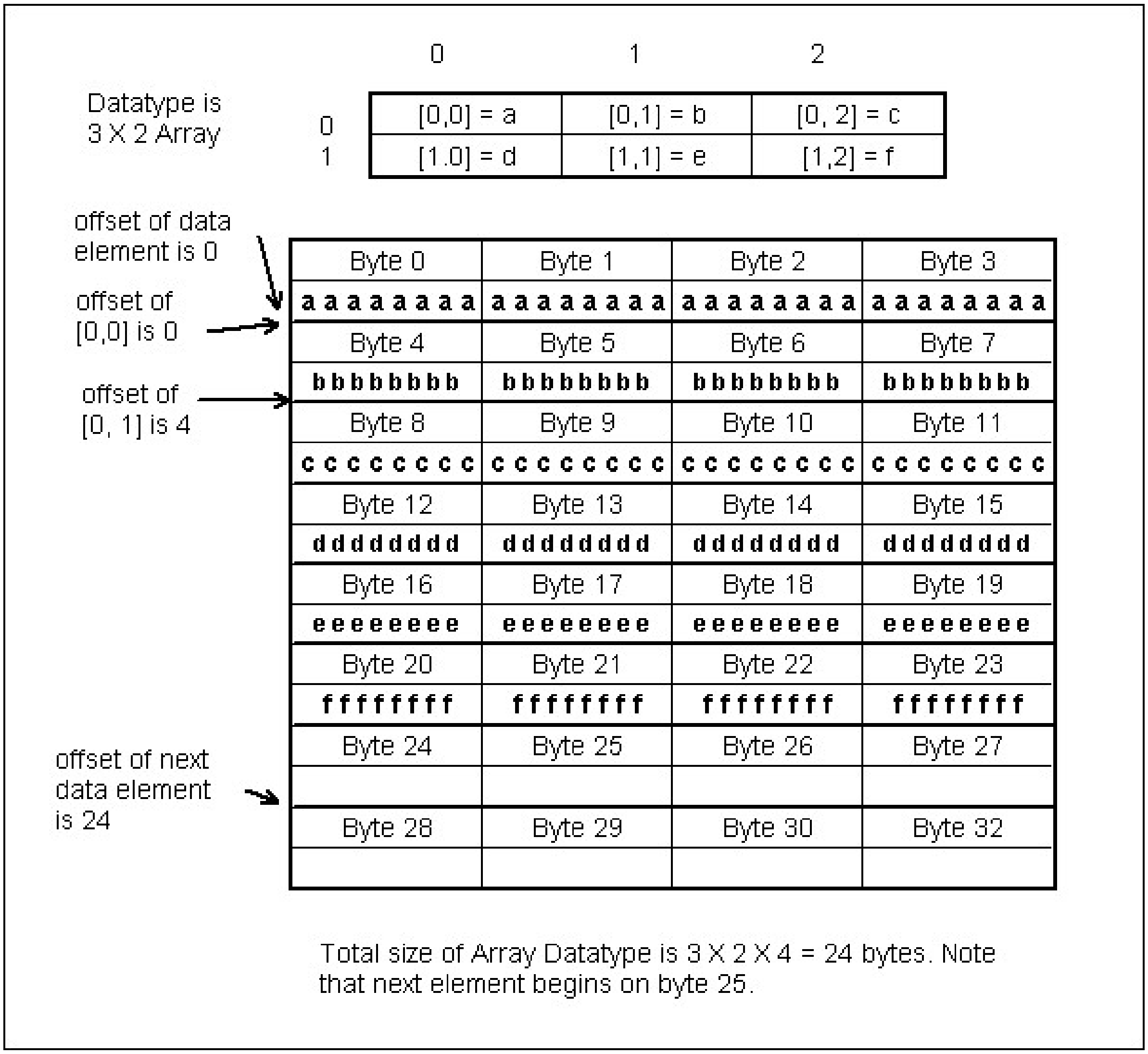
An array datatype may be multi-dimensional, with 1 to H5S_MAX_RANK (the maximum rank of a dataset is currently 32). The dimensions can be any size greater than 0, but unlimited dimensions are not supported (although the datatype can be a variable length datatype).

- An array datatype may be multi-dimensional, with 1 to H5S_MAX_RANK (the maximum rank of a dataset is currently 32).
- The elements of the array can be any HDF5 datatype (including compound and array),
- An array datatype element cannot be subdivided for I/O, the entire array of the data element must be transferred.

An array datatype is created with the `H5Tarray_create` call, which specifies the number of dimensions, the size of each dimension, and the base type of the array. The array datatype can then be used in any way that any datatype object is used. Figure 27 shows the creation of a datatype that is a two-dimensional array of native integers, which is then used to create a dataset. Note that the dataset can a dataspace that is any number and size of dimensions. Figure 28 shows the layout in memory, assuming that the native integers are 4 bytes. Each data element has 6 elements, for a total of 24 bytes.

```
hid_t       file, dataset;
hid_t       datatype, dataspace;
hsize_t     adims[] = {3, 2};

datatype = H5Tarray_create(H5T_NATIVE_INT, 2, adims, NULL);

dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace,
      H5P_DEFAULT);
```

Figure 27



	0	1	2
0	[0,0] = a	[0,1] = b	[0, 2] = c
1	[1.0] = d	[1,1] = e	[1,2] = f

Byte 0	Byte 1	Byte 2	Byte 3
a a a a a a a a	a a a a a a a a	a a a a a a a a	a a a a a a a a
Byte 4	Byte 5	Byte 6	Byte 7
b b b b b b b b	b b b b b b b b	b b b b b b b b	b b b b b b b b
Byte 8	Byte 9	Byte 10	Byte 11
c c c c c c c c	c c c c c c c c	c c c c c c c c	c c c c c c c c
Byte 12	Byte 13	Byte 14	Byte 15
d d d d d d d d	d d d d d d d d	d d d d d d d d	d d d d d d d d
Byte 16	Byte 17	Byte 18	Byte 19
e e e e e e e e	e e e e e e e e	e e e e e e e e	e e e e e e e e
Byte 20	Byte 21	Byte 22	Byte 23
f f f f f f f f	f f f f f f f f	f f f f f f f f	f f f f f f f f
Byte 24	Byte 25	Byte 26	Byte 27
Byte 28	Byte 29	Byte 30	Byte 32

Total size of Array Datatype is 3 X 2 X 4 = 24 bytes. Note that next element begins on byte 25.

Figure 28

4.3.2.3 Variable-length (VL) Datatypes

A variable-length (VL) datatype is a one-dimensional sequence of a datatype which are not fixed in length from one dataset location to another, i.e., each data element may have a different number of members. Variable-length datatypes cannot be divided, the entire data element must be transferred.

VL datatypes are useful to the scientific community in many different ways, possibly including:

- *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.
- *Fractal arrays*: A nested VL datatype 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. A 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, e.g. of objects within the file*: 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.
- *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.

A VL datatype is created by calling `H5Tvlen_create`, which specifies the base datatype. Figure 29 shows an example of code that creates a VL datatype of unsigned integers. Each data element is a one-dimensional array of zero or more members, which must be stored in a structure, `hvl_t` (Figure 30).

```
tid1 = H5Tvlen_create (H5T_NATIVE_UINT);

dataset=H5Dcreate(fid1,"Dataset1",tid1,sid1,H5P_DEFAULT);
```

Figure 29

```
typedef struct  {
    size_t len; /* Length of VL data (in base type units) */
    void *p;    /* Pointer to VL data */
} hvl_t;
```

Figure 30

Figure 31 shows how the VL data is written. For each of the 10 data elements, a length and data buffer must be allocated. Figure 33 shows how the data is laid out in memory.

An analogous procedure must be used to read the data (Figure 32). An appropriate array of `vl_t` must be allocated, and the data read. It is then traversed one data element at a time. The `H5Dvlen_free` call frees the data buffer for the buffer. 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 (set with `H5Pset_vlen_mem_manager`). Since the memory allocated when reading (or writing) may be complicated to release, the `H5Dvlen_reclaim`) is provided to traverse a memory buffer and free the VL datatype information without leaking memory.

```
hvl_t wdata[10];   /* Information to write */

/* Allocate and initialize VL data to write */
for(i=0; i <10; i++) {
    wdata[i].p = malloc((i+1)*sizeof(unsigned int));
    wdata[i].len = i+1;
    for(j=0; j
```

Figure 31

```
hvl_t rdata[SPACE1_DIM1];
ret=H5Dread(dataset,tid1,H5S_ALL,H5S_ALL,xfer_pid,rdata);

for(i=0; i<SPACE1_DIM1; i++) {
  printf("%d: len %d ",rdata[i].len);
  for(j=0; j<rdata[i].len; j++) {
     printf(" value: %u\n",((unsigned int *)rdata[i].p)[j]);
  }
}
ret=H5Dvlen_reclaim(tid1,sid1,xfer_pid,rdata);
```

Figure 32

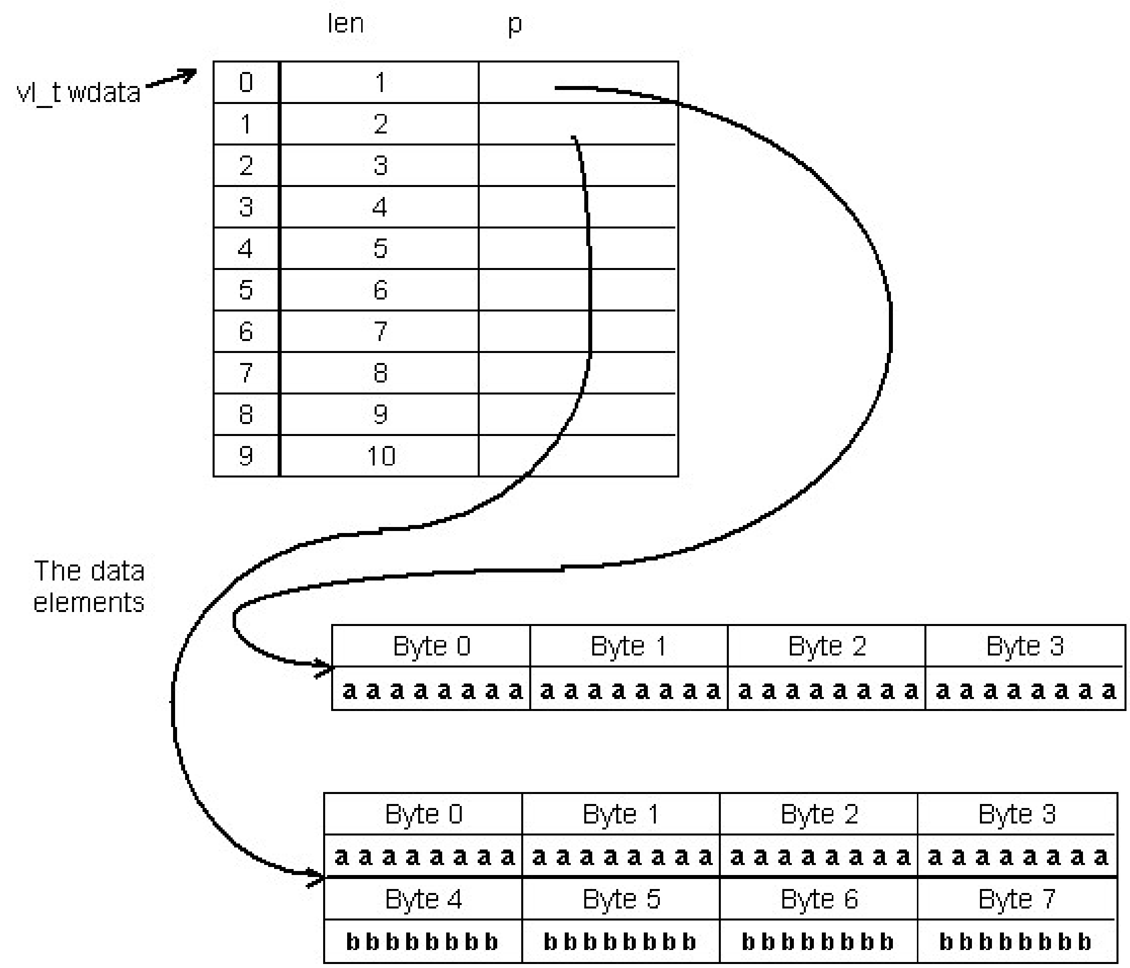


Figure 33

The user program must carefully manage these relatively complex data structures, such as suggested by Figure 33. The `H5Dvlen_reclaim` function performs a standard traversal, freeing all the data. This function analyzes the datatype and dataspace objects, and visits each VL data element, recursing through nested types. By default, the system `free` is called for the pointer in each `vl_t`. Obviously, this call assumes that all of this memory was allocated with the system `malloc`.

The user program may specify custom memory manager routines, one for allocating and one for freeing. These may be set with the `H5Pvlen_mem_manager`, and must have the following prototypes:

- `typedef void *(*H5MM_allocate_t)(size_t size, void *info);`
- `typedef void (*H5MM_free_t)(void *mem, void *free_info);`

The utility function `H5Dget_vlen_buf_size` checks the number of bytes required to store the VL data from the dataset. This function analyzes the datatype and dataspace object to visit all the VL data elements, to determine the number of bytes required to store the data for the in the destination storage (memory). The `size` value is adjusted for data conversion and alignment in the destination.

5. Other Non-numeric Datatypes

Several datatype classes define special types of objects.

5.1 Strings

Text data is represented by arrays of characters, called strings. Many programming languages support different conventions for storing strings, which may be fixed or variable length, and may have different rules for padding unused storage. HDF5 can represent strings in several ways.

The Strings to store are: "Four score",
"lazy programmers."

a) H5T_NATIVE_CHAR
The dataset is a one-dimensional array with 29 elements, each element is a single character.

0	1	2	3	4	...	25	26	27	28
'F'	'o'	'u'	'r'	' '	...	'r'	's'	'.'	'\0'

b) Fixed-length string
The dataset is a one-dimensional array with 2 elements, each element is 20 characters.

0	`"Four score\0        "`
1	`"lazy programmers.\0"`

c) Variable Length string
The dataset is a one-dimensional array with 2 elements, each element is a variable-length string.
This is the same result when stored as fixed-length string, except that first element of the array will need only 11 bytes for storage instead of 20.

0	`"Four score\0"`
1	`"lazy programmers.\0"`

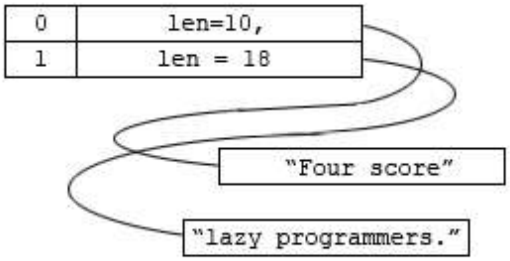


Figure 34

First, a dataset may have a dataset with datatype H5T_NATIVE_CHAR, with each character of the string as an element of the dataset. This will store an unstructured block of text data, but gives little indication of any structure in the text (Figure 34a).

A second alternative is to store the data using the datatype class H5T_STRING, with each element a fixed length (Figure 34b). In this approach, each element might be a word or a sentence, addressed by the dataspace. The dataset reserves space for the specified number of characters, although some strings may be shorter. This approach is simple and usually is fast to access, but can waste storage space if the length of the Strings varies.

A third alternative is to use a variable-length datatype (Figure 34c). This can be done using the standard mechanisms described above (e.g., using `H5T_NATIVE_CHAR` instead of `H5T_NATIVE_INT` in Figure 29 above). The program would use `vl_t` structures to write and read the data.

A fourth alternative is to use a special feature of the string datatype class, to set the size of the datatype to `H5T_VARIABLE` (Figure 34c). Figure 35 shows a declaration of a datatype of type `H5T_C_S1`, which is set to `H5T_VARIABLE`. The HDF5 Library automatically translates between this and the `vl_t` structure. (Note: the `H5T_VARIABLE` size can only be used with string datatypes.)

```
   tid1 = H5Tcopy (H5T_C_S1);

ret = H5Tset_size (tid1,H5T_VARIABLE);
```

Figure 35

Variable-length strings can be read into C strings (i.e., pointers to zero terminated arrays of `char`) (Figure 36).

```
char *rdata[SPACE1_DIM1];

ret=H5Dread(dataset,tid1,H5S_ALL,H5S_ALL,xfer_pid,rdata);

for(i=0; i<SPACE1_DIM1; i++) {
          printf("%d: len: %d, str is: %s\n", strlen(rdata[i]),rdata[I]);
}

ret=H5Dvlen_reclaim(tid1,sid1,xfer_pid,rdata);
```

Figure 36

5.2 Reference

In HDF5, objects (i.e. groups, datasets, and named datatypes) are usually accessed by name. There is another way to access stored objects -- by reference. There are two reference datatypes, object reference and region reference. Object reference objects are created with the `H5Rcreate` and other calls (cross reference). These objects can be stored and retrieved in a dataset as elements with reference datatype. Figure 37 shows an example of code that creates references to four objects, and then writes the array of object references to a dataset. Figure 38 shows a dataset of datatype reference being read, and one of the object reference objects being dereferenced to obtain an object pointer.

In order to store references to regions of a dataset, the datatype should be H5T_REGION_OBJ. Note that a data element must be either an object reference or a region reference: these are different types and cannot be mixed within a single array.

A reference datatype cannot be divided for I/O, an element is read or written completely.

```
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);
```

Figure 37

```
  rbuf = malloc(sizeof(hobj_ref_t)*SPACE1_DIM1);

 /* 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]);
```

Figure 38

5.3 ENUM

The enum datatype implements a set of (name, value) pairs, similar to C/C++ enum. The values are currently limited to integer dataype class. Each name can be the name of only one value, and each value can have only one name. There can be up to 2^16 different names for a given enumeration.

The data elements of the ENUMERATION are stored according to the datatype, e.g., as an array of integers. Figure 39 shows an example of how to create an enumeration with five elements. The elements map symbolic names to 2-byte integers (Table 23).

```
hid_t hdf_en_colors = H5Tcreate(H5T_ENUM, sizeof(short));
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));

     H5Dcreate(fileid,spaceid,hdf_en_colors,H5P_DEFAULT);
```

Figure 39

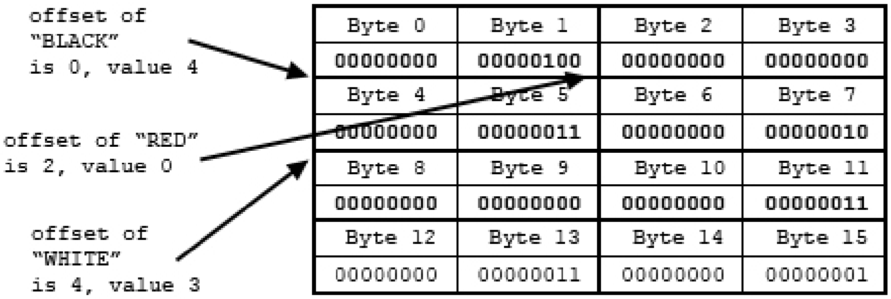
Table 23

Name	Value
RED	0
GREEN	1
BLUE	2
WHITE	3
BLACK	4

Figure 40 shows how an array of eight values might be stored. Conceptually, the array is an array of symbolic names [BLACK, RED, WHITE, BLUE,] (Figure 40a). These are stored as the values, i.e., as short integers. So, the first 2 bytes are the value associated with "BLACK", which is the number 4, and so on (Figure 40b).

a) Logical Data to be written 8 (elements)

Index	Name
0	:BLACK
1	RED
2	WHITE
3	BLUE
4	RED
5	WHITE
6	BLUE
7	GREEN


offset of "BLACK" is 0, value 4
offset of "RED" is 2, value 0
offset of "WHITE" is 4, value 3

Byte 0	Byte 1	Byte 2	Byte 3
00000000	00000100	00000000	00000000
Byte 4	Byte 5	Byte 6	Byte 7
00000000	00000011	00000000	00000010
Byte 8	Byte 9	Byte 10	Byte 11
00000000	00000000	00000000	00000011
Byte 12	Byte 13	Byte 14	Byte 15
00000000	00000011	00000000	00000001

b) The storage layout. Total size of the array is 16 bytes, 2 bytes per element.

Figure 40

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.

5.4 Opaque

In some cases, a user may have data objects that should be stored and retrieved as blobs, with no attempt to interpret them. For example, an application might wish to store an array of encrypted certificates, which are 100 bytes long

While an arbitrary block of data may always be stored as bytes, characters, integers, or whatever, this might mislead programs about the meaning of the data. The opaque datatype defines data elements which are uninterpreted by HDF5. The opaque data may be labeled with H5Tset_tag, with a string that might be used by an application. For example, the encrypted certificates might have a tag to indicate the encryption and the certificate standard.

5.5 Bitfield

Some data is represented as bits, where the number of bits is not an integral byte and the bits are not necessarily interpreted as a standard type. Some examples might include readings from machine registers (e.g., switch positions), a cloud mask, or data structures with several small integers that should be store in a single byte.

This data could be stored as integers, strings, or enumerations. However, these storage methods would likely have considerable wasted space. For example, storing a cloud mask with one byte per value would use 8 times the space of a packed array of bits. Similarly, the status of an inst

The HDF5 bitfield dataype class defines a data element that is a contiguous sequence of bits, which are stored on disk in a packed array. The programming model is the same as for unsigned integers: the dataype object is created by copying a predefined datatype, and then the precision, offset, and padding are set.

5.6 Time

The HDF5 time datatype defines storage layout for various date and time standards. Currently, only Unix "time" and "timeval" structs are supported. The H5T_UNIX_D32BE (LE) defines storage for 4 bytes (sufficient for the time struct), H5T_UNIX_D64BE (LE) is sufficient for timeval. The data is treated as a single opaque value.

6. Fill Values

The "fill value" for a dataset is the specification of the default value assigned to data elements that have not yet been written. In the case of a dataset with an atomic datatype, the fill value is a single value of the appropriate datatype, such as '0' or '-1.0'. In the case of a dataset with a composite datatype, the "fill value" is a single data element of the appropriate type. For example, for an array or compound datatype, the "fill value" is a single data element with values for all the component elements of the array or compound datatype.

The fill value is set (permanently) when the dataset is created. The fill value is set in the dataset creation properties in the `H5Dcreate` call. Note that the `H5Dcreate` call must also include the datatype of the dataset, and the value provided for the fill value will be interpreted as a single element of this datatype. Figure 41 shows example code which creates a dataset of integers with fill value -1. Any unwritten data elements will be set to -1.

```
    hid_t       plist_id;
    int filler;

    filler = -1;
    plist_id = H5Pcreate(H5P_DATASET_CREATE);
    H5Pset_fill_value(plist,H5T_NATIVE_INT,&filler);

    /* Create the dataset with filel value '-1'. */
    dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE,
dataspace_id, plist);
```

Figure 41

```
typedef struct s1_t {
   int    a;
   char  b;
   double c;
} s1_t;
s1_t       filler;

s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_CHAR);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);

filler.a = -1;
filler.b = '*';
filler.c = -2.0;

plist_id = H5Pcreate(H5P_DATASET_CREATE);
H5Pset_fill_value(plist_id, s1_tid, &filler);

/* Create the dataset with fill value (-1, '*', -2.0). */
dataset = H5Dcreate(file, DATASETNAME, s1_tid, space, plist_id);
```

Figure 42

Figure 42 shows how to create a "fill value" for a compound datatype. The procedure is the same as the previous example, except the filler must be a structure with the correct fields. Each field is initialized to the desired fill value.

The fill value for a dataset can be retrieved by reading the dataset creation properties of the dataset, and then reading the fill value with `H5Pget_fill_value`. The data will be read into memory using the storage layout specified by the datatype. This transfer will convert data in the same way as `H5Dread`. Figure 43 shows how to get the fill value from the dataset created in Figure 41 above.

```
hid_t plist2;
int filler;

dataset_id = H5Dopen(file_id, "/dset" );
plist2 = H5Dget_create_plist(dataset_id);

H5Pget_fill_value(plist, H5T_NATIVE_INT, &filler);

/* filler has the fill value, '-1' */
```

Figure 43

A similar procedure is followed for any datatype. Figure 45 shows how to read the fill value created in Figure 42. Note that the program must pass an element large enough to hold a fill value of the datatype indicated by the argument to `H5Pget_fill_value`. Also, the program must understand the datatype in order to interpret its components. This may be difficult to determine without knowledge of the application that created the dataset.

```
char *       fillbuf;
int sz;
dataset = H5Dopen( file, DATASETNAME);

s1_tid = H5Dget_type(dataset);

sz = H5Tget_size(s1_tid);

fillbuf = (char *)malloc(sz);

plist_id = H5Dget_create_plist(dataset);

H5Pget_fill_value(plist_id, s1_tid, fillbuf);

printf("filler.a: %d\n",((s1_t *) fillbuf)->a);
printf("filler.b: %c\n",((s1_t *) fillbuf)->b);
printf("filler.c: %f\n",((s1_t *) fillbuf)->c);
```

Figure 44

7. Complex Combinations of Datatypes

7.1

Several composite datatype classes define collections of other datatypes, including other composite datatypes. In general, a datatype can be nested to any depth, with any combination of datatypes.

For example, a compound datatype can have members that are other compound datatypes, arrays, VL datatypes. An array can be an array of array, an array of compound, or an array of VL. And a VL datatype can be a variable length array of compound, array, or VL datatypes.

These complicated combinations of datatypes form a logical tree, with a single root datatype, and leaves which must be atomic datatypes (predefined or user-defined). Figure 45 shows an example of a logical tree describing a compound datatype constructed from different datatypes.

Recall that the datatype is a description of the layout of storage. The complicated compound datatype is constructed from component datatypes, each of which describe the layout of part of the storage. Any datatype can be used as a component of a compound datatype, with the following restrictions:

1. No byte can be part of more than one component datatype (i.e., the fields cannot overlap within the compound datatype).
2. The total size of the components must be less than or equal to the total size of the compound datatype.

These restrictions are essentially the rules for C structures and similar record types familiar from programming languages. Multiple typing, such as a C union, is not allowed in HDF5 datatypes.

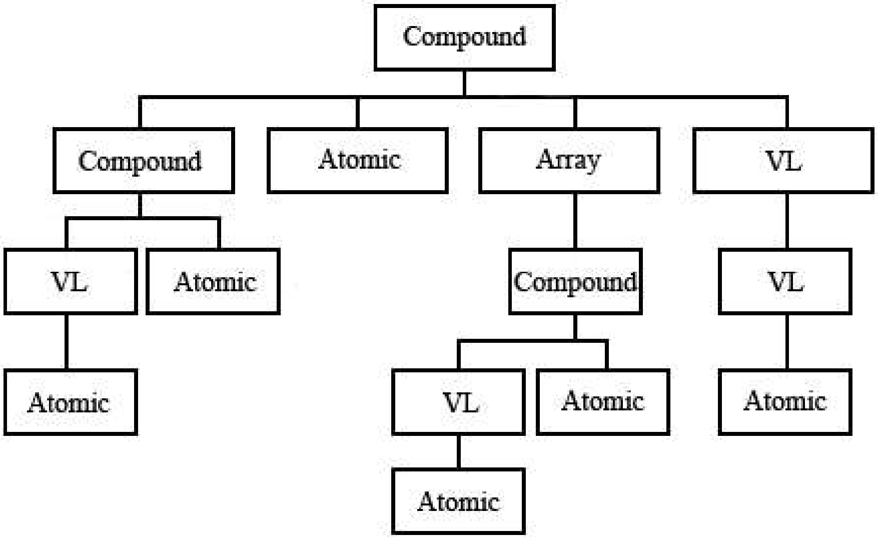


Figure 45

7.2 Creating a complicated compound datatype

To construct a complicated compound datatype, each component is constructed, and then added to the enclosing datatype description. Figure 46 shows some example code to create a compound datatype with four members:

- "T1", a compound datatype with three members
- "T2", a compound datatype with two members
- "T3", a one-dimensional array of integers
- "T4", a string

This datatype is shown as a logical tree in Figure 47, the output of the *h5dump* utility is shown in Figure 48.

Each datatype is created as a separate datatype object. Figure 49 shows the storage layout for the four individual datatypes. Then the dataypes are inserted into the outer datatype at an appropriate offset. Figure 50 shows the resulting storage layout. The combined record is 89 bytes long.

The Dataset is created using the combined compound datatype. The dataset is declared to be a 4 by 3 array of compound data. Each data element is an instance of the 89-byte compound datatype. Figure 51 shows the layout of the dataset, and expands one of the elements to show the relative position of the component data elements.

Each data element is a compound datatype, which can be written or read as a record, or each field may be read or written individually. The first field ("T1") is itself a compound datatype with three fields ("T1.a", "T1.b", and "T1.c"). "T1" can be read or written as a record, or individual fields can be accessed. Similarly, the second filed is a compound datatype with two fields ("T2.f1", "T2.f2").

The third field ("T3") is an array datatype. Thus, "T3" should be accessed as an array of 40 integers. Array data can only be read or written as a single element, so all 40 integers must be read or written to the third field. The fourth field ("T4") is a single string of length 25.

```
typedef struct s1_t {
   int    a;
   char  b;
   double c;
} s1_t;

typedef struct s2_t {
   float f1;
   float f2;
} s2_t;
hid_t      s1_tid, s2_tid, s3_tid, s4_tid, s5_tid;

/* Create a datatype for s1 */
s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_CHAR);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);

/* Create a data type for s2. *.
s2_tid = H5Tcreate (H5T_COMPOUND, sizeof(s2_t));
H5Tinsert(s2_tid, "f1", HOFFSET(s2_t, f1), H5T_NATIVE_FLOAT);
H5Tinsert(s2_tid, "f2", HOFFSET(s2_t, f2), H5T_NATIVE_FLOAT);

/* Create a datatype for an Array of integers */
s3_tid = H5Tarray_create(H5T_NATIVE_INT, RANK, dim, NULL);

/* Create a data type for a String of 25 characters */
s4_tid = H5Tcopy(H5T_C_S1);
H5Tset_size(s4_tid, 25);

/*
 * Create a compound datatype composed of one of each of these
 *  types.
 * The total size is the sum of the size of each.
 */

sz = H5Tget_size(s1_tid) + H5Tget_size(s2_tid) + H5Tget_size(s3_tid)
     + H5Tget_size(s4_tid);

s5_tid = H5Tcreate (H5T_COMPOUND, sz);

/* insert the component types at the appropriate offsets */

H5Tinsert(s5_tid, "T1", 0, s1_tid);
H5Tinsert(s5_tid, "T2", sizeof(s1_t), s2_tid);
H5Tinsert(s5_tid, "T3", sizeof(s1_t)+sizeof(s2_t), s3_tid);
H5Tinsert(s5_tid, "T4", (sizeof(s1_t) +sizeof(s2_t)+
        H5Tget_size(s3_tid)), s4_tid);

/*
 * Create the dataset with this datatype.
 */
dataset = H5Dcreate(file, DATASETNAME, s5_tid, space, H5P_DEFAULT);
```

Figure 46

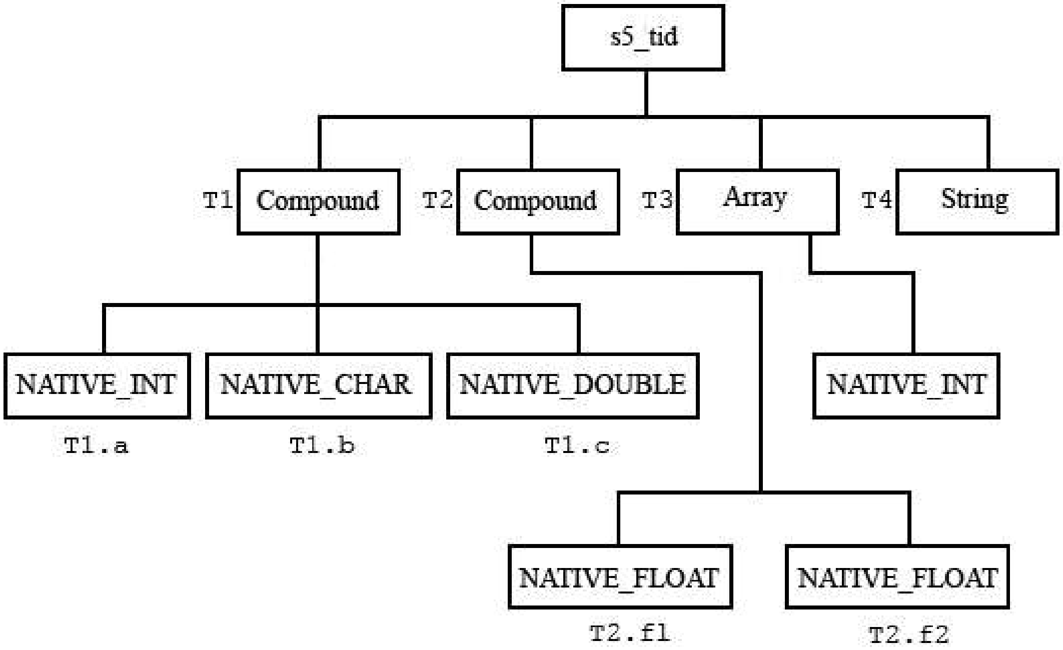


Figure 47

```
  DATATYPE  H5T_COMPOUND {
   H5T_COMPOUND {
      H5T_STD_I32LE "a_name";
      H5T_STD_I8LE "b_name";
      H5T_IEEE_F64LE "c_name";
   } "T1";
   H5T_COMPOUND {
      H5T_IEEE_F32LE "f1";
      H5T_IEEE_F32LE "f2";
   } "T2";
   H5T_ARRAY { [10] H5T_STD_I32LE } "T3";
   H5T_STRING {
      STRSIZE 25;
      STRPAD H5T_STR_NULLTERM;
      CSET H5T_CSET_ASCII;
      CTYPE H5T_C_S1;
   } "T4";
}
```

Figure 48

a) Compound type 's1_t', size 16 bytes.

Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
Byte 4	Byte 5	Byte 6	Byte 7
bbbbbbbb			
Byte 8	Byte 9	Byte 10	Byte 11
cccccccc	cccccccc	cccccccc	cccccccc
Byte 12	Byte 13	Byte 14	Byte 15
cccccccc	cccccccc	cccccccc	cccccccc

b) Compound type 's2_t', size 8 bytes.

Byte 0	Byte 1	Byte 2	Byte 3
ffffffff	ffffffff	ffffffff	ffffffff
Byte 4	Byte 5	Byte 6	Byte 7
gggggggg	gggggggg	gggggggg	gggggggg

c) Array type 's3_tid', 40 integers, total size 40 bytes.

Byte 0	Byte 1	Byte 2	Byte 3
00000000	00000000	00000000	00000000
Byte 4	Byte 5	Byte 6	Byte 7
00000000	00000000	00000000	00000001

...

Byte 36	Byte 37	Byte 38	Byte 39
00000000	00000000	00000000	00001010

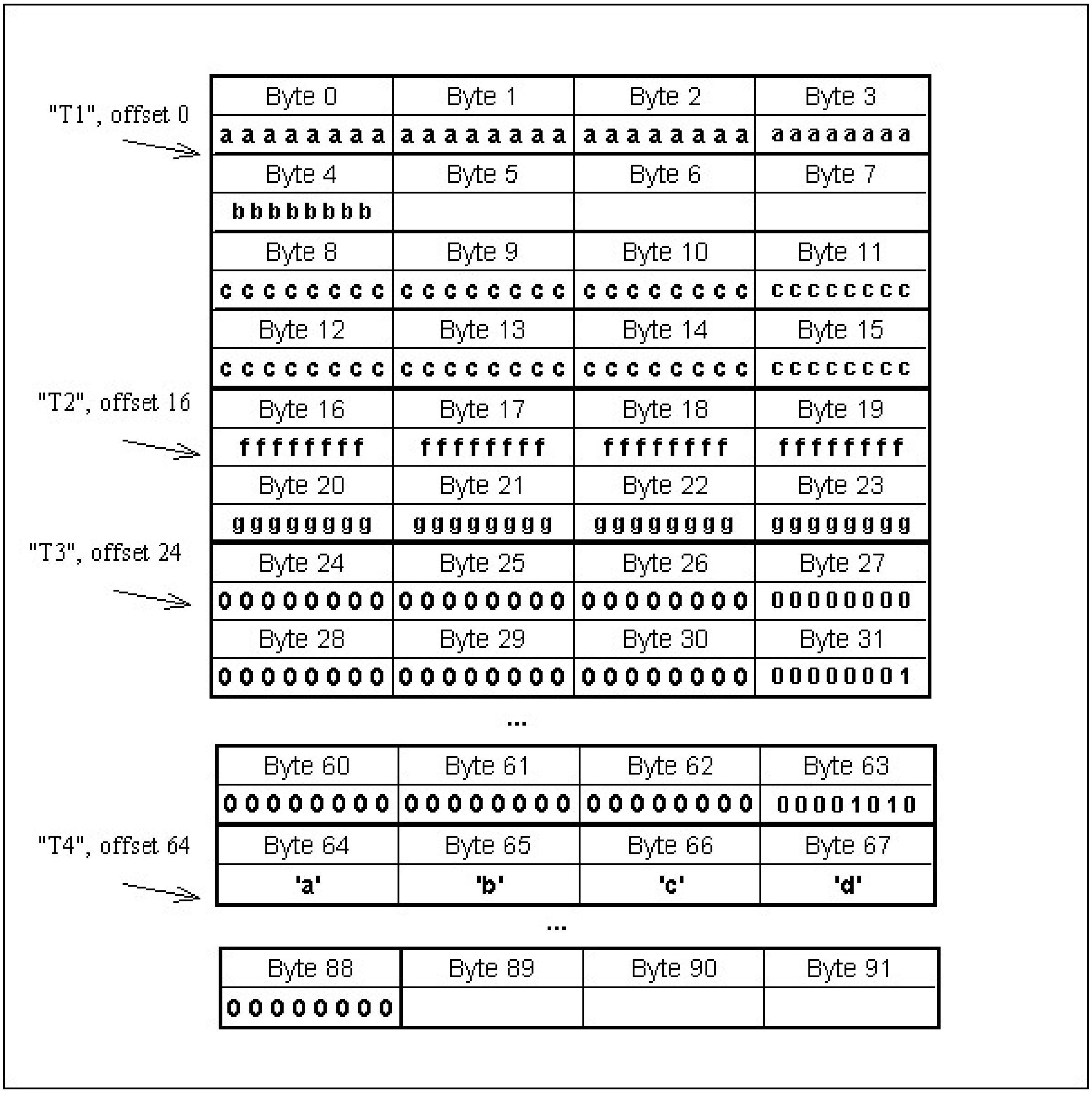
d) String type 's4_tid', size 25 bytes.

Byte 0	Byte 1	Byte 2	Byte 3
'a'	'b'	'c'	'd'

...

Byte 24	Byte 25	Byte 26	Byte 27
00000000			

Figure 49



Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
Byte 4	Byte 5	Byte 6	Byte 7
bbbbbbbb			
Byte 8	Byte 9	Byte 10	Byte 11
cccccccc	cccccccc	cccccccc	cccccccc
Byte 12	Byte 13	Byte 14	Byte 15
cccccccc	cccccccc	cccccccc	cccccccc
Byte 16	Byte 17	Byte 18	Byte 19
ffffffff	ffffffff	ffffffff	ffffffff
Byte 20	Byte 21	Byte 22	Byte 23
gggggggg	gggggggg	gggggggg	gggggggg
Byte 24	Byte 25	Byte 26	Byte 27
00000000	00000000	00000000	00000000
Byte 28	Byte 29	Byte 30	Byte 31
00000000	00000000	00000000	00000001

Byte 60	Byte 61	Byte 62	Byte 63
00000000	00000000	00000000	00001010
Byte 64	Byte 65	Byte 66	Byte 67
'a'	'b'	'c'	'd'

Byte 88	Byte 89	Byte 90	Byte 91
00000000			

Figure 50

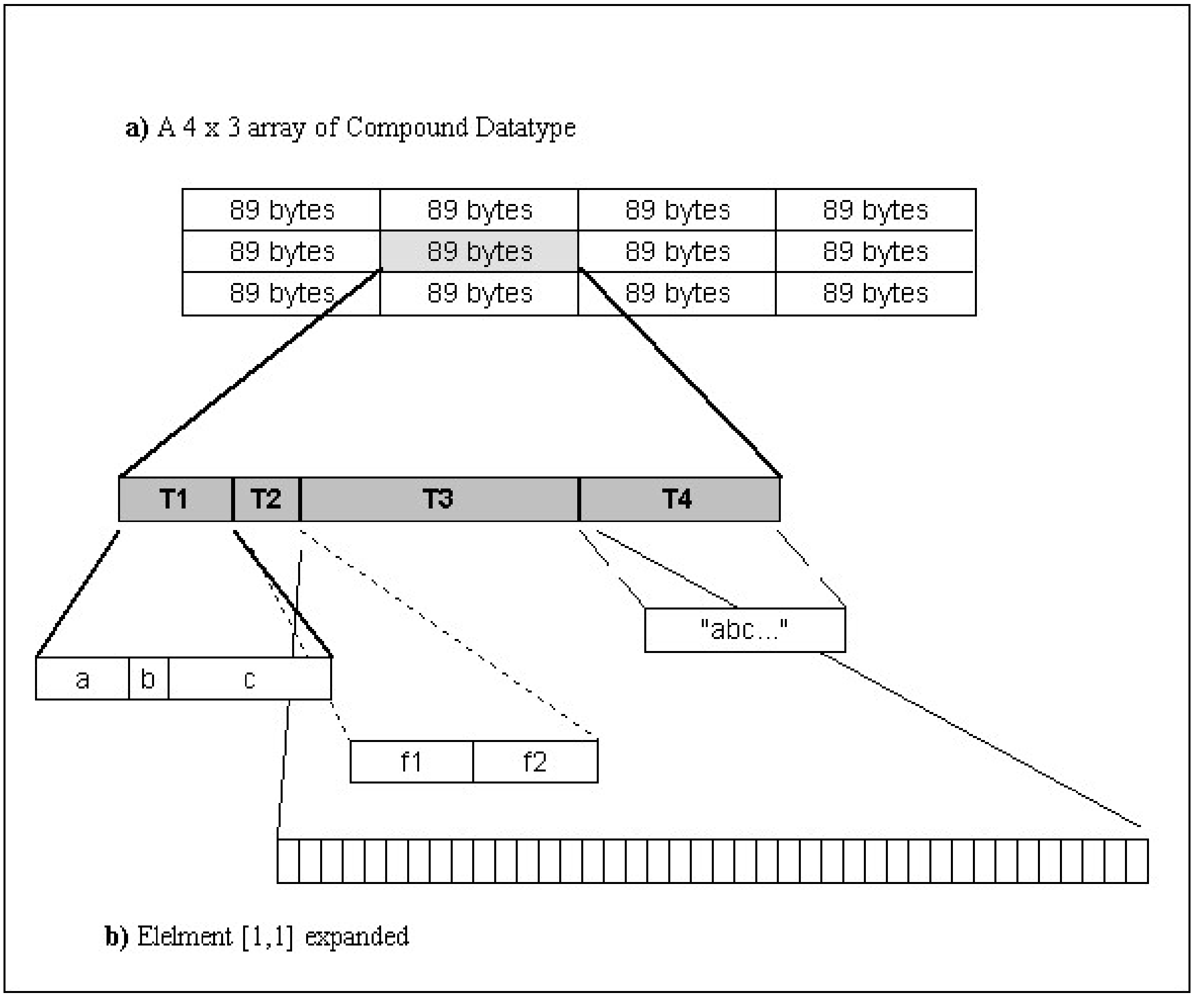

a) A 4 x 3 array of Compound Datatype
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
89 bytes
T1
T2
T3
T4
a
b
c
"abc..."
f1
f2
b) Elelment [1,1] expanded

Figure 51

7.3 Analyzing and Navigating a Compound Datatype

A complicated compound datatype can be analyzed piece by piece, to discover the exact storage layout. In the example above, the outer datatype is analyzed to discover that it is a compound datatype with 4 members. Each member is analyzed in turn to construct a complete map of the storage layout.

Figure 52 shows an example of code that partially analyses a nested compound datatype. The name and overall offset and size of the component datatype is discovered, and then it's type is analyzed, depending on the datatype class. Through this method, the complete storage layout can be discovered.

```
     s1_tid = H5Dget_type(dataset);

     if (H5Tget_class(s1_tid) == H5T_COMPOUND) {
         printf("COMPOUND DATATYPE {\n");
         sz = H5Tget_size(s1_tid);
         nmemb = H5Tget_nmembers(s1_tid);
         printf("  %d bytes\n",sz);
         printf("  %d members\n",nmemb);
         for (i =0; i < nmemb; i++) {
                s2_tid = H5Tget_member_type(s1_tid,i);
                if (H5Tget_class(s2_tid) == H5T_COMPOUND) {
                        /* recursively analyze the nested type. */

                } else if (H5Tget_class(s2_tid) == H5T_ARRAY) {
                        sz2 = H5Tget_size(s2_tid);
                        printf("  %s: NESTED ARRAY DATATYPE offset %d
size %d  {\n",
                   H5Tget_member_name(s1_tid,i),
                   H5Tget_member_offset(s1_tid,i),
                    sz2);
                   H5Tget_array_dims(s2_tid,dim,NULL);
                    s3_tid = H5Tget_super(s2_tid);
                   /* Etc., analyze the base type of the array */
                } else {
                   /* analyze a simple type */
                   printf("    %s: type code %d offset %d size %d\n",
                                 H5Tget_member_name(s1_tid,i),
                                 H5Tget_class(s2_tid),
                                 H5Tget_member_offset(s1_tid,i),
                                 H5Tget_size(s2_tid));
                }
           /* and so on . */
```

Figure 52

8. Life Cycle of the Datatype Object

Applications programs access HDF5 datatypes through handles, which are obtained by creating a new datatype, or copying or opening an existing datatype. The handle can be used until it is closed, or the program exits (Figure 53a,b). By default, a datatype object is *transient*, and disappears when it is closed.

When a dataset or attribute is created (`H5Dcreate` or `H5Acreate`), its datatype object is stored in the HDF5 file as part of the HDF5 object (the dataset or attribute) (Figure 53c). Once an object created, its datatype cannot be changed or deleted. The datatype can be accessed by calling `H5Dget_type`, `H5Aget_type`, `H5Tget_super`, or `H5Tget_member_type` (Figure 53d). These calls return a handle to a *transient* copy of the datatype of the dataset or attribute unless the datatype is a named datatype as explained below.

Note that when an object is created, the stored datatype is a copy of the transient datatype. If two objects are created with the same datatype, the information is stored in each object, with the same effect as if two different datatypes were created and used.

A transient datatype can be stored (`H5Tcommit`) in the HDF5 file as an independent, named object, called a named datatype (Figure 53e). Subsequently, when a named datatype is opened with `H5Topen` (Figure 53f), or is obtained with `H5Tget_type` or similar call (Figure 53k), the return is a handle to a transient copy of the stored datatype. The handle can be used in the same way as other datatype handles, except the named datatype cannot be modified. When a named datatype is copied with `H5Tcopy`, the return is a new, modifiable, transient datatype object (Figure 53f).

When an object is created using a named datatype (`H5Dcreate`, `H5Acreate`), the stored datatype is used without copying it to the object (Figure 53j). In this case, if multiple objects are created using the same named datatype, they all share the exact same datatype object. This saves space and makes clear that the datatype is shared. Note that a named datatype can be shared by objects within the same HDF5 file, but not by objects in other files.

A named datatype can be deleted from the file by calling `H5Gunlink` (Figure 53i). If one or more objects are still using the datatype, the named datatype cannot be accessed with H5Topen, but will not be removed from the file until it is no longer used. The `H5Tget_type` and similar calls will return a transient copy of the datatype.

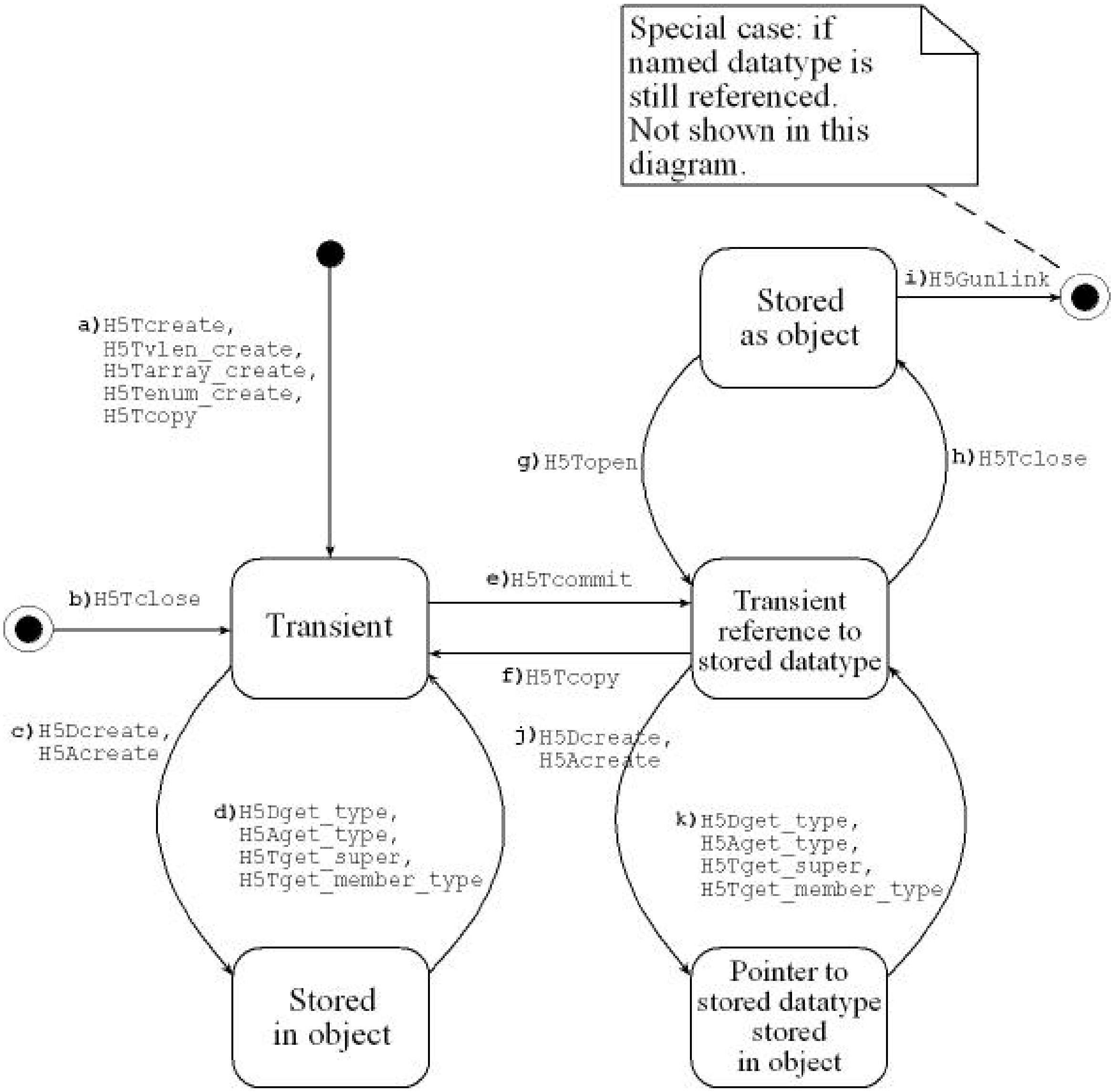

Special case: if named datatype is still referenced. Not shown in this diagram.
Stored as object
i) H5Gunlink
a) H5Tcreate, H5Tvlen_create, H5Tarray_create, H5Tenum_create, H5Tcopy
g) H5Topen
h) H5Tclose
b) H5Tclose
Transient
e) H5Tcommit
Transient reference to stored datatype
f) H5Tcopy
c) H5Dcreate, H5Acreate
j) H5Dcreate, H5Acreate
d) H5Dget_type, H5Aget_type, H5Tget_super, H5Tget_member_type
k) H5Dget_type, H5Aget_type, H5Tget_super, H5Tget_member_type
Stored in object
Pointer to stored datatype stored in object

Figure 53

Transient datatypes are initially *modifiable*, its properties can be changed. Note that when a datatype is copied or when it is written to the file (when an object is created) or the datatype is used to create a composite datatype, a copy of the current state of the datatype is used. If the datatype is then modified, the changes have no effect on datasets, attributes, or datatypes that have already been created.

A transient datatype can be made *read-only* (`H5Tlock`), after which it can no longer be changed. Note that the datatype is still transient, and otherwise does not change. A datatype that is *immutable* is *read-only* but cannot be closed except when the entire library is closed. The predefined types such as `H5T_NATIVE_INT` are *immutable transient* types.

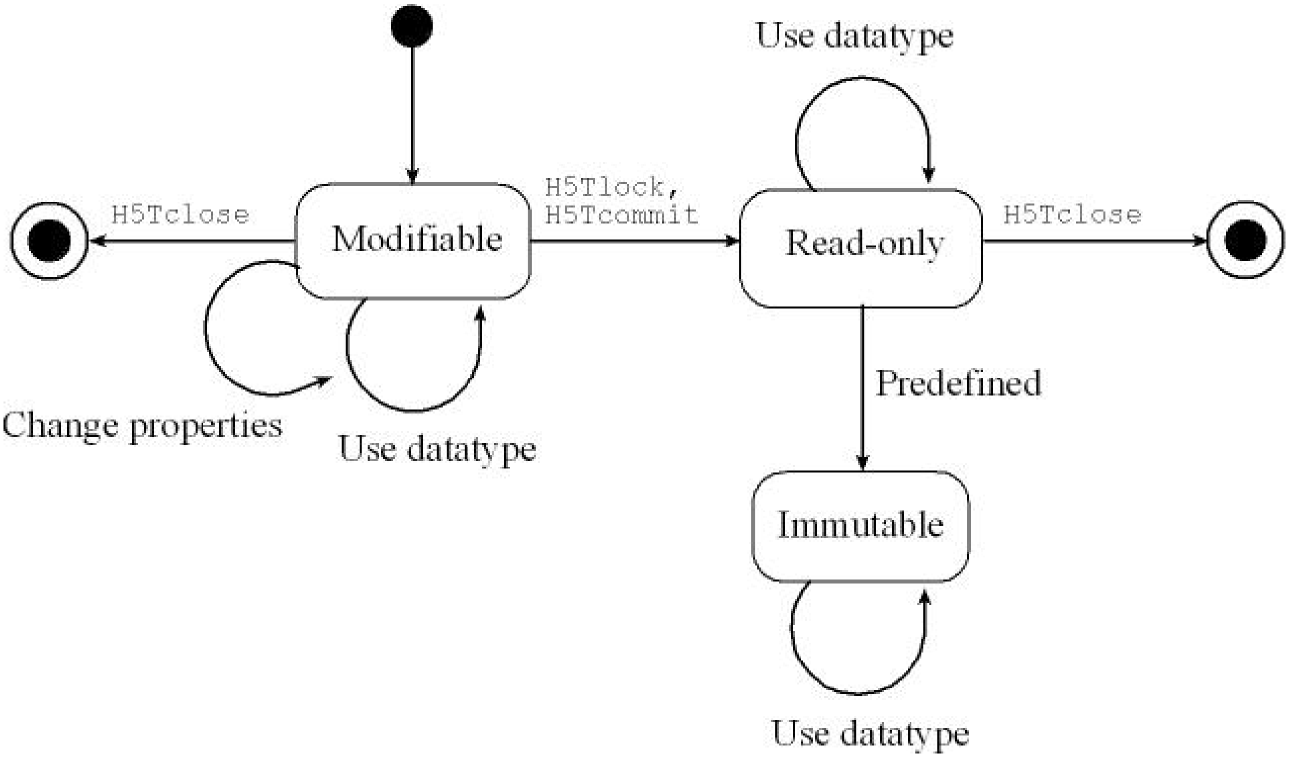


Figure 54

To create two or more datasets that share a common datatype, one first commits the datatype, giving it a name, 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);


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);
```

Figure 55

Table 24

Function	Description
`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.

9. Data Transfer: Datatype Conversion and Selection

When data is transferred (write or read) the storage layout of the data elements may be different. For example, an integer might be stored on disk in big endian byte order, and read into memory with little endian byte order. In this case, each data element will be transformed by the HDF5 library during the data transfer.

The conversion of data elements is controlled by specifying datatype of the source and specifying the intended datatype of the destination. The storage format on disk is the datatype specified when the dataset is create. The datatype of memory must be specified in the library call.

In order to be convertible, the datatype of the source and destination must have the same datatype class. Thus, integers can be converted to other integers, and floats to other floats, but integers cannot (yet) be converted to floats. For each atomic datatype class, the possible conversions are defined.

Basically, any datatype can be converted to another datatype of the same datatype class. The HDF5 library automatically converts all properties. If the destination is too small to hold the source value then an overflow or underflow exception occurs. If a handler is defined, with `H5Tset_overflow()`, it will be called. Otherwise, a default action will be performed. Table 25 summarizes the default action.

Table 25

Datatype Class	Possible Exceptions	Default Action
Integer	size, offset, pad	
Float	size, offset, pad, ebits, etc.	
String	size	Truncates, zero terminate if required.
Enumeration	No field	All Bits set

When data is transferred (write or read) the format of the data elements may be transformed between the source and the destination, according to the datatypes of the source and destination.

In order to be convertible, the datatype of the source and destination must have the same datatype class.

For example, when reading data from a dataset, the source datatype is the datatype set when the dataset was created, and the destination datatype is the description of the storage layout in memory, which must be specified in the *H5Dread* call. Figure 56 shows an example of reading a dataset of 32-bit integers. Figure 57 shows the data transformation that is performed.

```
/* Stored as H5T_STD_BE32 */
/* Use the native memory order in the destination */
mem_space = H5Tcopy(H5T_NATIVE_INT);
status = H5Dread(dataset_id, mem_type_id, mem_space_id,
                file_space_id,  xfer_plist_id,  buf );
```

Figure 56

Source Datatype: H5T_STD_BE32

Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	bbbbbbbb	cccccccc	dddddddd
Byte 4	Byte 5	Byte 6	Byte 7
wwwwwwww	xxxxxxxx	yyyyyyyy	zzzzzzzz

....

⇩ Automatically byte swapped during the H5Dread

Destination Datatype: H5T_STD_LE32

Byte 0	Byte 1	Byte 2	Byte 3
bbbbbbbb	aaaaaaaa	dddddddd	cccccccc
Byte 4	Byte 5	Byte 6	Byte 7
xxxxxxxx	wwwwwwww	zzzzzzzz	yyyyyyyy

....

Figure 57

One thing to note in Figure 56 is the use of the predefined native datatype, H5T_NATIVE_INT. Recall that in this example, the data was stored as a 4-bytes in big endian order. The application wants to read this data into an array of integers in memory. Depending on the system, the storage layout of memory might be either big or little endian, so the data may need to be transformed on some platforms and not on others. The H5T_NATIVE_INT type is set by the HDF5 library to be the correct type to describe the storage layout of the memory on the system. Thus, the code in Figure 56 will work correctly on any platform, performing a transformation when needed.

There are predefined native types for most atomic datatypes, which can be combined in composite datatypes. In general, the predefined native datatypes should always be used for data stored in memory.

Predefined native datatypes describe
the storage properties of memory.

For composite datatypes, the component atomic datatypes will be converted. For a variable length datatype, the source and destination must have compatible base datatypes. For a fixed-size string datatype, the length and padding of the strings will be converted. Variable length strings are converted as variable length datatypes.

For an array datatype, the source and destination must have the same rank and dimensions, and the base datatype must be compatible. For example an array datatype of 4 x 3 32-bit big endian integers can be transferred to an array datatype of 4 x 3 little endian integers, but not to a 3 x 4 array.

For an enumeration datatype, data elements are converted by matching the symbol names of the source and destination Datatype. Figure 58 shows an example of how two enumerations with the same names and different values would be converted. The value '2' in the source dataset would be converted to '0x0004' in the destination.

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.

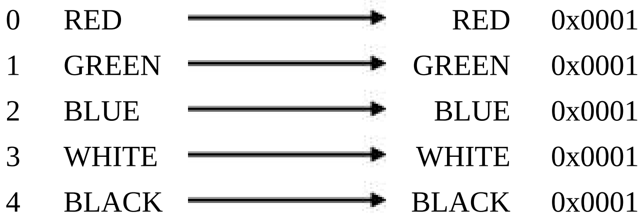


Figure 58

For compound datatypes, each field of the source and destination datatype is converted according to its type. The name and order of the fields must be the same in the source and the destination but the source and destination may have different alignments of the fields, and only some of the fields might be transferred.

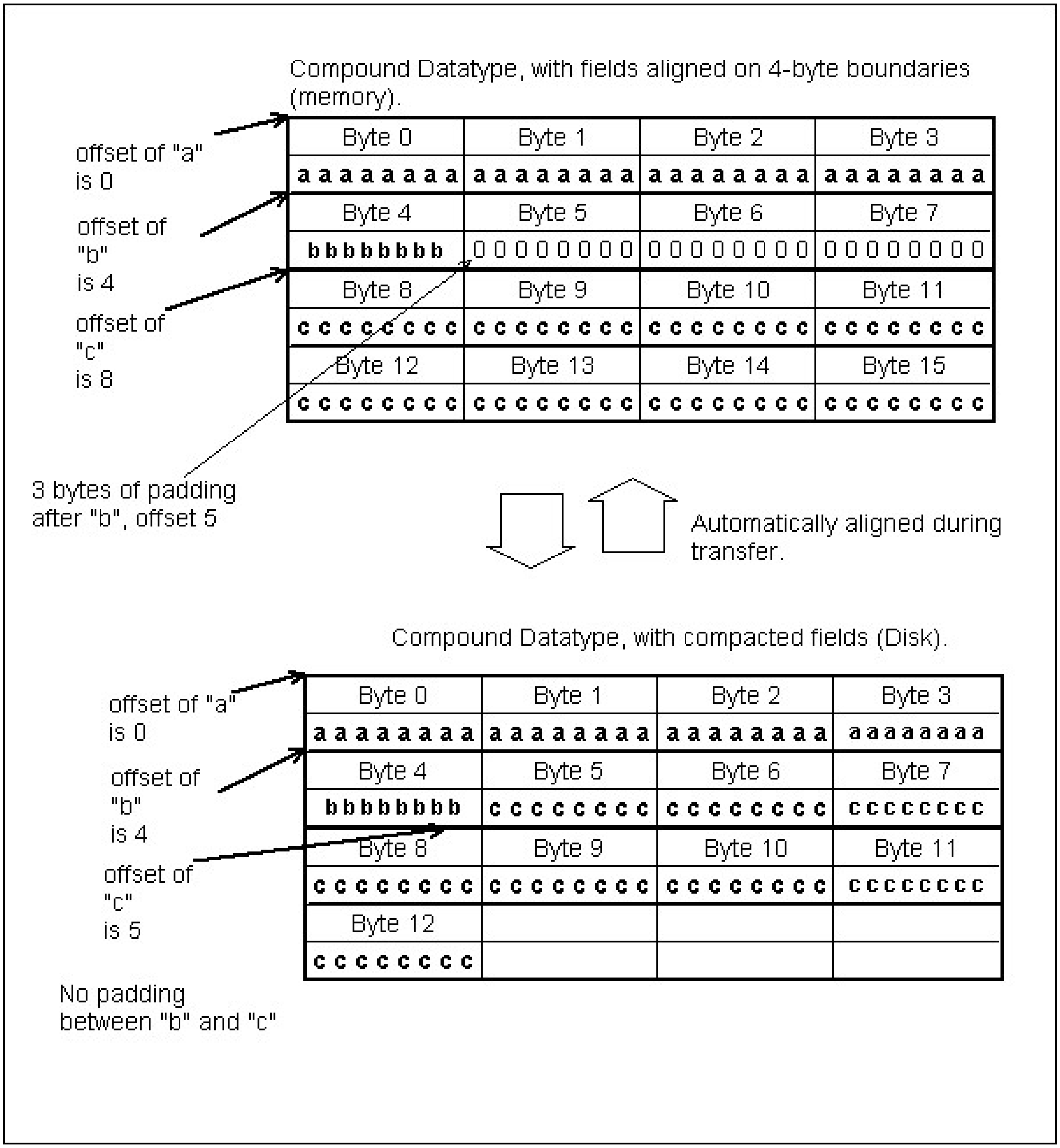
Figure 59 shows the compound datatypes shows sample code to create a compound datatype with the fields aligned on word boundaries (s1_tid) and with the fields packed (s2_tid). The former is suitable as a description of the storage layout in memory, the latter would give a more compact store on disk. These types can be used for transferring data, with `s2_tid` used to create the dataset, and `s1_tid` used as the memory datatype.

```
 typedef struct s1_t {
   int    a;
   char   b;
   double c;
} s1_t;

      s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_CHAR);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);

s2_tid = H5Tcopy(s1_tid);
H5Tpack(s2_tid);
```

Figure 59

When the data is transferred, the fields within each data element will be aligned according to the datatype specification. Figure 60 shows how one data element would be aligned in memory and on disk. Note that the size and byte order of the elements might also be converted during the transfer.

It is also possible to transfer some of the fields of a compound datatypes. Continuing the example, from Figure 59, Figure 61 shows a compound datatype that selects the first and third fields of the `s1_tid`. The second datatype can be used as the memory datatype, in which case data is read from or written to these two fields, while skipping the middle field. Figure 62 shows the data for two data elements.


Compound Datatype, with fields aligned on 4-byte boundaries (memory).
offset of "a" is 0
offset of "b" is 4
offset of "c" is 8
3 bytes of padding after "b", offset 5
Automatically aligned during transfer.
Compound Datatype, with compacted fields (Disk).
offset of "a" is 0
offset of "b" is 4
offset of "c" is 5
No padding between "b" and "c"

Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
Byte 4	Byte 5	Byte 6	Byte 7
bbbbbbbb	00000000	00000000	00000000
Byte 8	Byte 9	Byte 10	Byte 11
cccccccc	cccccccc	cccccccc	cccccccc
Byte 12	Byte 13	Byte 14	Byte 15
cccccccc	cccccccc	cccccccc	cccccccc

Byte 0	Byte 1	Byte 2	Byte 3
aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
Byte 4	Byte 5	Byte 6	Byte 7
bbbbbbbb	cccccccc	cccccccc	cccccccc
Byte 8	Byte 9	Byte 10	Byte 11
cccccccc	cccccccc	cccccccc	cccccccc
Byte 12			
cccccccc			

Figure 60

```
 typedef struct s1_t {
   int    a;
   char  b;
   double c;
} s1_t;

typedef struct s2_t {   /* two fields from s1_t */
   int    a;
   double c;
} s2_t;

      s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_CHAR);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);

s2_tid = H5Tcreate (H5T_COMPOUND, sizeof(s2_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s2_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "c_name", HOFFSET(s2_t, c), H5T_NATIVE_DOUBLE);
```

Figure 61

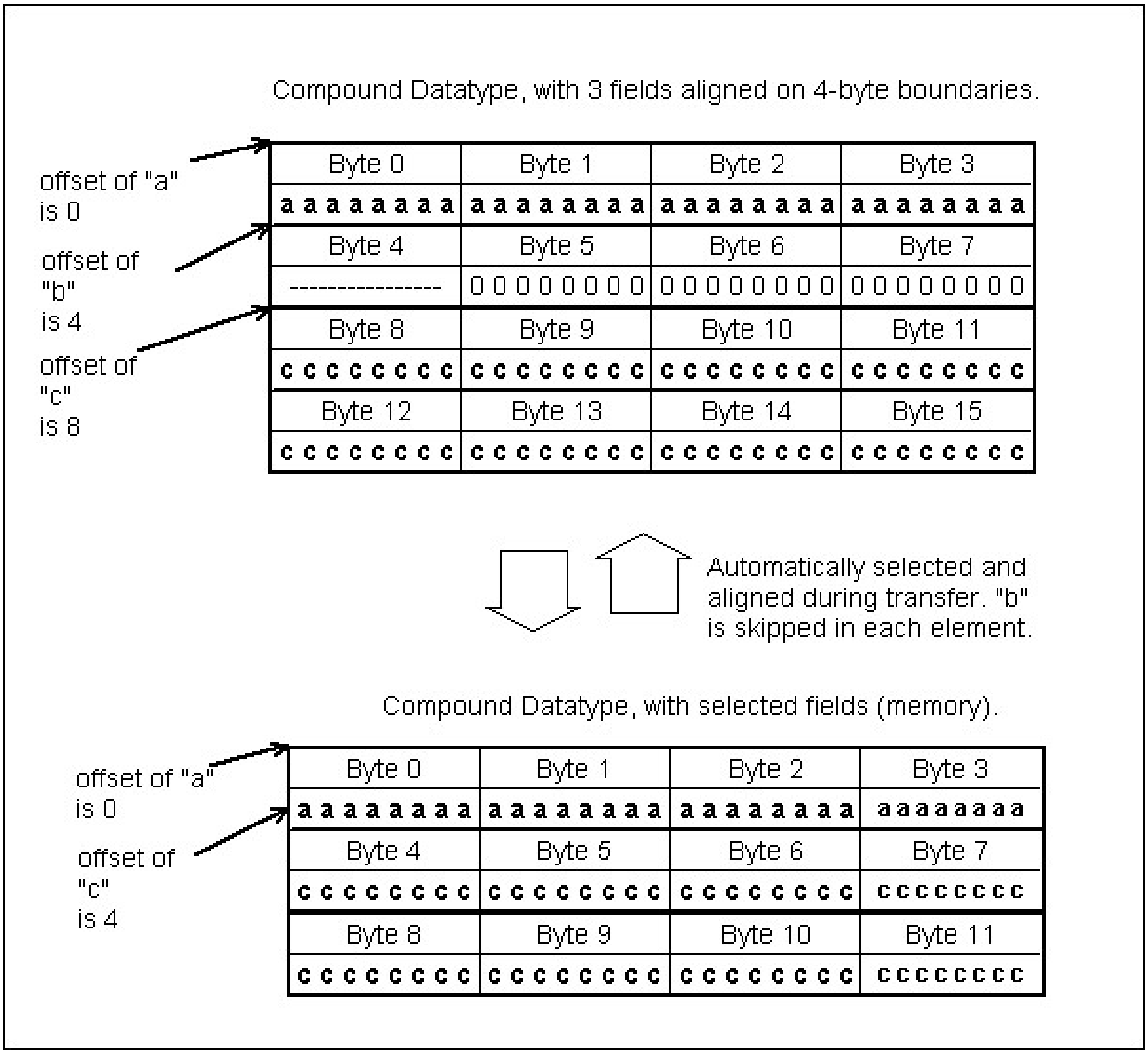


Figure 62

Chapter 6
Dataspaces and Partial I/O

1. Introduction

The HDF5 *dataspace* is a required component of an HDF5 dataset or attribute definition. The dataspace defines the size and shape of the dataset or attribute raw data, i.e., the number of dimensions and the size of each dimension of the multideminesional array in which the raw data is represented. The dataspace must be defined when the dataset or attribute is created.

The *dataspace* is also used during dataset I/O operations, defining the elements of the dataset that participate in the I/O operation.

This chapter explains the *dataspace* object and its use in dataset and attribute creation and data transfer. It also describes selection operations on a dataspace used to implement subsetting, subsampling, and scatter-gather access to datasets.

The rest of this chapter is structured as follows:

- Section 2, "Dataspace Functions," provides a categorized list of dataspace functions, also known as the H5S APIs.
- Section 3,"Definition of Dataspace Objects and the Dataspace Programming Model," describes dataspace objects and the programming model, including the creation and use of dataspaces.
- Section 4, "Dataspaces and Data Transfer," describes the use of dataspaces in data transfer.
- Section 5, "Dataspace Selection Operations and Data Transfer," describes selection operations on dataspaces and their usage in data transfer.
- Section 6, "References to Dataset Regions," briefly discusses references to dataset regions.
- Section 7, "Sample Programs," contains the full programs from which several of the code samples in this chapter were derived.

2. Dataspace Function Summaries

This section provides a reference list of dataspace functions, the H5S APIs, with brief descriptions. The functions are presented in several functional catagories:

- ♦ dataspace management functions
- ♦ dataspace query functions
- ♦ dataspace selection functions
    - ◊ hyperslab selections
    - ◊ point selections

Sections 3 through 6 will provide examples and explanations of how to use these functions.

Dataspace management functions

C Function F90 Function	Purpose
`H5Screate` `h5screate_f`	Creates a new dataspace of a specified type.
`H5Scopy` `h5scopy_f`	Creates an exact copy of a dataspace.
`H5Sclose` `h5sclose_f`	Releases and terminates access to a dataspace.
`H5Screate_simple` `h5screate_simple_f`	Creates a new simple dataspace and opens it for access.
`H5Sis_simple` `h5sis_simple_f`	Determines whether a dataspace is a simple dataspace.
`H5Sextent_copy` `h5sextent_copy_f`	Copies the extent of a dataspace.
`H5Sset_extent_simple` `h5sset_extent_simple_f`	Sets or resets the size of an existing dataspace.
`H5Sset_extent_none` `h5sset_extent_none_f`	Removes the extent from a dataspace.

Dataspace query functions

C Function F90 Function	Purpose
`H5Sget_simple_extent_dims` `h5sget_simple_extent_dims_f`	Retrieves dataspace dimension size and maximum size.
`H5Sget_simple_extent_ndims` `h5sget_simple_extent_ndims_f`	Determines the dimensionality of a dataspace.
`H5Sget_simple_extent_npoints` `h5sget_simple_extent_npoints_f`	Determines the number of elements in a dataspace.
`H5Sget_simple_extent_type` `h5sget_simple_extent_type_f`	Determine the current class of a dataspace.

Dataspace selection functions: Hyperslabs

C Function F90 Function	Purpose
`H5Soffset_simple` `h5soffset_simple_f`	Sets the offset of a simple dataspace.
`H5Sget_select_type` `h5sget_select_type_f`	Determines the type of the dataspace selection.
`H5Sget_select_hyper_nblocks` `h5sget_select_hyper_nblocks_f`	Get number of hyperslab blocks.
`H5Sget_select_hyper_blocklist` `h5sget_select_hyper_blocklist_f`	Gets the list of hyperslab blocks currently selected.
`H5Sget_select_bounds` `h5sget_select_bounds_f`	Gets the bounding box containing the current selection.
`H5Sselect_all` `h5sselect_all_f`	Selects the entire dataspace.
`H5Sselect_none` `h5sselect_none_f`	Resets the selection region to include no elements.
`H5Sselect_valid` `h5sselect_valid_f`	Verifies that the selection is within the extent of the dataspace.
`H5Sselect_hyperslab` `h5sselect_hyperslab_f`	Selects a hyperslab region to add to the current selected region.

Dataspace selection functions: Points

C Function F90 Function	Purpose
`H5Sget_select_npoints` `h5sget_select_npoints_f`	Determines the number of elements in a dataspace selection.
`H5Sget_select_elem_npoints` `h5sget_select_elem_npoints_f`	Gets the number of element points in the current selection.
`H5Sget_select_elem_pointlist` `h5sget_select_elem_pointlist_f`	Gets the list of element points currently selected.
`H5Sselect_elements` `h5sselect_elements_f`	Selects array elements to be included in the selection for a dataspace.

3. Definition of Dataspace Objects and the Dataspace Programming Model

This section introduces the notion of the HDF5 dataspace object and a programming model for creating and working with dataspaces.

Dataspace Objects

An HDF5 dataspace is a required component of an HDF5 dataset or attribute. A dataspace defines the size and the shape of a dataset's or an attribute s raw data. Currently HDF5 supports two types of the dataspaces:

- scalar dataspaces
- simple dataspaces

A *scalar dataspace* represents just one element, a scalar. Note that the datatype of this one element may be very complex, e.g., a compound structure with members being of any allowed HDF5 datatype, including multidimensional arrays, strings, and nested compound structures.

A *simple dataspace* is a multidimensional array of elements. The dimensionality of the dataspace (or the rank of the array) is fixed and is defined at the creation time. The size of each dimension can grow during the life time of the dataspace from the *current size* up to the *maximum size*. Both the current size and the maximum size are specified at the creation time. The sizes of dimensions at any particular time of the datatype life are called the *current dimensions* or the *dataspace extent*. They can be queried along with the maximum sizes.

As shown in the UML diagram in Figure 1, an HDF5 simple dataspace object has three attributes: the rank or number of dimensions; the current sizes, expressed as an array of length rank with each element of the array denoting the current size of the corresponding dimension; and the maximum sizes, expressed as an array of length rank with each element of the array denoting the maximum size of the corresponding dimension.

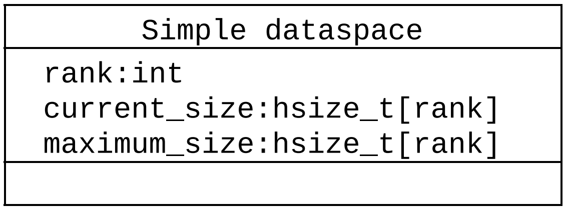


Figure 1: A simple dataspace is defined by its rank, the current size of each dimension, and the maximum size of each dimension.

The size of a current dimension cannot be greater than the maximum size, which can be unlimited, specified as `H5S_UNLIMITED`. Note that while the HDF5 file format and library impose no maximum size on an unlimited dimension, practically speaking its size will always be limited to the biggest integer available on the particular system being used.

Dataspace rank is restricted to 32, the standard limit in C on the rank of an array, in the current implementation of the HDF5 library. The HDF5 file format, on the other hand, allows any rank up to the maximum integer value on the system, so the library restriction can be raised in the future if higher dimensionality is required. (Note that most of the time Fortran applications calling HDF5 will work with dataspaces of rank less than or equal to seven, since seven is the maximum number of dimensions in a

Fortran array. But dataspace rank is not limited to seven for Fortran applications.

The current dimensions of a dataspace, also referred to as the dataspace extent, define the bounding box for dataset elements that can participate in I/O operations.

Programming Model

The programming model for creating and working with HDF5 dataspaces can be summarized as follows:

1. Create a dataspace.
2. Use the dataspace to create a dataset in the file or to describe a data array in memory.
3. Modify the dataspace to define dataset elements that will participate in I/O operations.
4. Use the modified dataspace while reading/writing dataset raw data or to create a region reference.
5. Close the dataspace when no longer needed.

The rest of this section will address steps 1, 2, and 5 of the programming model; steps 3 and 4 will be discussed in later sections of this chapter.

Creating a dataspace

Both scalar and simple dataspace can be created by calling the H5Screate (h5screate_f in Fortran) function. Since definition of the simple dataspace requires the specification of dimensionality (or rank) and initial and maximum dimension sizes, the HDF5 Library provides a *convenience* API, H5Screate_simple (h5screate_simple_f) to create a simple dataspace in on step. Examples below illustrate the usage of these APIs.

Creating a scalar dataspace

A scalar dataspace is created with the `H5Screate` or the `h5screate_f` function:

In C:

```
hid_t space_id;
. . .
space_id = H5Screate(H5S_SCALAR);
```

In Fortran:

```
INTEGER(HID_T) :: space_id
. . .
CALL h5screate_f(H5S_SCALAR_F, space_id, error)
```

As mentioned above, the dataspace will contain only one element. Scalar dataspaces are used more often for describing attributes that have just one value, e.g. the attribute Temperature with the value Celsius is used to indicate that the dataset with this attribute stores temperature values using the Celsius scale.

Creating a simple dataspace

Let s assume that an application wants to store a two-dimensional array of data A(20,100). During the life of the application the first dimension of the array can grow up to 30, and there is no restriction on the size of the second dimension. The following steps are used to declare a dataspace for the dataset in which the array data will be stored.

In C:

```
hid_t space_id;
int rank = 2;
hsize_t current_dims[2] = {10, 100};
hsize_t max_dims[2] = {H5S_UNLIMITED, 200};
. . .
space_id = H5Screate(H5S_SIMPLE);
H5Sset_extent_simple(space_id,rank,current_dims,max_dims);
```

In Fortran:

```
INTEGER(HID_T) :: space_id
INTEGER :: rank = 2
INTEGER(HSIZE_T) :: current dims = /( 10, 100)/
INTEGER(HSIZE_T) :: max_dims = /(H5S_UNLIMITED_F, 200)/
INTEGER error
. . .
CALL h5screate_f(H5S_SIMPLE_F, space_id, error)
CALL h5sset_extent_simple_f(space_id, rank, current_dims, max_dims, error)
```

Alternatively, the convenience APIs H5Screate_simple/h5screate_simple_f can replace the H5Screate/h5screate_f and H5Sset_extent_simple/h5sset_extent_simple_f calls.

In C:

```
space_id = H5Screate_simple( rank, current_dims, max_dims);
```

In Fortran:

```
CALL h5screate_simple_f(space_id, rank, current_dims, error, max_dims)
```

In this example (see Figure 2), a dataspace with current dimensions of 20 by 100 is created. The first dimension can be extended only up to 30. The second dimension, however, is declared unlimited; it can be extended up to the largest available integer value on the system. Recall that any dimension can be declared unlimited, and if a dataset uses a dataspace with any unlimited dimension, chunking has to be used (see section ??? in the "Datasets" chapter).

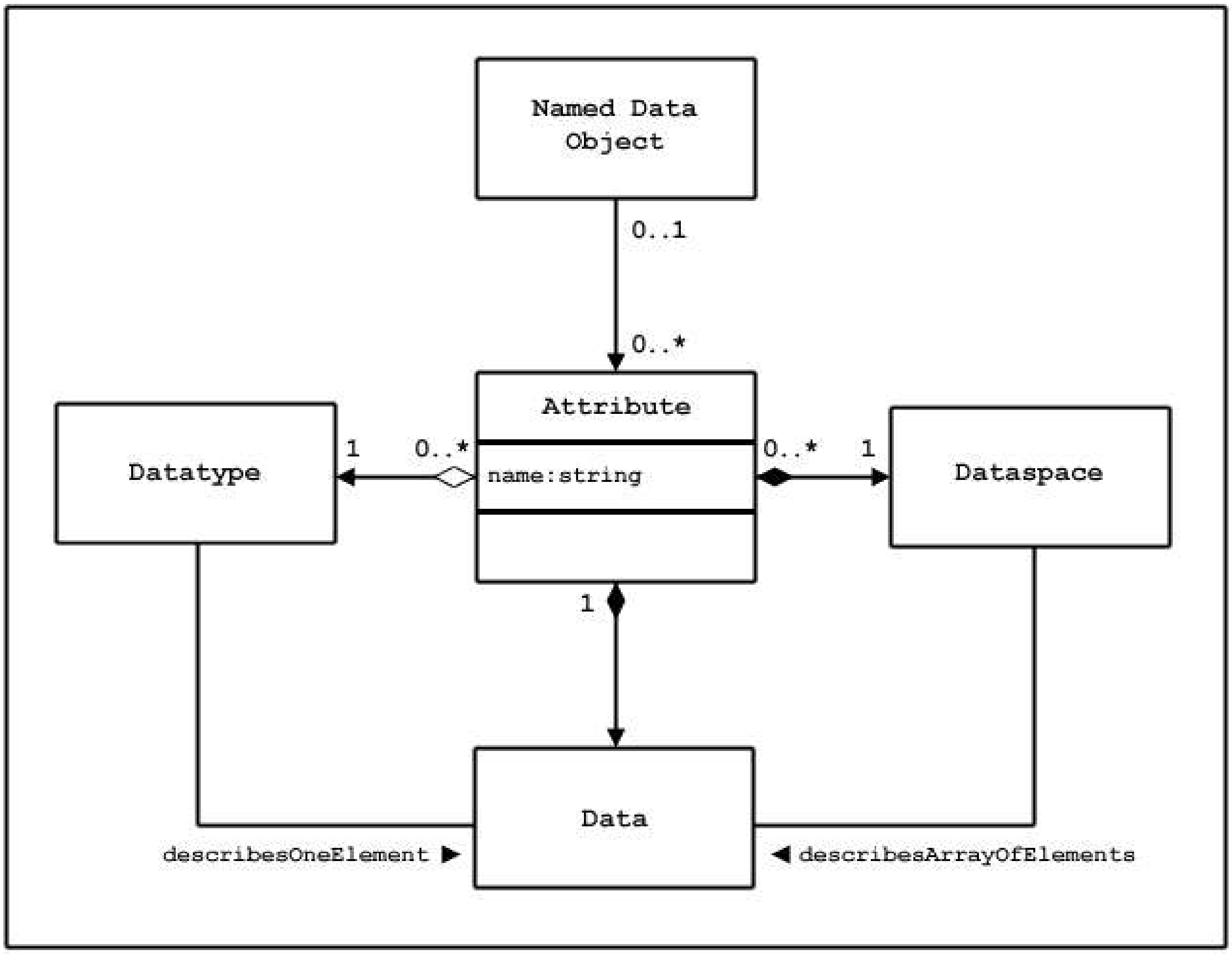


Figure 2: Create a simple dataspace with a 10x100 array

Maximum dimensions can be the same as current dimensions. In such a case, the sizes of dimensions cannot be changed during the life of the dataspace object. In C, NULL can be used to indicate to the H5Screate_simple and H5Sset_extent_simple functions that the maximum sizes of all dimensions are the same as the current sizes. In Fortran, the maximum size parameter is optional for h5screate_simple_f and can be omitted when the sizes are the same.

In C:

```
space_id = H5Screate_simple(rank, current_dims, NULL);
```

In Fortran:

```
 CALL h5screate_f(space_id, rank, current_dims, error)
```

The created dataspace will have current and maximum dimensions of 20 and 100 correspondingly, and the sizes of those dimensions cannot be changed.

C versus Fortran Dataspaces

Dataspace dimensions are numbered from 1 to rank. HDF5 uses C storage conventions, assuming that the last listed dimension is the fastest-changing dimension and the first-listed dimension is the slowest changing. The HDF5 file format storage layout specification adheres to the C convention and the HDF5 Library adheres to the same convention when storing dataspace dimensions in the file. This affects how C programs and tools interpret data written from Fortran programs and vice versa. The example below illustrates the issue.

When a Fortran application describes a dataspace to store an array as A(20,100), it specifies the value of the first dimension to be 20 and the second to be 100. Since Fortran stores data by columns, the first-listed dimension with the value 20 is the fastest-changing dimension and the last-listed dimension with the value 100 is the slowest-changing. In order to adhere to the HDF5 storage convention, the HDF5 Fortran wrapper transposes dimensions, so the first dimension becomes the last. The dataspace dimensions stored in the file will be 100,20 instead of 20,100 in order to correctly describe the fortran data that is stored in 100 columns, each containing 20 elements.

When a Fortran application reads the data back, the HDF5 Fortran wrapper transposes the dimensions once more, returning the first dimension to be 20 and the second to be 100, describing correctly the sizes of the array that should be used to read data in the Fortran array A(20,100).

When a C application reads data back, the dimensions will come out as 100 and 20, correctly describing the size of the array to read data into, since the data was written as 100 records of 20 elements each. Therefore C tools such as h5dump and h5ls always display transposed dimensions and values for the data written by a Fortran application.

Consider the following simple example of equivalent C 3x5 and Fortran 5x3 arrays. As illustrated in Figure 3, a C applications will store a 3x5 2-dimensional array as three 5-element rows. In order to store the same data in the same order, a Fortran application must view the array as as a 5x3 array with three 5-element columns. The dataspace of this dataset, as written from Fortran, will therefore be described as 5x3 in the application but stored and described in the file according to the C convention as a 3x5 array. This ensures that C and Fortran applications will always read the data in the order in which it was written. The HDF5 Fortran interface handles this transposition automatically.

In C (from `h5_write.c`):

```
#define NX     3                      /* dataset dimensions */
#define NY     5
. . .
int         data[NX][NY];          /* data to write */
. . .
/*
 * Data  and output buffer initialization.
 */
for (j = 0; j <NX; j++) {
   for (i = 0; i  <NY; i++)
       data[j][i] = i + 1 + j*NY;
}
/*
 *  1  2  3  4  5
 *  6  7  8  9 10
 * 11 12 13 14 15
 */
. . .
dims[0] = NX;
dims[1] = NY;
dataspace = H5Screate_simple(RANK, dims, NULL);
```

In Fortran (from `h5_write.f90`):

```
INTEGER, PARAMETER :: NX = 3
INTEGER, PARAMETER :: NY = 5
. . .
INTEGER(HSIZE_T), DIMENSION(2) :: dims = (/3,5/) ! Dataset dimensions
---
INTEGER     ::    data(NX,NY)
. . .
!
! Initialize data
!
  do i = 1, NX
     do j = 1, NY
        data(i,j) = j + (i-1)*NY
     enddo
  enddo
!
! Data
!
!  1  2  3  4  5
!  6  7  8  9 10
! 11 12 13 14 15
. . .
CALL h5screate_simple_f(rank, dims, dspace_id, error)
```

In Fortran (from `h5_write_tr.f90`):

```
INTEGER, PARAMETER :: NX = 3
INTEGER, PARAMETER :: NY = 5
. . .
INTEGER(HSIZE_T), DIMENSION(2) :: dims = (/NY, NX/) ! Dataset dimensions
. . .
!
! Initialize data
!
  do i = 1, NY
     do j = 1, NX
        data(i,j) = i + (j-1)*NY
     enddo
  enddo
!
! Data
!
!  1  6  11
!  2  7  12
!  3  8  13
!  4  9  14
!  5 10  15
. . .
CALL h5screate_simple_f(rank, dims, dspace_id, error)
```

A dataset stored by a
C program in a 3x5 array:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

The same dataset stored by a
Fortran program in a 5x3 array:

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

The left-hand dataset above as written to an HDF5 file from C or the right-hand dataset as written from Fortran:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The left-hand dataset above as written to an HDF5 file from Fortran:

1	6	11	2	7	12	3	8	13	4	9	14	5	10	15

Figure 3: The HDF5 library stores arrays along the fastest-changing dimension, an approach often referred to as being “in C order.” C, C++, and Java work with arrays in row-major order, i.e., the row, or the last dimension, is the fastest-changing dimension. Fortran, on the other hands, handles arrays in column-major order, making the column, or the first dimension, the fastest-changing dimension. Therefore, the C and Fortran arrays illustrated in the top portion of this figure are stored identically in an HDF5 file. This ensures that data written by any language can be meaningfully read, interpreted, and manipulated by any other.

Finding dataspace charateristics

The HDF5 Library provides several APIs designed to query the characteristics of a dataspace.

The function H5Sis_simple (h5sis_simple_f) returns information about the type of a dataspace. This function is rarely used and currently supports only simple and scalar dataspaces.

To find out the dimensionality, or rank, of a dataspace, use H5Sget_simple_extent_ndims (h5sget_simple_extent_ndims_f). H5Sget_simple_extent_dims can also be used to find out the rank. See the example below. Both functions return 0 for the value of rank the dataspace is scalar.

To query the sizes of the current and maximum dimensions, use H5Sget_simple_extent_dims (h5sget_simple_extent_dims_f).

The following example illsutrates querying the rank and dimensions of a dataspace using these functions.

In C:

```
hid_t space_id;
int rank;
hsize_t  *current_dims;
hsize_t  *max_dims;
---------

rank=H5Sget_simple_extent_ndims(space_id);
    (or rank=H5Sget_simple_extent_dims(space_id, NULL, NULL);)
current_dims= (hsize_t)malloc(rank*sizeof(hsize_t));
max_dims=(hsize_t)malloc(rank*sizeof(hsize_t));
H5Sget_simple_extent_dims(space_id, current_dims, max_dims);
Print values here for the previous example
```

4. Dataspaces and Data Transfer

The *Dataspace* object is also used to control data transfer when data is read or written. The *Dataspace* of the Dataset (Attribute) defines the stored form of the array data, the order of the elements as explained above. When reading from the file, the *Dataspace* of the Dataset defines the layout of the source data, a similar description is needed for the destination storage. A *Dataspace* object is used to define the organization of the data (rows, columns, etc.) in memory. If the program requests a different order for memory than the storage order, the data will be rearranged by the HDF5 Library during the H5Dread operation. Similarly, when writing data, the memory *Dataspace* defines the source data, which is converted to the Dataset *Dataspace* when stored by the H5Dwrite call.

Figure 4a shows a simple example of a read operation in which the data is stored as a 3 by 4 array in the file (Figure 4b), but the program wants it to be a 4 by 3 array in memory. This is accomplished by setting the memory *Dataspace* to describe the desired memory layout, as in Figure 4c. The HDF5 Library will transform the data to the correct arrangement during the read operation.

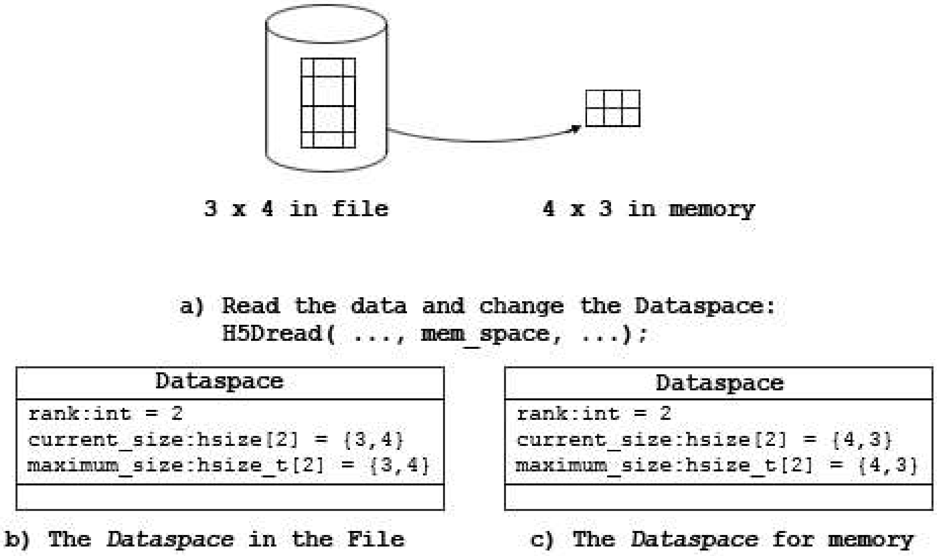

3 x 4 in file
4 x 3 in memory
a) Read the data and change the Dataspace:
H5Dread(..., mem_space, ...);
Dataspace
rank:int = 2
current_size:hsize[2] = {3,4}
maximum_size:hsize_t[2] = {3,4}
Dataspace
rank:int = 2
current_size:hsize[2] = {4,3}
maximum_size:hsize_t[2] = {4,3}
b) The Dataspace in the File
c) The Dataspace for memory

Figure 4

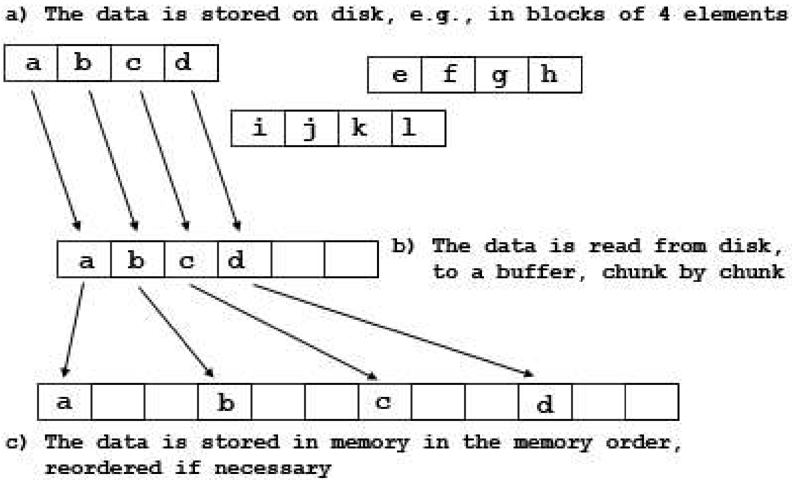

a) The data is stored on disk, e.g., in blocks of 4 elements
a
b
c
d
e
f
g
h
i
j
k
l
a
b
c
d
b) The data is read from disk, to a buffer, chunk by chunk
a
b
c
d
c) The data is stored in memory in the memory order, reordered if necessary

Figure 5

Both the source and destination are stored as contiguous blocks of storage, with the elements in the order specified by the *Dataspace*. Figure 5 shows one way the elements might be organized. In Figure 5a, the elements are stored as 3 blocks of 4 elements. The destination is an array of 12 elements in memory (Figure 5c). As the figure suggests, the transfer reads the disk blocks into a memory buffer (Figure 5b), and then writes the elements to the correct locations in memory. A similar process occurs in reverse when data is written to disk.

Data selection

In addition to rearranging data, the transfer may select the data elements from the source and destination.

Data selection is implemented by creating a *Dataspace* object that describes the selected elements (within the hyper rectangle) rather than the whole array. Two *Dataspace* objects with selections can be used in data transfers to read selected elements from the source and write selected elements to the destination. When data is transferred using the Dataspace object, only the selected elements will be transferred.

This can be used to implement partial I/O, including:

- ♦ Sub-setting - reading part of a large dataset
- ♦ Sampling - reading selected elements (e.g., every second element) of a dataset
- ♦ Scatter-gather - read non-contiguous elements into contiguous locations (gather) or read contiguous elements into non-contiguous locations (scatter) or both.

To use selections, the following steps are followed:

1. get or define the Dataspace for the source and destination.
2. specify one or more selections for source and destination Dataspaces.
3. transfer data using the Dataspaces with selections

A selection is created by applying one or more selections to a Dataspace. A selection may override any other selections (H5T_SELECT_SET) or may be 'Ored' with previous selections on the same Dataspace (H5T_SELECT_OR). In the latter case, the resulting selection is the union of the selection and all previously selected selections. Arbitrary sets of points from a Dataspace can be selected by specifying an appropriate set of selections.

Two selections are used in data transfer, so the source and destination must be compatible, as described below.

There are two forms of selection, hyperslab and point. A selection must be either a point selection or a set of hyperslab selections. Selections cannot be mixed.

Hyperslab selection

A hyperslab is a selection of elements from a hyper rectangle. An HDF5 hyperslab is a rectangular pattern defined by four arrays (Table 1). The *start* defines the origin of the hyperslab in the original Dataspace. The stride is the number of elements to increment between selected elements. A stride of '1' is every element, a stride of '2' is every second element, etc. Note that there may be a different stride for each dimension of the Dataspace. The default stride is 1.

The count is the number of elements in the hyperslab selection. When the stride is 1, the selection is a hyper rectangle with a corner at start and size count[0] by count[1] by . When stride is greater than one, the hyperslab bounded by start and the corners defined by stride[n] * count[n].

Table 1

Parameter	Description
start	Starting location for the hyperslab.
stride	The number of elements to separate each element or block to be selected.
count	The number of elements or blocks to select along each dimension.
block	The size of the block selected from the dataspace.

The *block* is a count on the number of repetitions of the hyperslab. The default block size is '1', which is one hyperslab. A block of 2 would be two hyperslabs in that dimension, with the second starting at start[n]+ (count[n] * stride[n]) + 1.

A hyperslab can be used to access a sub-set of a large Dataset. Figure 6 shows an example of a hyperslab that reads a rectangle from the middle of a larger two dimensional array. The destination is the same shape as the source.

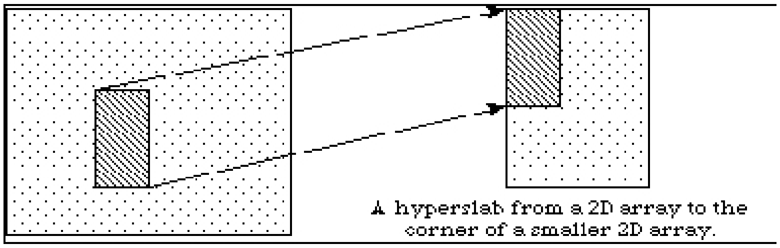

A hyperslab from a 2D array to the corner of a smaller 2D array.

Figure 6

Hyperslabs can be combined to select complex regions of the source and destination. Figure 7 shows an example of a transfer from one non-rectangular region into another non-rectangular region. The source is defined as the union of two hyperslabs, and the destination is the union of three hyperslabs.

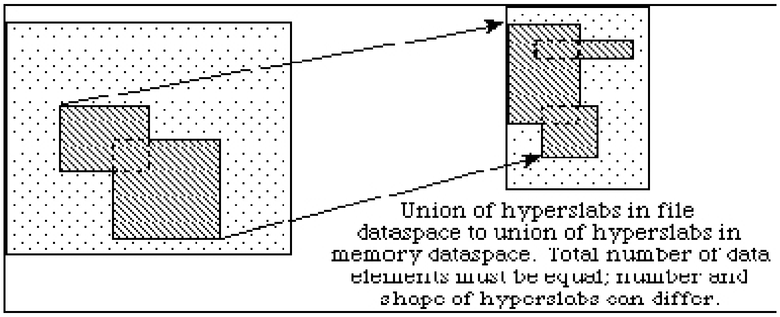

Union of hyperslabs in file dataspace to union of hyperslabs in memory dataspace. Total number of data elements must be equal; number and shape of hyperslabs can differ.

Figure 7

Hyperslabs may also be used to collect or scatter data from regular patterns. Figure 8 shows an example where the source is a repeating pattern of blocks, and the destination is a single, one dimensional array.

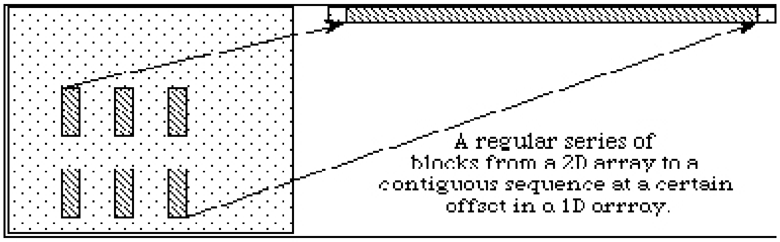


Figure 8

Select points

The second type of selection is an array of points, i.e., coordinates. Essentially, this selection is a list of all the points to include. Figure 9 shows an example of a transfer of seven elements from a two dimensional Dataspace to a three dimensional Dataspace using a point selection to specify the points.

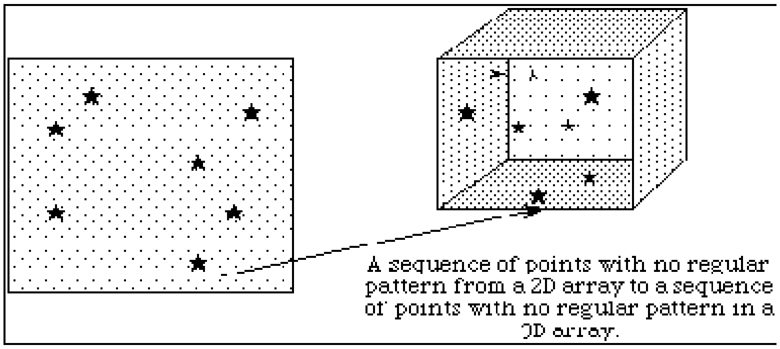

A sequence of points with no regular
pattern from a 2D array to a sequence
of points with no regular pattern in a
3D array.

Figure 9

Rules for Defining Selections

A selection must have the same number of dimensions (rank) as the Dataspace it is applied to, although it may select from only a small region, e.g., a plane from a 3D Dataspace. Selections do not affect the extent of the *Dataspace*, the selection may be larger than the *Dataspace*. The boundaries of selections are reconciled with the extent at the time of the data transfer.

Data Transfer with Selections

A data transfer (read or write) with selections is the same as any read or write, except the source and destination *Dataspace* have compatible selections.

During the data transfer, the following steps are executed by the library.

1. The source and destination *Dataspaces* are checked to assure that the selections are compatible.

    1. Each selection must be within the current extent of the *Dataspace*. A selection may be defined to extend outside the current extent of the *Dataspace*, but the *Dataspace* cannot be accessed if the selection is not valid at the time of the access.

    2. The total number of points selected in the source and destination must be the same. Note that the dimensionality of the source and destination can be different (e.g., the source could be 2D, the destination 1D or 3D), and the shape can be different, but the number of elements selected must be the same.

2. The data is transferred, element by element.

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.

The elements of the selections are transferred in row-major, or C order. That is, it is assumed that the first dimension varies slowest, the second next slowest, and so forth. For hyperslab selections, the order can be any permutation of the dimensions involved (defaulting to 'C' array order). When multiple hyperslabs are combined, the hyperslabs are coalesced into contiguous reads and writes

In the case of point selections, the points are read and written in the order specified.

Programming Model

Selecting hyperslabs

Suppose we want to read a 3x4 hyperslab from a dataset in a file beginning at the element <1,2> in the dataset, and read it into a 7x7x3 array in memory (Figure 10). 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.

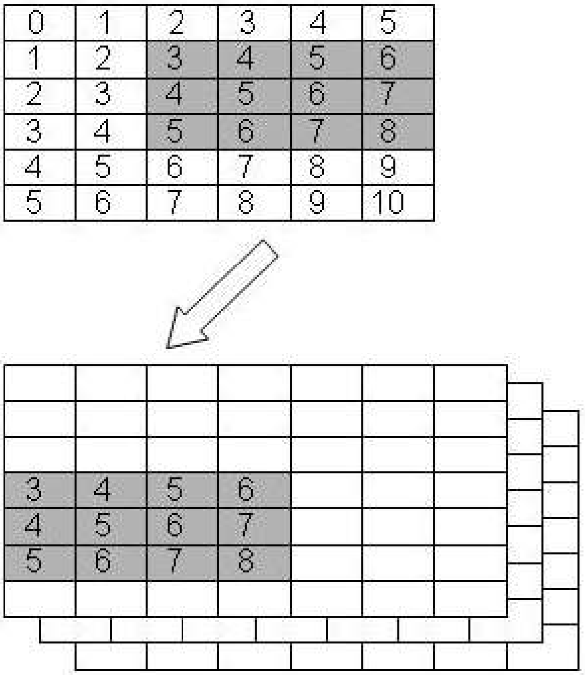


Figure 10

The code in Figure 11 illustrates the selection of the hyperslab in the file dataspace. Figure 12 shows de definition of the destination Dataspace in memory. Since the in-memory Dataspace has three dimensions, the hyperslab is an array with three dimensions, with the last dimension being 1: <3,4,1>. Figure 13 shows the read using the source and destination *Dataspaces* with selections.

```
/*
* get the file dataspace.
*/
dataspace = H5Dget_space(dataset);    /* dataspace identifier */

/*
* 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);
```

Figure 11

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

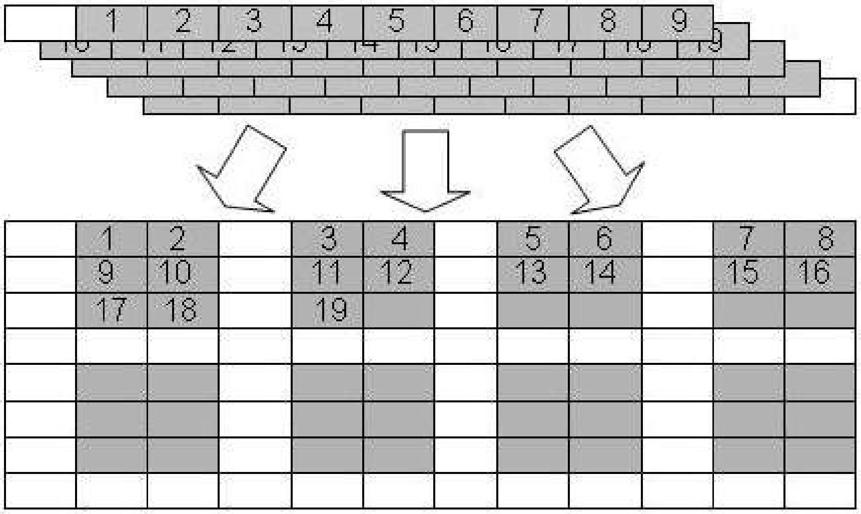
Figure 12

```
    ret = H5Dread(dataset, H5T_NATIVE_INT, memspace, dataspace, H5P_DEFAULT,
          data);
```

Figure 13

Example with strides and blocks.

Consider an 8x12 dataspace, in which we want to write into eight 3x2 blocks from a source dataspace in memory that is a 50-element one dimensional array called (Figure 14).



a) The source is a 1D array with 50 elements

b) The destination on disk is a 2D array with 48 selected elements

Figure 14

Figure 15 shows example code to write 48 elements from the 1D array to the file dataset, starting with the second element in vector. The destination hyperslab has the following parameters: start=(0,1), stride=(4,3), count=(2,4), block=(3,2). The source has the parameters: start=(1), stride=(1), count=(48), block=(1). After these operations, the file dataspace will have the values shown in Figure 14. Notice that the values are inserted in the file dataset in row-major order.

```
/* 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)
```

Figure 15

Selecting a union of hyperslabs

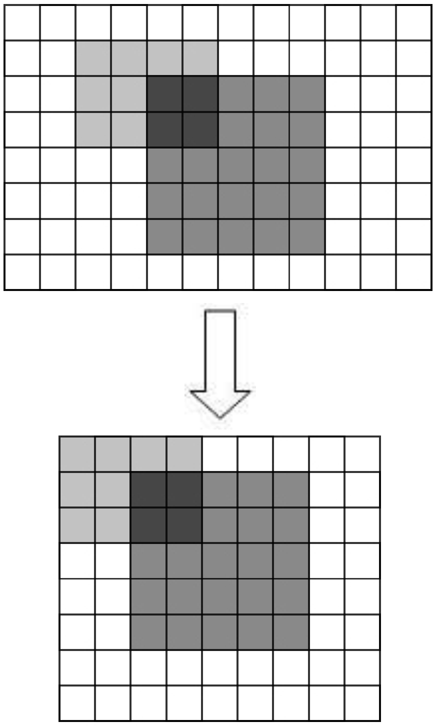


Figure 16

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.

Figure 16 shows a transfer of a selection that is two overlapping hyperslabs from the dataset into a union of hyperslabs in the memory dataset. Note that the destination dataset has a different shape from the source 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, the selection is that same shape at the destination.

To implement this transfer, it is necessary to:

1. get the source Dataspace
2. define one hyperslab selection for the source.
3. define a second hyperslab selection, unioned with the first.
4. get the destination Dataspace

5. define one hyperslab selection for the destination
6. define a second hyperslab seletion, unioned with the first.
7. execute the data transfer (H5Dread or H5Dwrite) using the source and destination Dataspaces.

Figure 17 shows example code to create the selections for the source Dataspace (the file). The first hyperslab is size 3x4 and the left upper corner at the position (1,2). The hyperslab is a simple rectangle, so the stride and block are 1. The second hyperslab is 6x5 at the position (2,4). The second selection is a union with the first hyperslab (H5S_SELECT_OR).

```
      fid = H5Dget_space(dataset);

    /*
      * Select first hyperslab for the dataset in the file.
      *
      */
     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);
     /*
      * Add second selected hyperslab to the selection.
      */
     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);
```

Figure 17

Figure 18 shows example code to create the selection for the destination in memory. The steps are similar. In this example, the hyperslabs are the same shape, but located in different positions in the Dataspace. The first hyperslab is 3x4 and starts at (0,0), and the second is 6x5 and starts at (1,2).

Finally the H5Dread call transfers the selected data from the file dataspace to the selection in memory.

In this example, the source and destination selections are two overlapping rectangles. In general, any number of rectangles can be OR'ed, and they do not have to be contiguous. The order of the selections does not matter, but the first should use H5S_SELECT_SET, subsequent selections are unioned using H5S_SELECT_OR.

It is important to emphasize that the source and destination do not have to be the same shape (or number of rectangles). As long as the two selections have the same number of elements, the data can be transferred.

```
    /*
     * 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);

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

Figure 18

Selecting a list of independent points

It is also possible to specify a list of elements to read or write using the function H5Sselect_elements. The procedure is similar to hyperslab selections.

1. get the source Dataspace
2. set the selected points
3. get the destination Dataspace
4. set the selected points
5. transfer the data using the source and destination Dataspaces

Figure 19 shows an example where four values are to be written to four separate points in a two dimensional Dataspace. The source Dataspace is a one dimensional array with the values 53, 59, 61, 67. The destination Dataspace is an 8x12 array. The elements are to be written to the points (0,0), (3,3), (3,5), and (5,6). In this example, the source does not require a selection. Figure 20 shows example code to implement this transfer.

A point selection lists the exact points to be transferred and the order they will be transferred. The source and destination are required to have the same number of elements. A point selection can be used with a hyperslab (e.g., the source could be a point selection and the destination a hyperslab, or vice versa), so long as the number of elements selected are the same.

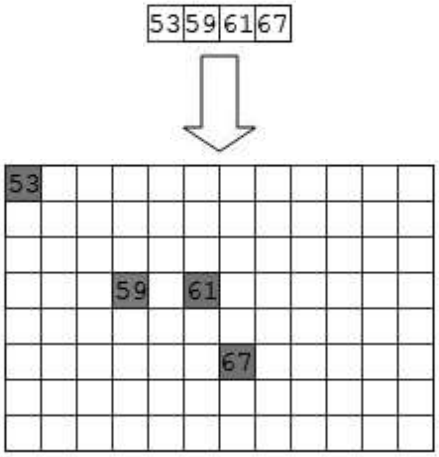


Figure 19

```
hsize_t dim2[] = {4};
int     values[] = {53, 59, 61, 67};

hssize_t coord[4][2]; /* Array to store selected points
                                         from the file dataspace */

/*
 * Create dataspace for the second dataset.
 */
mid2 = H5Screate_simple(1, 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);

ret = H5Dwrite(dataset, H5T_NATIVE_INT, mid2, fid, H5P_DEFAULT, values);
```

Figure 20

Combinations of selections

Selections are a very flexible mechanism for reorganizing data during a data transfer. With different combinations of *Dataspaces* and selections, it is possible to implement many kinds of data transfers, including sub-setting, sampling, and reorganizing the data. Table 2 gives some example combinations of source and destination, and the operations they implement.

Table 2

Source	Destination	Operation
all	all	Copy whole array
all	All (different shape)	Copy and reorganize array
hyperslab	all	Sub-set
hyperslab	Hyperslab (same shape)	selection
hyperslab	Hyperslab (different shape)	Select and rearrange
Hyperslab with stride or block	All or hyperslab with stride 1	Sub-sample, scatter
hyperslab	points	scatter
points	Hyperslab or all	gather
points	Points (same)	selection
points	Points (different)	Reorder points

5. Dataspace Selection Operations and Data Transfer

(With apologies to the reader, this section has yet to be written. -- The Editor)

6. References to Dataset Regions

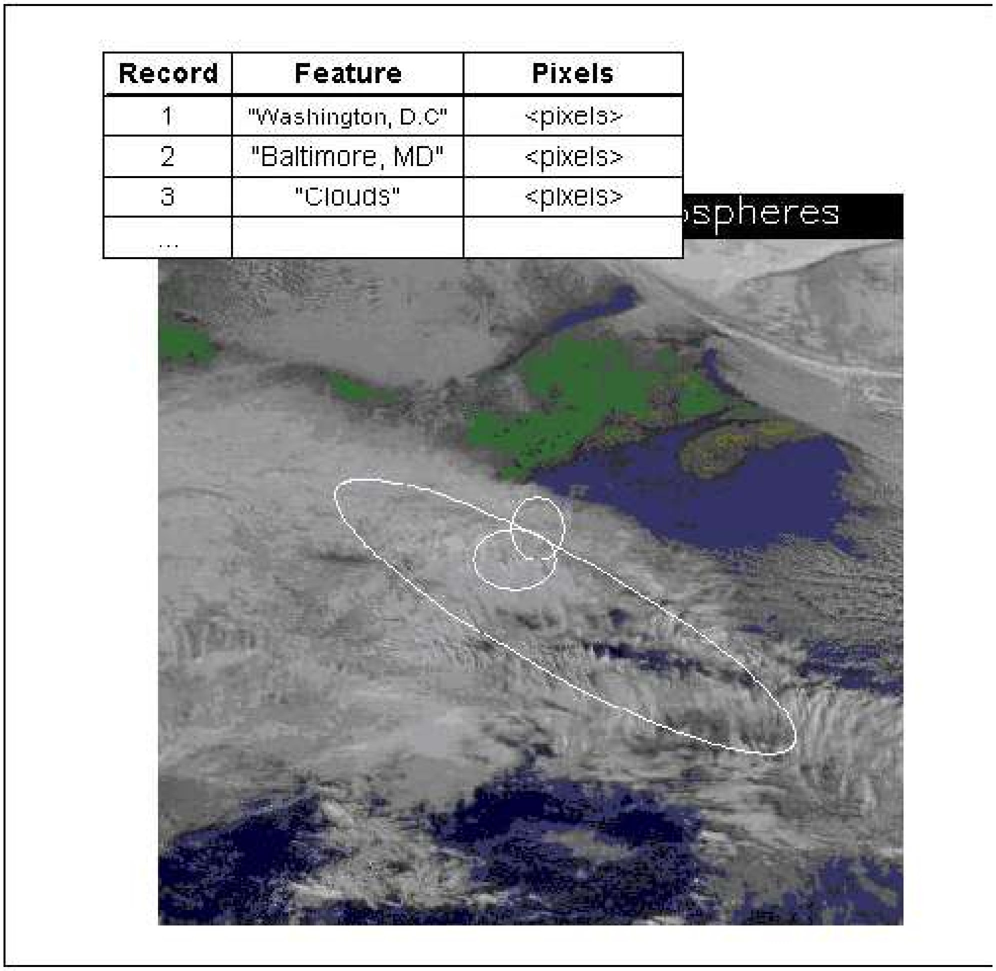
Another use of selections is to store a reference to a region of a Dataset. An HDF5 Object Reference Object is a pointer to an object (Dataset, Group, or Named Datatype) in the file. A selection can be used to create a pointer to a set of selected elements of a *Dataset*, called a Region Reference. The selection can be either a point selection or a hyperslab selection. A more complete description of Region References can be found in the chapter "HDF5 Datatypes."

A Region Reference is an object maintained by the HDF5 Library. The region reference can be stored in a Dataset or Attribute, and then read. The Dataset or Attribute is defined to have the special Datatype, H5T_STD_REF_DSETREG.

To discover the elements and/or read the data, the Region Reference can be deferenced. The H5Rdefrerence call returns a handle for the *Dataset*, and then the selected dataspace can be retrieved with H5Rget_select call. The selected *Dataspace* can be used to read the selected data elements.

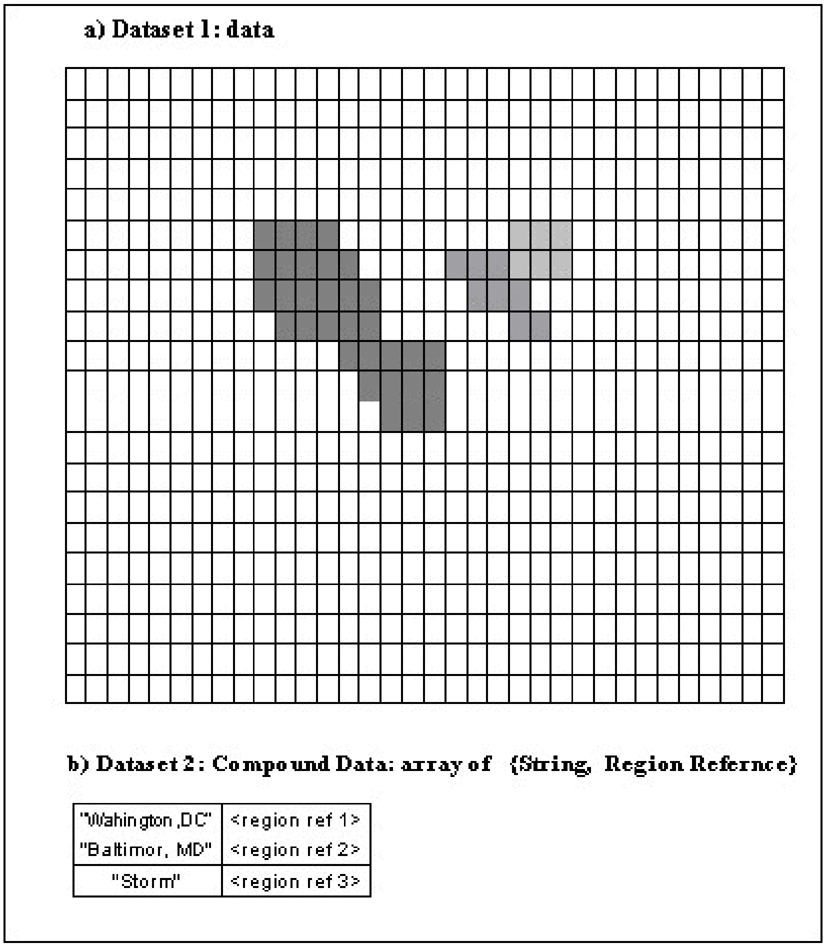
Example Uses for Region References

Region References are used to implement stored pointers to data within a Dataset. For example, features in a large dataset might be indexed by a table (Figure 21). This table could be stored as an HDF5 Dataset with a Compound Datatype, for example with a field for the name of the feature and a Region Reference to point to the feature in the Dataset (Figure 22).



Record	Feature	Pixels
1	"Washington, D.C"	<pixels>
2	"Baltimore, MD"	<pixels>
3	"Clouds"	<pixels>
...		

Figure 21


a) Dataset 1: data
b) Dataset 2: Compound Data: array of {String, Region Refernce}

"Wahington ,DC" "Baltimor, MD"	<region ref 1> <region ref 2>
"Storm"	<region ref 3>

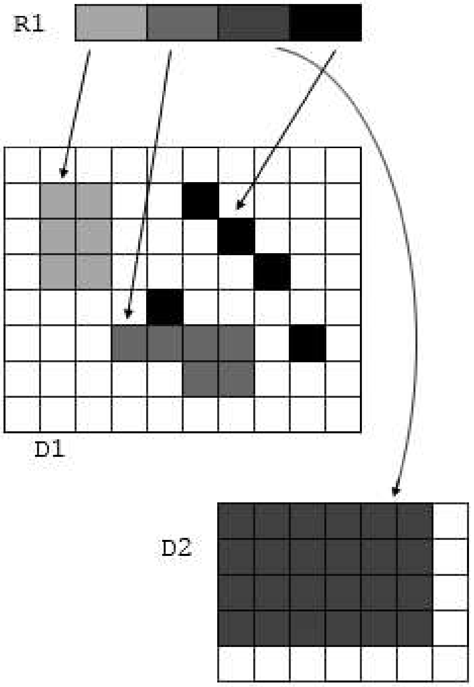
Figure 22

Creating References to Regions

To create a region reference:

1. Create or open the Dataset that contains the region.
2. Get the Dataspace for the Dataset.
3. Define a selection that specifies the region.
4. Create a Region reference using the Dataset and Dataspace with selection.
5. Write the Region Reference(s) to the desired Dataset or Attribute.

Figure 23 shows a diagram of a file with three Datasets. Dataset D1 and D2 are two dimensional arrays of integers. Dataset R1 is a one dimensional array of references to regions in D1 and D2. The regions can be any valid selection of the Dataspace of the target Dataset.



a) 1 D array of region pointers,
each pointer refers to a
selection in one Dataset.

Figure 23

Figure 24 shows example code to create the array of region references. The references are created in an array of type hdset_reg_ref_t. Each region is defined as a selection on the Dataspace of the Dataset, and a reference is created using H5Rcreate(). The call to H5Rcreate() specifies the file, Dataset, and the Dataspace with selection.

```
 /* create an array of 4 region references */
    hdset_reg_ref_t ref[4];
    /*
     * Create a reference to the first hyperslab in the first Dataset.
     */
    start[0] = 1; start[1] = 1;
    count[0] = 3; count[1] = 2;
    status =  H5Sselect_hyperslab(space_id, H5S_SELECT_SET, start, NULL,
          count, NULL);
    status = H5Rcreate(&ref[0], file_id, "D1", H5R_DATASET_REGION,
             space_id);

    /*
     * The second reference is to a union of hyperslabs in the first
 * Dataset
/*

    start[0] = 5;  start[1] = 3;
    count[0] = 1; count[1] = 4;
    status = H5Sselect_none(space_id);
    status = H5Sselect_hyperslab(space_id, H5S_SELECT_SET,start,
               NULL, count, NULL);
    start[0] = 6;   start[1] = 5;
    count[0] = 1;  count[1] = 2;
    status = H5Sselect_hyperslab(space_id, H5S_SELECT_OR, start, NULL,
          count, NULL);
    status = H5Rcreate(&ref[1], file_id, "D1", H5R_DATASET_REGION,
           space_id);

    /*
     * the fourth reference is to a selection of points in the first
     * Dataset
     */
    status = H5Sselect_none(space_id);
    coord[0][0] = 4; coord[0][1] = 4;
    coord[1][0] = 2; coord[1][1] = 6;
    coord[2][0] = 3; coord[2][1] = 7;
    coord[3][0] = 1; coord[3][1] = 5;
    coord[4][0] = 5; coord[4][1] = 8;
    status = H5Sselect_elements(space_id, H5S_SELECT_SET,num_points,
                                (const hssize_t **)coord);
    status = H5Rcreate(&ref[3], file_id, "D1", H5R_DATASET_REGION,
         space_id);
   /*
    * the third reference is to a hyperslab in the second Dataset
     */
    start[0] = 0;  start[1] = 0;
    count[0] = 4; count[1] = 6;
    status = H5Sselect_hyperslab(space_id2, H5S_SELECT_SET, start, NULL,
            count, NULL);
    status = H5Rcreate(&ref[2], file_id, "D2", H5R_DATASET_REGION,
            space_id2);
```

Figure 24

When all the references are created, the array of references is written to the Dataset R1. The Dataset is declared to have Datatype H5T_STD_REF_DSETREG (Figure 25).

```
Hsize_t dimsr[1];
dimsr[0] = 4;
/*
 * Dataset with references.
 */
spacer_id = H5Screate_simple(1, dimsr, NULL);
dsetr_id = H5Dcreate(file_id, "R1", H5T_STD_REF_DSETREG,
         spacer_id, H5P_DEFAULT);

/*
 * Write dataset with the references.
 */
status = H5Dwrite(dsetr_id, H5T_STD_REF_DSETREG, H5S_ALL, H5S_ALL,
        H5P_DEFAULT,ref);
```

Figure 25

When creating region references, the following rules are enforced.

- The selection must be a valid selection for the target *Dataset*, just as when transferring data.
- The *Dataset* must exist in the file when the reference is created (H5Rcreate).
- The target *Dataset* must be in the same file as the stored reference.

Reading References to Regions

To retrieve data from a region reference, the reference must be read from the file, and then the data can be retrieved. The steps are:

1. Open the Dataset or Attribute containing the reference objects.
2. Read the reference object(s).
3. For each Region Reference, get the Dataset (H5R_dereference) and Dataspace (H5Rget_space)
4. Use the Dataspace and Datatype to discover what space is needed to store the data, allocate the correct storage and create a Dataspace and Datatype to define the memory data layout.

Figure 26 shows example code to read an array of region references from a Dataset, and then read the data from the first selected region. Note that the region reference has information that records the Dataset (within the file) and the selection on the *Dataspace* of the *Dataset*. After dereferencing the regions reference, the *Datatype*, number of points, and some aspects of the selection can be discovered. (For a union of hyperslabs, it may not be possible to determine the exact set of hyperslabs that has been combined.) Table 3 shows the inquiry functions.

When reading data from a region reference, the following rules are enforced:

- The target *Dataset* must be present and accessible in the file.
- The selection must be a valid selection for the *Dataset*.

```
    dsetr_id = H5Dopen (file_id, "R1");

    status = H5Dread(dsetr_id, H5T_STD_REF_DSETREG, H5S_ALL, H5S_ALL,
                   H5P_DEFAULT, ref_out);

    /*
     * Dereference the first reference.
     *   1) get the dataset (H5Rdereference)
     *   2) get the selected dataspace (H5Rget_region)
     */
    dsetv_id = H5Rdereference(dsetr_id, H5R_DATASET_REGION,
           &ref_out[0]);
    space_id = H5Rget_region(dsetr_id, H5R_DATASET_REGION,&ref_out[0]);


    /*
     *  Discover how many points and shape of the data
     */
    ndims = H5Sget_simple_extent_ndims(space_id);

    H5Sget_simple_extent_dims(space_id,dimsx,NULL);

    /*
     * Read and display hyperslab selection from the dataset.
     */
      dimsy[0] = H5Sget_select_npoints(space_id);
      spacex_id = H5Screate_simple(1, dimsy, NULL);

    status = H5Dread(dsetv_id, H5T_NATIVE_INT, H5S_ALL, space_id,
                   H5P_DEFAULT, data_out);
    printf("Selected hyperslab: ");
    for (i = 0; i <8; i++)
    {
        printf("\n");
        for (j = 0; j  <10; j++)
            printf("%d ", data_out[i][j]);
    }
    printf("\n");
```

Figure 26

Table 3

Function	Information
H5Sget_select_npoints	The number of elements in the selection (hyperslab or point selection)
H5Sget_select_bounds	The bounding box that encloses the selected points (hyperslab or point selection)
H5Sget_select_hyper_nblocks	The number of blocks in the selection.
H5Sget_select_hyper_blocklist	A list of the blocks in the selection
H5Sget_select_elem_npoints	The number of points in the selection.
H5Sget_select_elem_pointlist	The points.

7. Sample Programs

This section contains the full programs from which several of the code examples in this chapter were derived. The h5dump output from the program's output file immediately follows each program.

```
h5_write.c
----------
#include "hdf5.h"

#define H5FILE_NAME        "SDS.h5"
#define DATASETNAME "C Matrix"
#define NX     3                      /* dataset dimensions */
#define NY     5
#define RANK   2

int
main (void)
{
     hid_t       file, dataset;         /* file and dataset handles */
     hid_t       datatype, dataspace;   /* handles */
     hsize_t     dims[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 + 1 + j*NY;
     }
     /*
      *  1  2  3  4  5
      *  6  7  8  9 10
      * 11 12 13 14 15
      */

     /*
      * Create a new file using H5F_ACC_TRUNC access,
      * default file creation properties, and default file
      * access properties.
      */
     file = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

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

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

     return 0;
}



SDS.out
-------
HDF5 "SDS.h5" {
GROUP "/" {
    DATASET "C Matrix" {
       DATATYPE  H5T_STD_I32BE
       DATASPACE  SIMPLE { ( 3, 5 ) / ( 3, 5 ) }
       DATA {
          1, 2, 3, 4, 5,
          6, 7, 8, 9, 10,
          11, 12, 13, 14, 15
       }
    }
}
}



h5_write.f90
------------
      PROGRAM DSETEXAMPLE

      USE HDF5 ! This module contains all necessary modules

      IMPLICIT NONE

      CHARACTER(LEN=7), PARAMETER :: filename = "SDSf.h5" ! File name
      CHARACTER(LEN=14), PARAMETER :: dsetname = "Fortran Matrix" ! Dataset name
      INTEGER, PARAMETER :: NX = 3
      INTEGER, PARAMETER :: NY = 5

      INTEGER(HID_T) :: file_id       ! File identifier
      INTEGER(HID_T) :: dset_id       ! Dataset identifier
      INTEGER(HID_T) :: dspace_id     ! Dataspace identifier


      INTEGER(HSIZE_T), DIMENSION(2) :: dims = (/3,5/) ! Dataset dimensions
      INTEGER     ::    rank = 2                       ! Dataset rank
      INTEGER     ::    data(NX,NY)

      INTEGER     ::   error ! Error flag
      INTEGER     :: i, j
```

```
!
! Initialize data
!
  do i = 1, NX
     do j = 1, NY
        data(i,j) = j + (i-1)*NY
     enddo
  enddo
!
! Data
!
!  1  2  3  4  5
!  6  7  8  9 10
! 11 12 13 14 15

!
! Initialize FORTRAN interface.
!
CALL h5open_f(error)

!
! Create a new file using default properties.
!
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error)

!
! Create the dataspace.
!
CALL h5screate_simple_f(rank, dims, dspace_id, error)

!
! Create and write dataset using default properties.
!
CALL h5dcreate_f(file_id, dsetname, H5T_NATIVE_INTEGER, dspace_id, &
                 dset_id, error)

CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, dims, error)

!
! End access to the dataset and release resources used by it.
!
CALL h5dclose_f(dset_id, error)

!
! Terminate access to the data space.
!
CALL h5sclose_f(dspace_id, error)

!
! Close the file.
!
CALL h5fclose_f(file_id, error)

!
! Close FORTRAN interface.
!
CALL h5close_f(error)

END PROGRAM DSETEXAMPLE
```

```
SDSf.out
--------
HDF5 "SDSf.h5" {
GROUP "/" {
    DATASET "Fortran Matrix" {
       DATATYPE  H5T_STD_I32BE
       DATASPACE  SIMPLE { ( 5, 3 ) / ( 5, 3 ) }
       DATA {
          1, 6, 11,
          2, 7, 12,
          3, 8, 13,
          4, 9, 14,
          5, 10, 15
       }
    }
}
}
```

```
h5_write_tr.f90
---------------
      PROGRAM DSETEXAMPLE

      USE HDF5 ! This module contains all necessary modules

      IMPLICIT NONE

      CHARACTER(LEN=10), PARAMETER :: filename = "SDSf_tr.h5" ! File name
      CHARACTER(LEN=24), PARAMETER :: dsetname = "Fortran Transpose Matrix"
                                                  ! Dataset name
      INTEGER, PARAMETER :: NX = 3
      INTEGER, PARAMETER :: NY = 5

      INTEGER(HID_T) :: file_id       ! File identifier
      INTEGER(HID_T) :: dset_id       ! Dataset identifier
      INTEGER(HID_T) :: dspace_id     ! Dataspace identifier


      INTEGER(HSIZE_T), DIMENSION(2) :: dims = (/NY, NX/) ! Dataset dimensions
      INTEGER     ::    rank = 2                       ! Dataset rank
      INTEGER     ::    data(NY,NX)

      INTEGER     ::   error ! Error flag
      INTEGER     :: i, j

      !
      ! Initialize data
      !
        do i = 1, NY
           do j = 1, NX
              data(i,j) = i + (j-1)*NY
           enddo
        enddo
      !
      ! Data
      !
      !  1  6  11
      !  2  7  12
      !  3  8  13
!  4  9  14
!  5 10  15

!
! Initialize FORTRAN interface.
!
CALL h5open_f(error)

!
! Create a new file using default properties.
!
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error)

!
! Create the dataspace.
!
CALL h5screate_simple_f(rank, dims, dspace_id, error)

!
! Create and write dataset using default properties.
!
CALL h5dcreate_f(file_id, dsetname, H5T_NATIVE_INTEGER, dspace_id, &
                 dset_id, error)

CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, dims, error)

!
! End access to the dataset and release resources used by it.
!
CALL h5dclose_f(dset_id, error)

!
! Terminate access to the data space.
!
CALL h5sclose_f(dspace_id, error)

!
! Close the file.
!
CALL h5fclose_f(file_id, error)

!
! Close FORTRAN interface.
!
CALL h5close_f(error)

END PROGRAM DSETEXAMPLE
```

```
SDSf_tr.out
-----------
HDF5 "SDSf_tr.h5" {
GROUP "/" {
    DATASET "Fortran Transpose Matrix" {
       DATATYPE  H5T_STD_I32LE
       DATASPACE  SIMPLE { ( 3, 5 ) / ( 3, 5 ) }
       DATA {
          1, 2, 3, 4, 5,
          6, 7, 8, 9, 10,
          11, 12, 13, 14, 15
       }
    }
}
}
```

Chapter 7
HDF5 Attributes

1. Introduction

An HDF5 attribute is a small meta data object describing the nature and/or intended usage of a primary data object, which may be a dataset, group, or named datatype.

Attributes are assumed to be very small as data objects go, so storing them as standard HDF5 datasets would be quite inefficient. HDF5 attributes are therefore managed through a special attributes interface, H5A, which is designed to easily attach attributes to primary data objects as small datasets containing metadata information and to minimize storage requirements

Consider, as examples of the simplest case, a set of laboratory readings taken under known temperature and pressure conditions of 18.0 degrees celsius and 0.5 atmospheres, respectively. The temperature and pressure could be stored as attributes of the dataset could be described as the following name/value pairs:

```
temp=18.0
pressure=0.5
```

While HDF5 attributes are not standard HDF5 datasets, they have much in common:

- An attribute has a user-defined dataspace and the included metadata has a user-assigned datatype.
- That metadata can be of any valid HDF5 datatype.
- Attributes are addressed by name.

Note:
Attributes are small datasets but not separate objects; they are contained within the object header of a primary data object. As such, attributes are opened, read, or written only with H5A functions.

But there are some very important differences:

- There is not provision for special storage, such as compression or chunking.
- There is no partial I/O or subsetting capability for attribute data.
- Attributes cannot be shared.
- Being small, an attributes is stored in the object header of the object it describes and is thus attached directly to that object.

Large attributes, described below in “Special Issues”, are best stored as separate HDF5 datasets and are not subject to the above limitations.

This chapter discusses or lists the following:

- The HDF5 attributes programming model
- H5A function summaries
- Working with HDF5 attributes
    - The structure of an attribute
    - Creating, writing, and reading attributes
    - Accessing attributes by name or index
    - Obtaining information regarding an object's attributes
    - Iterating across an object's attributes
    - Deleting an attribute
    - Closing attributes
- Special issues regarding attributes

In the following discussions, attributes are generally attached to datasets. Attributes attached to other primary data objects, i.e., groups or named datatypes, are handled in exactly the same manner.

2. Programming Model

2.1 To create and write a new attribute

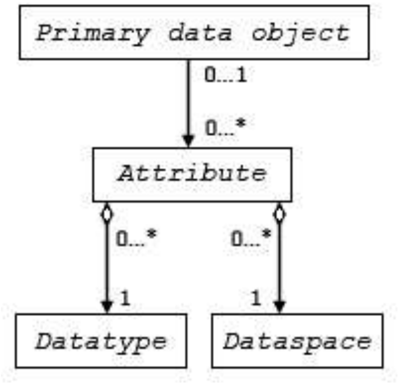


Figure 2: UML model for an HDF5 attribute and its associated dataspace and datatype.

Creating an attribute is similar to creating a dataset. To create an attribute, the application must specify the object to which the attribute is attached, the datatype and dataspace of the attribute data, and the attribute creation property list.

The following steps are required to create and write and an HDF5 attribute:

1. Obtain the object identifier for the attribute's primary data object.
2. Define the characteristics of the attribute and specify the attribute creation property list.
    - Define the datatype.
    - Define the dataspace.
    - Specify the attribute creation property list.
3. Create the attribute.
4. Write the attribute data (optional).
5. Close the attribute (and datatype, dataspace, and attribute creation property list, if necessary).
6. Close the primary data object (if appropriate).

2.2 To open and read/write an existing attribute

The following steps are required to open and read/write an existing attribute. Since HDF5 attributes allow no partial I/O, you need specify only the attribute and the attribute's memory datatype to read it:

1. Obtain the object identifier for the attribute's primary data object.
2. Obtain the attribute's name or index.
3. Open the attribute.
    - Get attribute dataspace and datatype (optional).
4. Specify the attribute's memory type.
5. Read and/or write the attribute data.
6. Close the attribute.
7. Close the primary data object (if appropriate).

3. Attribute (H5A) Function Summaries

C Function F90 Function	Purpose
`H5Acreate` `h5acreate_f`	Creates a dataset as an attribute of another group, dataset, or named datatype.
`H5Awrite` `H5awrite_f`	Writes an attribute.
`H5Aread` `h5aread_f`	Reads an attribute.
`H5Aopen_name` `h5aopen_name_f`	Opens an attribute specified by its name.
`H5Aopen_idx` `h5aopen_idx_f`	Opens the attribute specified by its index.
`H5Aclose` `h5aclose_f`	Closes the specified attribute.
`H5Aiterate` (none)	Calls a user’s function for each attribute attached to a data object.
`H5Adelete` `h5adelete_f`	Deletes an attribute.
`H5Aget_name` `h5aget_name_f`	Gets an attribute name.
`H5Aget_space` `h5aget_space_f`	Gets a copy of the dataspace for an attribute.
`H5Aget_type` `h5aget_type_f`	Gets an attribute datatype.
`H5Aget_num_attrs` `h5aget_num_attrs_f`	Determines the number of attributes attached to a data object.

4. Working with Attributes

4.1 The structure of an attribute

An attribute has two parts:

- name
- value(s)

HDF5 attributes are sometimes discussed as name/value pairs in the form name=value.

An attribute's name is a null-terminated ASCII character string. Each attribute attached to an object has a unique name.

The value portion of the attribute contains one or more data elements of the same datatype.

HDF5 attributes have all the characteristics of HDF5 datasets except that there is no partial I/O capability; attributes can be written and read only in full, with no subsetting.

4.2 Creating, writing, and reading attributes

If attributes are used at all in an HDF5 file, these three functions will be employed. `H5Acreate` and `H5Awrite` are used together to place the attribute in the file. If an attribute is to be used and it is not currently in memory, `H5Aread` generally comes into play, usually in concert with one each of the `H5Aget_*` and `H5Aopen_*` functions.

To create an attribute, call `H5Acreate`:

```
hid_t H5Acreate (hid_t loc_id, const char *name,
     hid_t type_id, hid_t space_id, hid_t create_plist)
```

loc_id identifies the object to which the attribute is to be attached, a dataset, group, or named datatype. This object, incidentally, is known as the primary data object; the attribute is a meta data object. name, type_id, space_id, and create_plist convey, respectively, the attribute's name, datatype, dataspace, and attribute creation property list. The attribute's name must be locally unique, i.e., it must be unique within the context of the object to which it is attached.

`H5Acreate` creates the attribute in memory; the attribute does not exist in the file until `H5Awrite` writes it there.

(Note that the attribute property list is currently unused. The only accepted value for create_plist is H5P_DEFAULT.)

To write or read an attribute, call `H5Awrite` or `H5Aread`, respectively:

```
herr_t H5Awrite (hid_t attr_id, hid_t mem_type_id,
     const void *buf)
herr_t H5Aread (hid_t attr_id, hid_t mem_type_id,
     void *buf)
```

attri_id identifies the attribute while mem_type_id identifies the in-memory datatype of the attribute data.

H5Awrite writes the attribute data, i.e., the meta data, from the buffer buf to the file; H5Aread reads attribute data from the file into buf.

The HDF5 library converts the meta data between the in-memory datatype, mem_type_id, and the in-file datatype, defined when the attribute was created, without user intervention.

4.3 Accessing attributes by name or index

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.

To access an attribute by its name, text text H5Aopen_name:

```
         hid_t H5Aopen_name (hid_t loc_id, const char *name)
```

H5Aopen_name returns an attribute identifier that can then be used by any function that must access an attribute, such as H5Aread.

Use the function H5Aget_name, described below, to determine an attribute's name. The information required to establish an index

To access an attribute by its index value, text text H5Aopen_idx:

```
         hid_t H5Aopen_idx (hid_t loc_id, unsigned index)
```

To determine an attribute index value when it is not already known, you must first use the function H5Aget_num_attrs, described below, to determine the number of attributes attached to the primary object. The index values of the attributes attached to that object range from 0 through 1 less than the value returned by H5Aget_num_attrs.

H5Aopen_idx is generally used in the course of opening several attributes for later access. Use H5Aiterate, described below, if the intent is to perform the same operation on every attribute attached to an object.

4.4 Obtaining information regarding an object's attributes

In the course of working with HDF5 attributes, one may need to obtain any of several pieces of information:

- An attribute name
- The dataspace of an attribute
- The datatype of an attribute
- The number of attributes attached to an object

To obtain an attribute's name, call H5Aget_name with an attribute identifier, attr_id:

```
         ssize_t H5Aget_name (hid_t attr_id, size_t buf_size,
              char *buf)
```

As with other attribut functions, attri_id identifies the attribute. buf is the buffer to which the attribute's name will be read; buf_size defines the size of that buffer.

If the length of the attribute name, and hence the value required for buf_size, is unknown, a first call to H5Aget_name will return that size. If the value of buf_size used in that first call is too small, the name will simply be truncated in buf. A second H5Aget_name call can then be used to retrieve the name in an appropriately-sized buffer.

To determine the dataspace or datatype of an attribute, call H5Aget_space or H5Aget_type, respectively:

```
hid_t H5Aget_space (hid_t attr_id)
hid_t H5Aget_type (hid_t attr_id)
```

H5Aget_space returns the dataspace identifier for the attribute attr_id.

H5Aget_type returns the datatype identifier for the attribute attr_id.

To determine the number or attributes attached to an object, call H5Aget_num_attrs:

```
int H5Aget_num_attrs (hid_t loc_id)
```

H5Aget_num_attrs returns the of number attributes attached to the object identified by the object identifier loc_id.

A call to H5Aget_num_attrs is generally the preferred first step in determining attribute index values. If the call to H5Aget_num_attrs returns N, the attributes attached to the object loc_id have index values of 0 through N-1

4.5 Iterating across an object's attributes

It is sometimes useful to be able to perform the identical operation across all of the objects attached to an object. At the simplest level, you might just want to open each attribute; at a higher level, you might wish to perform a rather complex operation on each attribute as you iterate across the set.

To iterate an operation across the attributes attached to an object, one must make a series of calls to H5Aiterate:

```
herr_t H5Aiterate (hid_t loc_id, unsigned *index,
     H5A_operator_t op_func, void *op_data)
```

H5Aiterate successively marches across all of the attributes attached to the object specified in loc_id, performing the operation(s) specified in op_func with the data specified in op_data on each attribute.

When H5Aiterate is called, index contains the index of the attribute to be accessed in this call; when H5Aiterate returns, index will contain the index of the next attribute. If the returned index is the null pointer, then all attributes have been processed and the iterative process is complete.

op_func is a user-defined operation that adheres to the H5A_operator_t prototype. This prototype and certain requirements imposed on the operator's behavior are described in the H5Aiterate entry in the *HDF5 Reference Manual*.

op_data is also user-defined to meet the requirements of op_func. Beyond providing a parameter with which to pass this data, HDF5 provides no tools for its management and imposes no restrictions.

4.6 Deleting an attribute

Once an attribute has outlived its usefulness or, for whatever reason, is no longer appropriate, it may become necessary to delete it.

To delete an attribute, call `H5Adelete`:

```
        herr_t H5Adelete (hid_t loc_id, const char *name)
```

`H5Adelete` removes the attribute name from the group, dataset, or named datatype specified in loc_id.

`H5Adelete` must not be called if there are any open attribute identifiers on the object loc_id. Such a call can cause the internal attribute indexes to change; future writes to an open attribute would then produce unintended results.

4.7 Closing an attribute

As is the case with all HDF5 objects, once access to an attribute it is no longer needed, that attribute must be closed. It is best practice to close it as soon as practicable; it is mandatory that it be closed prior to the `H5close` call closing the HDF5 library.

To close an attribute, call `H5Aclose`:

```
        herr_t H5Aclose (hid_t attr_id)
```

`H5Aclose` closes the specified attribute by terminating access to its identifier, attr_id.

Further use of attr_id is illegal; any function employing it will fail.

5. Special Issues

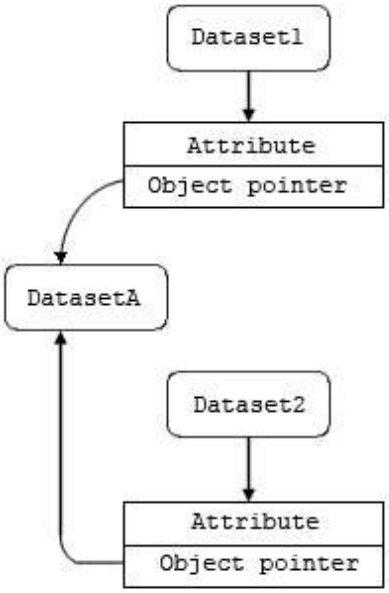


Figure 3: A large or shared HDF5 attribute and its associated dataset(s). DatasetA is an attribute of Dataset1 that may have been too large to store as an attribute. It is associated with Dataset1 by means of an object reference pointer attached as an attribute to Dataset1. Such an attribute can be shared among multiple datasets by means of addtional object reference pointers attached to addtional datasets.

Large attributes

Attributes are intended to be small objects. A large dataset intended as meta data for another dataset can be stored as a supplemental dataset. An attribute would then be attached to the original dataset indicating the relationship as an object reference pointer. This approach is illustrated in Figure 3.

How small is *small* and how large is *large* are not defined by the library; it is left to the user's interpretation. (In considering attributes and size, the HDF5 development team has considered attributes to be up to 16K, but this has never been set as a design or implementation limit.)

Shared attributes

Attributes written and managed through the H5A interface cannot be shared. If shared attributes are required, they must be handled in the manner described above for large attributes and illustrated in Figure 3.

Attribute names

While an attribute name may include any valid ASCII characters, including blanks, it is generally wise to keep readability issues in mind. In C, the name must be terminated with a null character, `\0`.

No special I/O or storage

HDF5 attributes have all the characteristics of HDF5 datasets except the following:

- ◊ Attributes are written and read only in full; there is no provision for partial I/O or subsetting.
- ◊ No special storage capability is provided for attributes; there is no compression or chunking and attributes are not extendable.