Learning HDF5 Basics

Copyright Notice and License Terms

See the Copyright Notices page on The HDF Group web site for the HDF5 Copyright Notice and Licensing Terms. This information can also be found in the COPYING file in the top directory of the HDF5 source code.

HDF5 is available with the SZIP compression library but SZIP is not part of HDF5 and has separate copyright and license terms. See Szip Compression in HDF Products for further details.

Documentation

See the HDF Support Portal for documentation and information on getting help.

Contents

HDF5 File Organization

An HDF5 file is a container for storing a variety of scientific data and is composed of two primary types of objects: groups and datasets.

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

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

Working with groups and datasets is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, an HDF5 object in an HDF5 file is often referred to by its **full path name** (also called an **absolute path name**).

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

The HDF5 API

The HDF5 library provides several interfaces, or APIs. These APIs provide routines for creating, accessing, and manipulating HDF5 files and objects.

The library itself is implemented in C. To facilitate the work of FORTRAN 90, C++ and Java programmers, HDF5 function wrappers have been developed in each of these languages. This tutorial discusses the use of the C functions and the FORTRAN wrappers.

All C routines in the HDF5 library begin with a prefix of the form *H5**, where * is one or two uppercase letters indicating the type of object on which the function operates. The FORTRAN wrappers come in the form of subroutines that begin with *h5* and end with *_f*. The APIs are listed below:

API	DESCRIPTION
H5	Library Functions: general-purpose H5 functions
H5A	Annotation Interface: attribute access and manipulation routines
H5D	Dataset Interface: dataset access and manipulation routines
H5E	Error Interface: error handling routines

H5F	File Interface: file access routines
H5G	Group Interface: group creation and operation routines
H5I	Identifier Interface: identifier routines
H5L	Link Interface: link routines
H5O	Object Interface: object routines
H5P	Property List Interface: object property list manipulation routines
H5R	Reference Interface: reference routines
H5S	Dataspace Interface: dataspace definition and access routines
H5T	Datatype Interface: datatype creation and manipulation routines
H5Z	Compression Interface: compression routine(s)

Programming Issues

Keep the following in mind when looking at the example programs included in this tutorial:

APIs vary with different languages

C routines begin with the prefix "H5*" where * is a single letter indicating the object on which the operation is to be performed:

File Interface: H5Fopen
Dataset Interface: H5Dopen

FORTRAN routines begin with "h5*" and end with "_f":

File Interface: h5fopen_f
Dataset Interface: h5dopen_f

APIS for languages like C++, Java, and Python use methods associated with specific objects.

HDF5 library has its own defined types

For portability, the HDF5 library has its own defined types. Some common types that you will see in the example code are:

hid_t is used for object handles
hsize_t is used for dimensions
herr_t is used for many return values

Language specific files must be included in applications

C: Add #include hdf5.h

FORTRAN: Add USE HDF5 and call h5open_f and h5close_f to initialize and close the HDF5 FORTRAN interface

Python: Add import h5py / import numpy

Example Programs

The HDF5 C, C++, and Fortran example programs used in this tutorial are included in the HDF5 source code and binary distributions

Compiling an Application

Compile scripts (h5cc, h5fc, h5c++) are provided with the HDF5 binary distribution to simplify compiling an application. For details see: [Compile Information]

Creating an HDF5 File

An HDF5 file is a binary file containing scientific data and supporting metadata.

To create an HDF5 file, an application must specify not only a file name, but a file access mode, a file creation property list, and a file access property list. These terms are described below:

File access mode:

When creating a file, the file access mode specifies the action to take if the file already exists:

- H5F_ACC_TRUNC specifies that if the file already exists, the current contents will be deleted so that the application can rewrite the file with new data.
- H5F_ACC_EXCL specifies that the open will fail if the file already exists. If the file does not already exist, the file access parameter is ignored.

In either case, the application has both read and write access to the successfully created file. Note that there are two different access modes for opening existing files:

- H5F_ACC_RDONLY specifies that the application has read access but will not be allowed to write any data.
- H5F_ACC_RDWR specifies that the application has read and write access.

File creation property list:

The file creation property list is used to control the file metadata. File metadata contains information about the size of the user-block*, the size of various file data structures used by the HDF5 library, etc. In this tutorial, the default file creation property list, H5P_DEFAULT, is used.

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

File access property list:

The file access property list is used to control different methods of performing I/O on files. It also can be used to control how a file is closed (whether or not to delay the actual file close until all objects in a file are closed). The default file access property list, H5P_DEFAULT, is used in this tutorial.

Please refer to the *H5F* section of the *HDF5 Users' Guide* and *Reference Manual* for detailed information regarding file access/creation property lists and access modes.

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

- Specify the file creation and access property lists, if necessary
- Create the file
- Close the file, and if necessary, close the property lists

Programming Example

Description

The following example code demonstrates how to create and close an HDF5 file.

C:

```
#include "hdf5.h"
#define FILE "file.h5"

int main() {

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

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

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

Fortran 90:

```
  PROGRAM FILEEXAMPLE

    USE HDF5 ! This module contains all necessary modules

    IMPLICIT NONE

    CHARACTER(LEN=8), PARAMETER :: filename = "filef.h5" ! File name
    INTEGER(HID_T) :: file_id                            ! File identifier

    INTEGER     ::   error  ! Error flag

!
!    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)

    !
    ! Terminate access to the file.
    !
    CALL h5fclose_f(file_id, error)
!
!    Close FORTRAN interface.
!
    CALL h5close_f(error)
    END PROGRAM FILEEXAMPLE
```

Remarks

- **In C:** The include file hdf5.h contains definitions and declarations and must be included in any program that uses the HDF5 library.

  In FORTRAN: The module HDF5 contains definitions and declarations and must be used in any program that uses the HDF5 library. Also note that h5open_f MUST be called at the beginning of an HDF5 Fortran application (prior to any HDF5 calls) to initialize the library and variables. The h5close_f call MUST be at the end of the HDF5 Fortran application.

- H5F_CREATE creates an HDF5 file and returns the file identifier.
  For Fortran, the file creation property list and file access property list are optional. They can be omitted if the default values are to be used.

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

- H5F_CLOSE terminates access to an HDF5 file.
  When an HDF5 file is no longer accessed by a program, H5F_CLOSE must be called to release the resources used by the file. This call is mandatory.

  Note that if H5F_CLOSE 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. This can cause access problems for other users, if objects were inadvertently left open. A File Access property controls how the file is closed.

File Contents

The HDF Group has developed tools for examining the contents of HDF5 files. The tool used throughout the HDF5 tutorial is the HDF5 dumper, h5dump, which displays the file contents in human-readable form. The output of h5dump is an ASCII display formatted according to the HDF5 DDL grammar. This grammar is defined, using Backus-Naur Form, in the DDL in BNF for HDF5.

To view the HDF5 file contents, simply type:

```
h5dump <filename>
```

Figure 4.1 describes the file contents of file.h5 (filef.h5) using a directed graph.

Fig. 4.1 *Contents of file.h5 (filef.h5)*

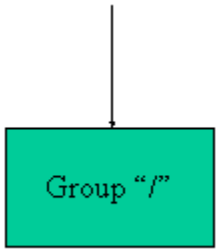


Figure 4.2 is the text description of file.h5, as generated by h5dump. The HDF5 file called file.h5 contains a group called /, or the *root group*. (The file called filef.h5, created by the FORTRAN version of the example, has the same output except that the filename shown is filef.h5.)

Fig. 4.2 *file.h5 in DDL*

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

File Definition in DDL

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

Fig. 4.3 *HDF5 File Definition*

The following symbol definitions are used in the DDL:

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

The simplified DDL for file definition is as follows:

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

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

<group_attribute> ::= <attribute>

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

Creating a Dataset

A dataset is a multidimensional array of data elements, together with supporting metadata. To create a dataset, the application program must specify the location at which to create the dataset, the dataset name, the datatype and dataspace of the data array, and the property lists.

Datatypes

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

There are two categories of datatypes in HDF5:

- **Pre-defined:** These datatypes are opened and closed by HDF5.

  Pre-defined datatypes can be atomic or composite:

  - Atomic datatypes cannot be decomposed into smaller datatype units at the API level. For example: integer, float, reference, string.
  - Composite datatypes are aggregations of one or more datatypes. For example: array, variable length, enumeration, compound.

- **Derived:** These datatypes are created or derived from the pre-defined types.

  A simple example of creating a derived datatype is using the string datatype, H5T_C_S1, to create strings of more than one character:

```
hid_t strtype;                      /* Datatype ID */
herr_t status;

strtype = H5Tcopy (H5T_C_S1);
status = H5Tset_size (strtype, 5); /* create string of length 5 */
```

Figure 5.1 shows the HDF5 pre-defined datatypes. Some of the HDF5 predefined atomic datatypes are listed in Figures 5.2a and 5.2b.

In this tutorial, we consider only HDF5 predefined integers.

For further information on datatypes, see The Datatype Interface (H5T) in the *HDF5 User's Guide*, in addition to the Datatypes tutorial topic.

Fig 5.1 *HDF5 datatypes*

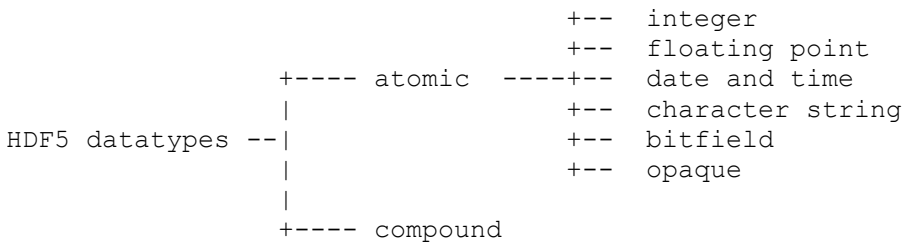


Fig. 5.2a *Examples of HDF5 predefined datatypes*

Datatype	Description
H5T_STD_I32LE	Four-byte, little-endian, signed, two's complement integer
H5T_STD_U16BE	Two-byte, big-endian, unsigned integer

H5T_IEEE_F32BE	Four-byte, big-endian, IEEE floating point
H5T_IEEE_F64LE	Eight-byte, little-endian, IEEE floating point
H5T_C_S1	One-byte, null-terminated string of eight-bit characters

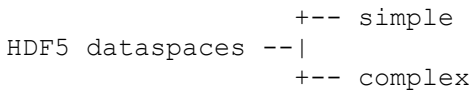
Fig. 5.2b *Examples of HDF5 predefined native datatypes*

Native Type	Corresponding C or FORTRAN Type
C:	
H5T_NATIVE_INT	int
H5T_NATIVE_FLOAT	float
H5T_NATIVE_CHAR	char
H5T_NATIVE_DOUBLE	double
H5T_NATIVE_LDOUBLE	long double
FORTRAN:	
H5T_NATIVE_INTEGER	integer
H5T_NATIVE_REAL	real
H5T_NATIVE_DOUBLE	double precision
H5T_NATIVE_CHARACTER	character

Datasets and Dataspaces

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

Fig 5.3 *HDF5 dataspaces*



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

Property Lists

Property lists are a mechanism for modifying the default behavior when creating or accessing objects. For more information on property lists see the Property List tutorial topic.

The following property lists can be specified when creating a dataset:

- Dataset Creation Property List

  When creating a dataset, HDF5 allows the user to specify how raw data is organized and/or compressed on disk. This information is stored in a dataset creation property list and passed to the dataset interface. The raw data on disk can be stored contiguously (in the same linear way that it is organized in memory), partitioned into chunks, stored externally, etc. In this tutorial, we use the default dataset creation property list (contiguous storage layout and no compression). For more information about dataset creation property lists, see The Dataset Interface (H5D) in the *HDF5 User's Guide*.

- Link Creation Property List

  The link creation property list governs creation of the link(s) by which a new dataset is accessed and the creation of any intermediate groups that may be missing.

- Dataset Access Property List

  Dataset access property lists are properties that can be specified when accessing a dataset.

Steps to Create a Dataset

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

1. Obtain the location identifier where the dataset is to be created.
2. Define or specify the dataset characteristics:
   a. Define a datatype or specify a pre-defined datatype.
   b. Define a dataspace.
   c. Specify the property list(s) or use the default.
3. Create the dataset.
4. Close the datatype, the dataspace, and the property list(s) if necessary.
5. Close the dataset.

In HDF5, datatypes and dataspaces 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 the definition of the datatype and dataspace. In this tutorial, we use the HDF5 predefined datatypes (integer) and consider only simple dataspaces. Hence, only the creation of dataspace objects is needed.

High Level APIs

The High Level HDF5 Lite APIs (H5LT) include functions that simplify and condense the steps for creating datasets in HDF5. The examples in the following section use the standard APIs. For a quick start you may prefer to look at the HDF5 Lite APIs at this time.

If you plan to work with images, please look at the High Level HDF5 Image APIs (H5IM), as well.

Programming Example

Description

The *Create a dataset* example shows how to create an empty dataset. It creates a file called dset.h5 in the C version (dsetf.h5 in Fortran), defines the dataset dataspace, creates a dataset which is a 4x6 integer array, and then closes the dataspace, the dataset, and the file.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

Remarks

H5S_CREATE_SIMPLE creates a new simple dataspace and returns a dataspace identifier.
H5S_CLOSE releases and terminates access to a dataspace.

Example code:

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

 FORTRAN:
    CALL h5screate_simple_f (rank, dims, dataspace_id, hdferr,
maxdims=max_dims)
         or
    CALL h5screate_simple_f (rank, dims, dataspace_id, hdferr)

    CALL h5sclose_f (dataspace_id, hdferr)
```

H5D_CREATE creates an empty dataset at the specified location and returns a dataset identifier.
H5D_CLOSE closes the dataset and releases the resource used by the dataset. This call is mandatory.

Example code:

C:

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

```
   status = H5Dclose (dataset_id);
```

FORTRAN:

```
   CALL h5dcreate_f (loc_id, name, type_id, dataspace_id, dset_id, hdferr)
   CALL h5dclose_f (dset_id, hdferr)
```

Note that if using the pre-defined datatypes in FORTRAN, then a call must be made to initialize and terminate access to the pre-defined datatypes:

```
  CALL h5open_f (hdferr)
  CALL h5close_f (hdferr)
```

H5_OPEN must be called before any HDF5 library subroutine calls are made;
H5_CLOSE must be called after the final HDF5 library subroutine call.

See the programming example for an illustration of the use of these calls.

File Contents

The contents of the file dset.h5 (dsetf.h5 for FORTRAN) are shown in **Figure 5.4** and **Figures 5.5a** and **5.5b**.

Figure 5.4 *Contents of dset.h5 (dsetf.h5)*

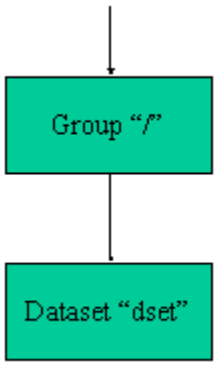


Figure 5.5a *dset.h5 in DDL*

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

Figure 5.5b *dsetf.h5 in DDL*

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

```
          0, 0, 0, 0, 0, 0
       }
    }
}
}
```

```
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0
            }
        }
    }
}
```

Note in Figures 5.5a and 5.5b that H5T_STD_I32BE, a 32-bit Big Endian integer, is an HDF atomic datatype.

Dataset Definition in DDL

The following is the simplified DDL dataset definition:

Fig. 5.6 *HDF5 Dataset Definition*

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

<datatype> ::= DATATYPE { <atomic_type> }

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

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

Reading From and Writing to a Dataset

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

- The dataset
- The dataset's datatype in memory
- The dataset's dataspace in memory
- The dataset's dataspace in the file
- The dataset transfer property list

  The dataset transfer property list controls various aspects of the I/O operations, such as the number of processes participating in a collective I/O request or hints to the library

to control caching of raw data. In this tutorial, we use the default dataset transfer property list.

- The data buffer

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

1. Obtain the dataset identifier.
2. Specify the memory datatype.
3. Specify the memory dataspace.
4. Specify the file dataspace.
5. Specify the transfer properties.
6. Perform the desired operation on the dataset.
7. Close the dataset.
8. Close the dataspace, datatype, and property list if necessary.

To read from or write to a dataset, the H5D_READ and H5D_WRITE routines are used.

```
C:
   status = H5Dread (set_id, mem_type_id, mem_space_id, file_space_id,
                     xfer_prp, buf );
   status = H5Dwrite (set_id, mem_type_id, mem_space_id, file_space_id,
                     xfer_prp, buf);

FORTRAN:
   CALL h5dread_f(dset_id, mem_type_id, buf, dims, error, &
                     mem_space_id=mspace_id, file_space_id=fspace_id, &
                     xfer_prp=xfer_plist_id)
        or
   CALL h5dread_f(dset_id, mem_type_id, buf, dims,  error)


   CALL h5dwrite_f(dset_id, mem_type_id, buf, dims, error, &
                     mem_space_id=mspace_id, file_space_id=fspace_id, &
                     xfer_prp=xfer_plist_id)
        or
   CALL h5dwrite_f(dset_id, mem_type_id, buf, dims, error)
```

High Level APIs

The High Level HDF5 Lite APIs include functions that simplify and condense the steps for creating and reading datasets. Please be sure to review them, in addition to this tutorial.

Programming Example

Description

The *Read and write to a dataset* example shows how to read and write an existing dataset. It opens the file created in the previous example, obtains the dataset identifier for the dataset /dset, writes the dataset to the file, and then reads the dataset back. It then closes the dataset and file.

Note that H5S_ALL is passed in for both the memory and file dataspace parameters in the read and write calls. This indicates that the entire dataspace of the dataset will be read or written to. H5S_ALL by itself does not necessarily have this meaning. See the Reference Manual entry for H5Dread or H5Dwrite for more information on using H5S_ALL.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

Remarks

H5F_OPEN opens an existing file and returns a file identifier.

H5D_OPEN opens an existing dataset with the specified name and location.

H5D_WRITE writes raw data from an application buffer to the specified dataset, converting from the datatype and dataspace of the dataset in memory to the datatype and dataspace of the dataset in the file. Specifying H5S_ALL for both the memory and file dataspaces indicates that the entire dataspace of the dataset is to be written to. H5S_ALL by itself does not necessarily have this meaning. See the Reference Manual entry for H5Dwrite for more information on using H5S_ALL.

H5D_READ reads raw data from the specified dataset to an application buffer, converting from the file datatype and dataspace to the memory datatype and dataspace. Specifying H5S_ALL for both the memory and file dataspaces indicates that the entire dataspace of the dataset is to be read. H5S_ALL by itself does not necessarily have this meaning. See the Reference Manual entry for H5Dread for more information on using H5S_ALL.

File Contents

Figure 6.1a shows the contents of dset.h5 (created by the C program).
Figure 6.1b shows the contents of dsetf.h5 (created by the FORTRAN program).

Fig. 6.1a *dset.h5 in DDL*

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

```
                1, 2, 3, 4, 5, 6,
                7, 8, 9, 10, 11, 12,
                13, 14, 15, 16, 17, 18,
                19, 20, 21, 22, 23, 24
             }
          }
       }
       }
```

Fig. 6.1b *dsetf.h5 in DDL*

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

Creating an Attribute

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

Creating an attribute is similar to creating a dataset. To create an attribute, the application must specify the object which the attribute is attached to, the datatype and dataspace of the attribute data, and the attribute creation property list.

The steps to create an attribute are as follows:

- Obtain the object identifier that the attribute is to be attached to.
- 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.
- Create the attribute.

- Close the attribute and datatype, dataspace, and attribute creation property list, if necessary.

To create and close an attribute, the calling program must use H5A_CREATE and H5A_CLOSE. For example:

```
C:
  attr_id = H5Acreate (dataset_id, "Units", H5T_STD_I32BE, dataspace_id,
H5P_DEFAULT, H5P_DEFAULT)
  status = H5Aclose (attr_id);

FORTRAN:
  CALL h5acreate_f (dset_id, attr_nam, type_id, space_id, attr_id, &
                    hdferr, creation_prp=creat_plist_id)
  or
  CALL h5acreate_f (dset_id, attr_nam, type_id, space_id, attr_id, hdferr)

  CALL h5aclose_f (attr_id, hdferr)
```

Reading/Writing an Attribute

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

The steps to read or write an attribute are as follows.

- Obtain the attribute identifier.
- Specify the attribute's memory datatype.
- Perform the desired operation.
- Close the memory datatype if necessary.

To read and/or write an attribute, the calling program must contain the H5A_READ and/or H5A_WRITE routines. For example:

```
C:
  status = H5Aread (attr_id, mem_type_id, buf);
  status = H5Awrite (attr_id, mem_type_id, buf);

FORTRAN:
  CALL h5awrite_f (attr_id, mem_type_id, buf, dims, hdferr)
  CALL h5aread_f (attr_id, mem_type_id, buf, dims, hdferr)
```

High Level APIs

The High Level HDF5 Lite APIs include functions that simplify and condense the steps for creating attributes in HDF5. Please be sure to review them, in addition to this tutorial.

Programming Example

Description

The *Create an attribute* example shows how to create and write a dataset attribute. It opens an existing file dset.h5 in C (dsetf.h5 in FORTRAN), obtains the identifier of the dataset /dset, defines the attribute's dataspace, creates the dataset attribute, writes the attribute, and then closes the attribute's dataspace, attribute, dataset, and file.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

Remarks

H5A_CREATE creates an attribute which is attached to the object specified by the first parameter, and returns an identifier.

H5A_WRITE writes the entire attribute, and returns the status of the write.

When an attribute is no longer accessed by a program, H5A_CLOSE must be called to release the attribute from use. An H5Aclose/h5aclose_f call is mandatory.

File Contents

The contents of dset.h5 (dsetf.h5 for FORTRAN) and the attribute definition are shown below:

Fig. 7.1a *dset.h5 in DDL*

```
HDF5 "dset.h5" {
GROUP "/" {
DATASET "dset" {
DATATYPE { H5T_STD_I32BE }
DATASPACE { SIMPLE ( 4, 6 ) / ( 4, 6 ) }
DATA {
 1, 2, 3, 4, 5, 6,
 7, 8, 9, 10, 11, 12,
 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24
}
ATTRIBUTE "attr" {
 DATATYPE { H5T_STD_I32BE }
 DATASPACE { SIMPLE ( 2 ) / ( 2 ) }
 DATA {
    100, 200
```

```
 }
}
}
}
}
```

Fig. 7.1b *dsetf.h5 in DDL*

```
HDF5 "dsetf.h5" {
GROUP "/" {
DATASET "dset" {
DATATYPE { H5T_STD_I32BE }
DATASPACE { SIMPLE ( 6, 4 ) / ( 6, 4 ) }
DATA {
 1, 7, 13, 19,
 2, 8, 14, 20,
 3, 9, 15, 21,
 4, 10, 16, 22,
 5, 11, 17, 23,
 6, 12, 18, 24
}
ATTRIBUTE "attr" {
 DATATYPE { H5T_STD_I32BE }
 DATASPACE { SIMPLE ( 2 ) / ( 2 ) }
 DATA {
    100, 200
 }
}
}
}
}
```

Attribute Definition in DDL

Fig. 7.2 *HDF5 Attribute Definition*

```
<attribute> ::= ATTRIBUTE "<attr_name>" { <datatype>
                                          <dataspace>
                                          <data>  }
```

Creating a Group

An HDF5 group is a structure containing zero or more HDF5 objects. The two primary HDF5 objects are groups and datasets. To create a group, the calling program must:

- Obtain the location identifier where the group is to be created.
- Create the group.
- Close the group.

To create a group, the calling program must call H5G_CREATE.
To close the group, H5G_CLOSE must be called. The close call is mandatory.

For example:

```
C:
  group_id = H5Gcreate(file_id, "/MyGroup", H5P_DEFAULT, H5P_DEFAULT,
H5P_DEFAULT);
  status = H5Gclose (group_id);

FORTRAN:
  CALL h5gcreate_f (loc_id, name, group_id, error)
  CALL h5gclose_f (group_id, error)
```

Programming Example

Description

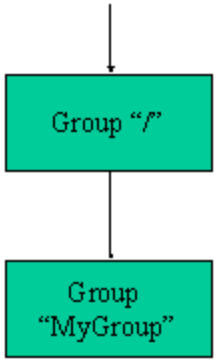
The *Create a group* example shows how to create and close a group. It creates a file called group.h5 (groupf.h5 for FORTRAN), creates a group called MyGroup in the root group, and then closes the group and file.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

File Contents

The contents of group.h5 and the definition of the group are shown below. (The FORTRAN program creates the HDF5 file groupf.h5 and the resulting DDL shows groupf.h5 in the first line.)

Fig. 8.1 *The Contents of group.h5.* **Fig. 8.2** *group.h5 in DDL*




```
HDF5 "group.h5" {
GROUP "/" {
   GROUP "MyGroup" {
   }
}
}
```

Creating Groups using Absolute and Relative Paths

Recall that to create an HDF5 object, we have to specify the location where the object is to be created. This location is determined by the identifier of an HDF5 object and the name of the object to be created. The name of the created object can be either an absolute name or a name relative to the specified identifier. In the previous example, we used the file identifier and the absolute name /MyGroup to create a group.

In this section, we discuss HDF5 names and show how to use absolute and relative names.

Names

HDF5 object names are a slash-separated list of components. There are few restrictions on names: component names may be any length except zero and may contain any character except slash (/) and the null terminator. A full name may be composed of any number of component names separated by slashes, with any of the component names being the special name . (a dot or period). A name which begins with a slash is an *absolute name* which is accessed beginning with the root group of the file; all other names are *relative names* and and the named object is accessed beginning with the specified group. A special case is the name / (or equivalent) which refers to the root group.

Functions which operate on names generally take a location identifier, which can be either a file identifier or a group identifier, and perform the lookup with respect to that location. Several possibilities are described in the following table:

File identifier	/foo/bar	The object bar in group foo in the root group.
Group	/foo/bar	The object bar in group foo in the root group of the file containing the specified

identifier		group. In other words, the group identifier's only purpose is to specify a file.
File identifier	/	The root group of the specified file.
Group identifier	/	The root group of the file containing the specified group.
Group identifier	foo/bar	The object bar in group foo in the specified group.
File identifier	.	The root group of the file.
Group identifier	.	The specified group.
Other identifier	.	The specified object.

Programming Example

Description

The *Create groups in a file using absolute and relative paths* example code shows how to create groups using absolute and relative names. It creates three groups: the first two groups are created using the file identifier and the group absolute names while the third group is created using a group identifier and a name relative to the specified group.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

Remarks

H5G_CREATE creates a group at the location specified by a location identifier and a name. The location identifier can be a file identifier or a group identifier and the name can be relative or absolute.

The first H5Gcreate/h5gcreate_f creates the group MyGroup in the root group of the specified file.

The second H5Gcreate/h5gcreate_f creates the group Group_A in the group MyGroup in the root group of the specified file. Note that the parent group (MyGroup) already exists.

The third H5Gcreate/h5gcreate_f creates the group Group_B in the specified group.

File Contents

The file contents are shown below:

Fig. 9.1 *The Contents of groups.h5 (groupsf.h5 for FORTRAN)*

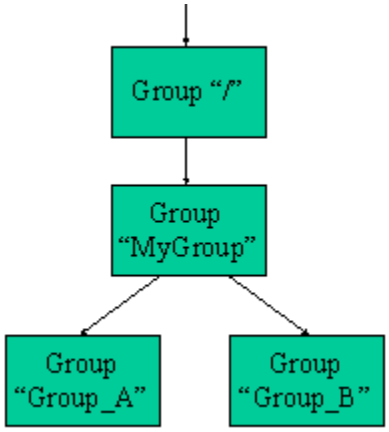


Fig. 9.2 *groups.h5 in DDL (for FORTRAN, the name in the first line is groupsf.h5)*

```
HDF5 "groups.h5" {
GROUP "/" {
 GROUP "MyGroup" {
    GROUP "Group_A" {
    }
    GROUP "Group_B" {
    }
 }
}
}
```

Creating Datasets in Groups

We have shown how to create groups, datasets, and attributes. In this section, we show how to create datasets in groups. Recall that H5D_CREATE creates a dataset at the location specified by a location identifier and a name. Similar to H5G_CREATE, the location identifier can be a file identifier or a group identifier and the name can be relative or absolute. The location identifier and the name together determine the location where the dataset is to be created. If the location identifier and name refer to a group, then the dataset is created in that group.

Programming Example

Description

The *Create datasets in a group* example shows how to create a dataset in a particular group. It opens the file created in the previous example and creates two datasets:

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages, including Java.

File Contents

Fig. 10.1 *The Contents of groups.h5 (groupsf.h5 for FORTRAN)*

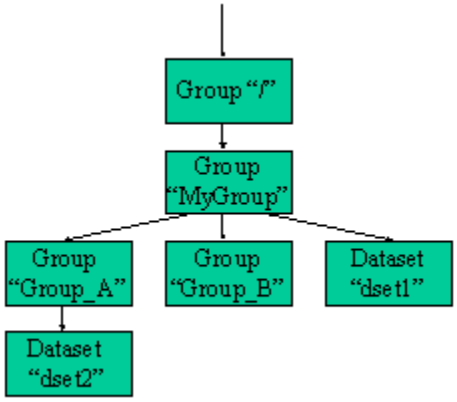


Fig. 10.2a *groups.h5 in DDL*

```
HDF5 "groups.h5" {
GROUP "/" {
GROUP "MyGroup" {
GROUP "Group_A" {
 DATASET "dset2" {
    DATATYPE { H5T_STD_I32BE }
    DATASPACE { SIMPLE ( 2, 10 ) / ( 2, 10 ) }
    DATA {
       1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    }
 }
}
GROUP "Group_B" {
}
DATASET "dset1" {
 DATATYPE { H5T_STD_I32BE }
 DATASPACE { SIMPLE ( 3, 3 ) / ( 3, 3 ) }
 DATA {
    1, 2, 3,
    1, 2, 3,
```

```
    1, 2, 3
 }
}
}
}
}
```

Fig. 10.2b *groupsf.h5 in DDL*

```
HDF5 "groupsf.h5" {
GROUP "/" {
GROUP "MyGroup" {
GROUP "Group_A" {
 DATASET "dset2" {
    DATATYPE { H5T_STD_I32BE }
    DATASPACE { SIMPLE ( 10, 2 ) / ( 10, 2 ) }
    DATA {
       1, 1,
       2, 2,
       3, 3,
       4, 4,
       5, 5,
       6, 6,
       7, 7,
       8, 8,
       9, 9,
       10, 10
    }
 }
}
GROUP "Group_B" {
}
DATASET "dset1" {
 DATATYPE { H5T_STD_I32BE }
 DATASPACE { SIMPLE ( 3, 3 ) / ( 3, 3 ) }
 DATA {
    1, 1, 1,
    2, 2, 2,
    3, 3, 3
 }
}
}
}
}
```

Reading from or Writing to a Subset of a Dataset

There are two ways that you can select a subset in an HDF5 dataset and read or write to it:

Hyperslab Selection : The H5S_SELECT_HYPERSLAB call selects a logically contiguous collection of points in a dataspace, or a regular pattern of points or blocks in a dataspace.

Element Selection*:* The H5S_SELECT_ELEMENTS call selects elements in an array.

HDF5 allows you to read from or write to a portion or subset of a dataset by:

- Selecting a Subset of the Dataset's Dataspace,
- Selecting a Memory Dataspace,
- Reading From or Writing to a Dataset Subset.

Selecting a Subset of the Dataset's Dataspace

First you must obtain the dataspace of a dataset in a file by calling H5D_GET_SPACE .

Then select a subset of that dataspace by calling H5S_SELECT_HYPERSLAB. The *offset*, *count*, *stride* and *block* parameters of this API define the shape and size of the selection. They must be arrays with the same number of dimensions as the rank of the dataset's dataspace. These arrays **ALL** work together to define a selection. A change to one of these arrays can affect the others.

offset: An array that specifies the offset of the starting element of the specified hyperslab.

count: An array that determines how many blocks to select from the dataspace in each dimension. If the block size for a dimension is one then the *count* is the number of elements along that dimension.

stride: An array that allows you to sample elements along a dimension. For example, a stride of one (or NULL) will select every element along a dimension, a stride of two will select every other element, and a stride of three will select an element after every two elements.

block: An array that determines the size of the element block selected from a dataspace. If the block size is one or NULL then the block size is a single element in that dimension.

Selecting a Memory Dataspace

You must select a memory dataspace in addition to a file dataspace before you can read a subset from or write a subset to a dataset. A memory dataspace can be specified by calling H5S_CREATE_SIMPLE.

The memory dataspace passed to the read or write call *must* contain the same number of elements as the file dataspace. The number of elements in a dataspace selection can be determined with the H5S_GET_SELECT_NPOINTS API.

Reading From or Writing To a Dataset Subset

To read from or write to a dataset subset, the H5D_READ and H5D_WRITE routines are used. The memory and file dataspace identifiers from the selections that were made are passed into the read or write call. For example (C):

```
status = H5Dwrite (.., .., memspace_id, dataspace_id, .., ..);
```

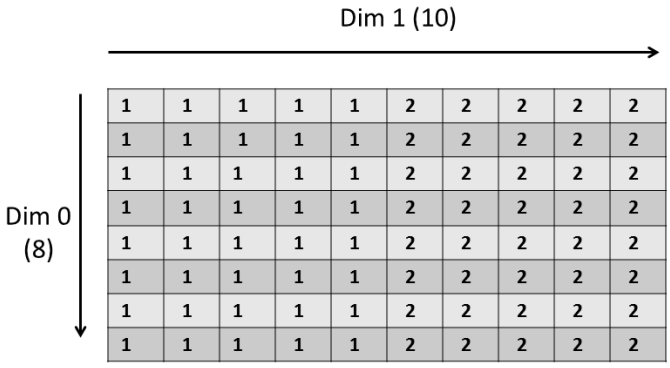
Programming Example

Description

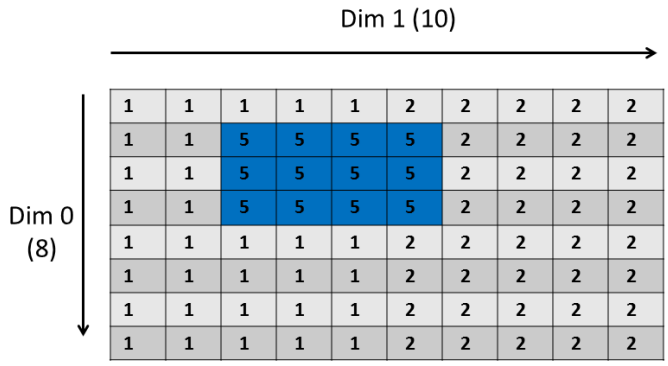
The *Create a file and dataset and select/read a subset from the dataset* example creates an 8 x 10 integer dataset in an HDF5 file. It then selects and writes to a 3 x 4 subset of the dataset created with the dimensions offset by 1 x 2. (If using FORTRAN, the dimensions will be swapped. The dataset will be 10 x 8, the subset will be 4 x 3, and the offset will be 2 x 1.)

PLEASE NOTE that the examples and images below were created using C.

The following image shows the dataset that gets written originally, and the subset of data that gets modified afterwards. Dimension 0 is vertical and Dimension 1 is horizontal as shown below:



Contents of Original Dataset Created



Contents of Dataset after 3 x 4 Subset Written

The subset on the right above is created using these values for offset, count stride, and block:

```
offset = {1, 2}
count = {3, 4}
stride = {1, 1}
block = {1, 1}
```

Sample code:

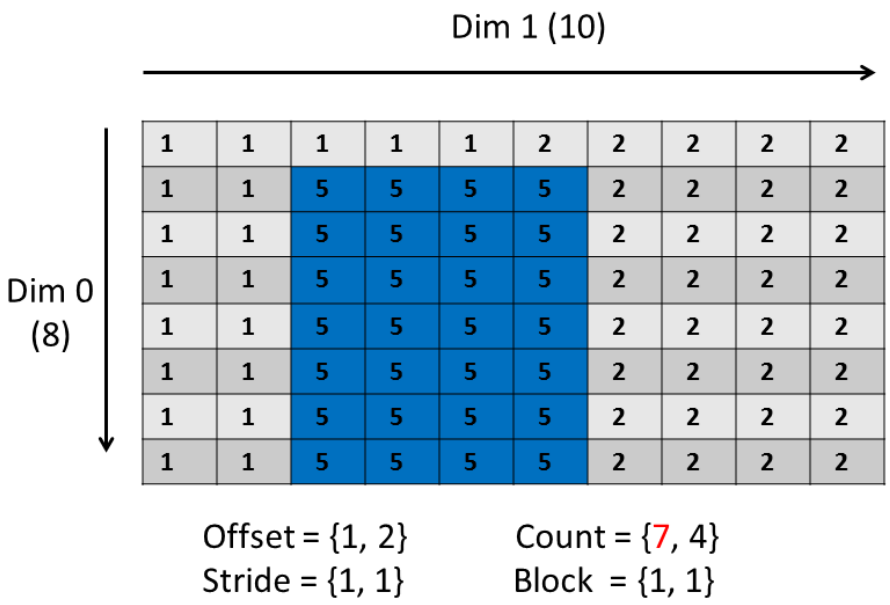
See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages.

Experiments with Different Selections

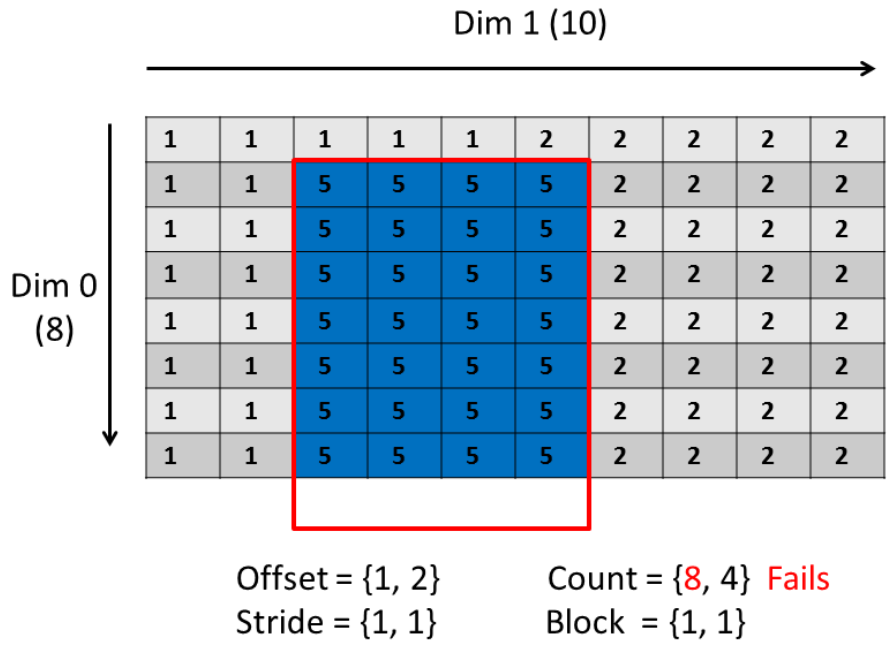
Following are examples of changes that can be made to the example code provided to better understand how to make selections.

Example 1:

By default the example code will select and write to a 3 x 4 subset. You can modify the *count* parameter in the example code to select a different subset, by changing the value of DIM0_SUB (C, C++) / dim0_sub (Fortran) near the top. Change its value to 7 to create a 7 x 4 subset:



If you were to change the subset to 8 x 4, the selection would be beyond the extent of the dimension:



The write will fail with the error: **"file selection+offset not within extent"**

Example 2:

In the example code provided, the memory and file dataspaces passed to the H5Dwrite call have the same size, 3 x 4 (DIM0_SUB x DIM1_SUB). Change the size of the memory dataspace to be 4 x 4 so that they do not match, and then compile:

```
dimsm[0] = DIM0_SUB + 1;
dimsm[1] = DIM1_SUB;
memspace_id = H5Screate_simple (RANK, dimsm, NULL);
```

The code will fail with the error: **"src and dest data spaces have different sizes"**

How many elements are in the memory and file dataspaces that were specified above? Add these lines:

```
hssize_t    size;

/* Just before H5Dwrite call the following */
size = H5Sget_select_npoints (memspace_id);
printf ("\nmemspace_id size: %i\n", size);
size = H5Sget_select_npoints (dataspace_id);
printf ("dataspace_id size: %i\n", size);
```

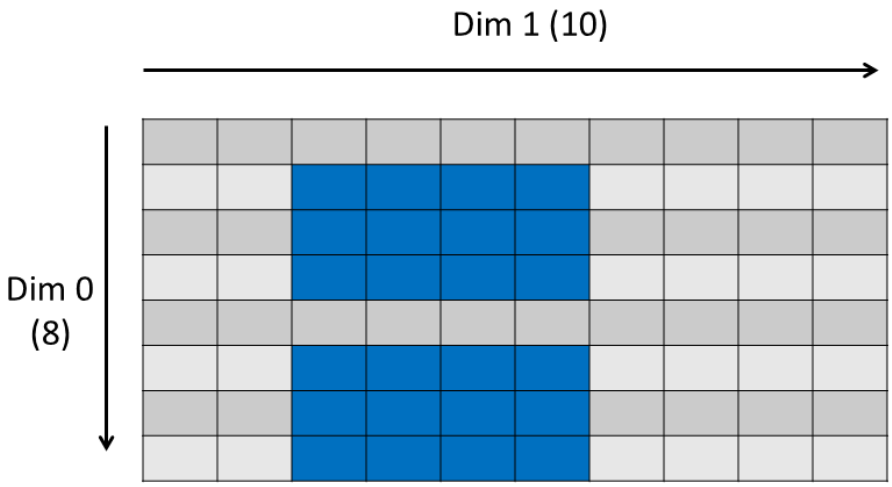
You should see these lines followed by the error:

```
memspace_id size: 16
dataspace_id size: 12
```

Example 3:

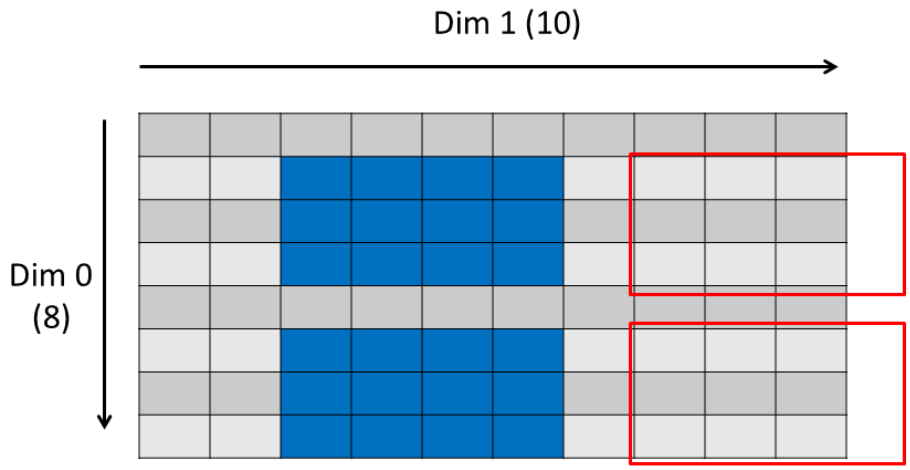
This example shows the selection that occurs if changing the values of the *offset*, *count*, *stride* and *block* parameters in the example code.

This will select two blocks. The *count* array specifies the number of blocks. The *block* array specifies the size of a block. The *stride* must be modified to accommodate the block size.



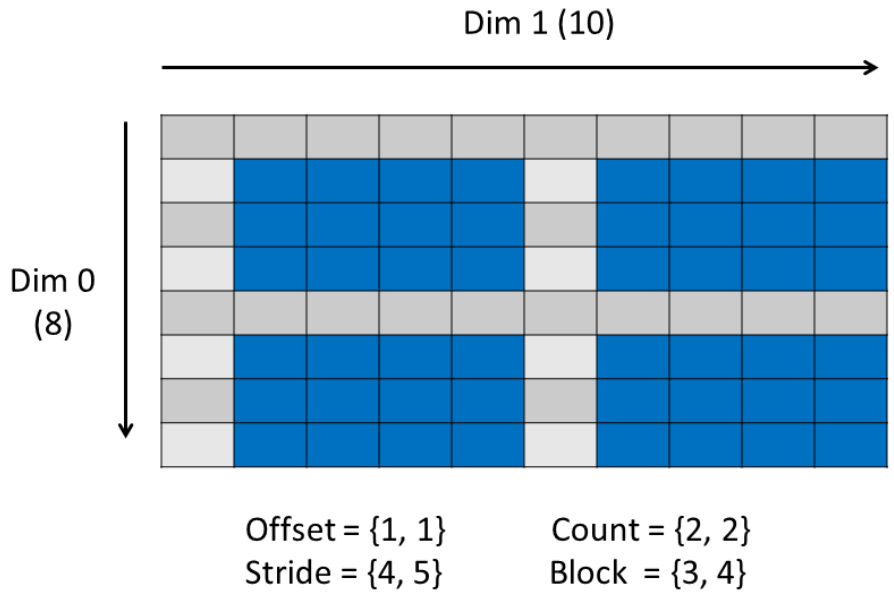
Offset = {1, 2} Count = {2, 1}
Stride = {4, 5} Block = {3, 4}

Now try modifying the *count* as shown below. The write will fail because the selection goes beyond the extent of the dimension:



Offset = {1, 2} Count = {2, 2}
Stride = {4, 5} Block = {3, 4}

If the offset were 1x1 (instead of 1x2), then the selection can be made:



The selections above were tested with the h5_subsetbk.c example code. The memory dataspace was defined as one-dimensional.

Remarks

- In addition to H5S_SELECT_HYPERSLAB, this example introduces the H5D_GET_SPACE call to obtain the dataspace of a dataset.
- If using the default values for the *stride* and *block* parameters of H5S_SELECT_HYPERSLAB, then, for C you can specify NULL for these parameters, rather than passing in an array for each, and for Fortran 90 you can omit these parameters.

Datatype Basics

What is a Datatype?

A datatype is a collection of datatype properties which provide complete information for data conversion to or from that datatype.

Datatypes in HDF5 can be grouped as follows:

- **Pre-Defined Datatypes:** These are datatypes that are created by HDF5. They are actually opened (and closed) by HDF5, and can have a different value from one HDF5 session to the next.
- **Derived Datatypes:** These are datatypes that are created or derived from the pre-defined datatypes. Although created from pre-defined types, they represent a category unto themselves. An example of a commonly used derived datatype is a string of more than one character.

Pre-Defined Datatypes

The properties of pre-defined datatypes are:

- Pre-defined datatypes are opened and closed by HDF5.
- A pre-defined datatype is a handle and is *NOT PERSISTENT*. Its value can be different from one HDF5 session to the next.
- Pre-defined datatypes are Read-Only.
- As mentioned, other datatypes can be derived from pre-defined datatypes.

There are two types of pre-defined datatypes, *standard (file)* and *native*:

STANDARD

A standard (or file) datatype can be:

- **Atomic:** A datatype which cannot be decomposed into smaller datatype units at the API level.
  The atomic datatypes are: integer, float, string, date and time, bitfield, reference, opaque
- **Composite:** An aggregation of one or more datatypes.
  Composite datatypes include: array, variable length, enumeration, compound datatypes
- Array, variable length, and enumeration datatypes are defined in terms of a single atomic datatype, whereas a compound datatype is a datatype composed of a sequence of datatypes.

Notes:

- Standard pre-defined datatypes are the **SAME** on all platforms.
- They are the datatypes that you see in an HDF5 file.
- They are typically used when *creating* a dataset.

NATIVE

Native pre-defined datatypes are used for memory operations, such as reading and writing. They are **NOT THE SAME** on different platforms. They are similar to C type names, and are aliased to the appropriate HDF5 standard pre-defined datatype for a given platform.

For example, when on an Intel based PC, H5T_NATIVE_INT is aliased to the standard pre-defined type, H5T_STD_I32LE. On a MIPS machine, it is aliased to H5T_STD_I32BE.

Notes:

- Native datatypes are **NOT THE SAME** on all platforms.
- Native datatypes simplify memory operations (read/write). The HDF5 library automatically converts as needed.
- Native datatypes are **NOT** in an HDF5 File. The standard pre-defined datatype that a native datatype corresponds to is what you will see in the file.

The following table shows the native types and the standard pre-defined datatypes they correspond to. (Keep in mind that HDF5 can convert between datatypes, so you can specify a buffer of a larger type for a dataset of a given type. For example, you can read a dataset that has a short datatype into a long integer buffer.)

Fig. 1 *Some HDF5 pre-defined native datatypes and corresponding standard (file) type*

C Type	HDF5 Memory Type	HDF5 File Type *
Integer:		
int	H5T_NATIVE_INT	H5T_STD_I32BE or H5T_STD_I32LE
short	H5T_NATIVE_SHORT	H5T_STD_I16BE or H5T_STD_I16LE
long	H5T_NATIVE_LONG	H5T_STD_I32BE, H5T_STD_I32LE, H5T_STD_I64BE or H5T_STD_I64LE
long long	H5T_NATIVE_LLONG	H5T_STD_I64BE or H5T_STD_I64LE
unsigned int	H5T_NATIVE_UINT	H5T_STD_U32BE or H5T_STD_U32LE

unsigned short	H5T_NATIVE_USHORT	H5T_STD_U16BE or H5T_STD_U16LE
unsigned long	H5T_NATIVE_ULONG	H5T_STD_U32BE, H5T_STD_U32LE, H5T_STD_U64BE or H5T_STD_U64LE
unsigned long long	H5T_NATIVE_ULLONG	H5T_STD_U64BE or H5T_STD_U64LE
Float:		
float	H5T_NATIVE_FLOAT	H5T_IEEE_F32BE or H5T_IEEE_F32LE
double	H5T_NATIVE_DOUBLE	H5T_IEEE_F64BE or H5T_IEEE_F64LE

F90 Type	HDF5 Memory Type	HDF5 File Type *
integer	H5T_NATIVE_INTEGER	H5T_STD_I32(8,16)BE or H5T_STD_I32(8,16)LE
real	H5T_NATIVE_REAL	H5T_IEEE_F32BE or H5T_IEEE_F32LE
double-precision	H5T_NATIVE_DOUBLE	H5T_IEEE_F64BE or H5T_IEEE_F64LE

***** Note that the HDF5 File Types listed are those that are most commonly created. The file type created depends on the compiler switches and platforms being used. For example, on the Cray an integer is 64-bit, and using `H5T_NATIVE_INT` (C) or `H5T_NATIVE_INTEGER` (F90) would result in an `H5T_STD_I64BE` file type.

The following code is an example of when you would use *standard* pre-defined datatypes vs. *native* types:

```
#include "hdf5.h"

main() {

   hid_t       file_id, dataset_id, dataspace_id;
   herr_t      status;
   hsize_t     dims[2]={4,6};
   int         i, j, dset_data[4][6];

   for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
         dset_data[i][j] = i * 6 + j + 1;
```

```
      file_id = H5Fcreate ("dtypes.h5", H5F_ACC_TRUNC, H5P_DEFAULT,
H5P_DEFAULT);

      dataspace_id = H5Screate_simple (2, dims, NULL);

      dataset_id = H5Dcreate (file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                              H5P_DEFAULT);

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

      status = H5Dclose (dataset_id);

      status = H5Fclose (file_id);
   }
```

By using the native types when reading and writing, the code that reads from or writes to a dataset can be the same for different platforms.

Can native types also be used when creating a dataset? Yes. However, just be aware that the resulting datatype in the file will be one of the standard pre-defined types and may be different than expected.

What happens if you do not use the correct native datatype for a standard (file) datatype? Your data may be incorrect or not what you expect.

Derived Datatypes

ANY pre-defined datatype can be used to derive user-defined datatypes.

To create a datatype derived from a pre-defined type:

- Make a copy of the pre-defined datatype:

  tid = H5Tcopy (H5T_STD_I32BE);

- *Change* the datatype.

  There are numerous datatype functions that allow a user to alter a pre-defined datatype. Refer to the Datatype Interface in the HDF5 Reference Manual. Example functions are *H5Tset_size* and *H5Tset_precision*.

Character Strings:

A simple example of creating a derived datatype is using the string datatype, H5T_C_S1 (H5T_FORTRAN_S1), to create strings of more than one character:

```
hid_t strtype;                      /* Datatype ID */
herr_t status;

strtype = H5Tcopy (H5T_C_S1);
status = H5Tset_size (strtype, 5); /* create string of length 5 */
```

The ability to derive datatypes from pre-defined types allows users to create any number of datatypes, from simple to very complex.

Property List Basics

What is a Property (or Property List)?

In HDF5, a property or property list is a characteristic or feature associated with an HDF5 object. There are default properties which handle the most common needs. These default properties are specified by passing in H5P_DEFAULT for the Property List parameter of a function. Default properties can be modified by use of the Property List interface and function parameters.

The Property List API allows a user to take advantage of the more powerful features in HDF5. It typically supports unusual cases when creating or accessing HDF5 objects. There is a programming model for working with Property Lists in HDF5 (see below).

For examples of modifying a property list, see these tutorial topics:

What is the dataset storage layout of a dataset ?
Extendible Datasets
Compressed Datasets

There are many Property Lists associated with creating and accessing objects in HDF5. See the Property List Interface documentation in the HDF5 Reference Manual for a list of the different properties associated with HDF5 interfaces.

In Summary:

Properties are features of HDF5 objects that can be changed by use of the Property List API and function parameters.

Property lists provide a mechanism for adding functionality to HDF5 calls without increasing the number of arguments used for a given call.

The Property List API supports unusual cases when creating and accessing HDF5 objects.

Programming Model

Default properties are specified by simply passing in H5P_DEFAULT (C) / H5P_DEFAULT_F (F90) for the property list parameter in those functions for which properties can be changed.

The programming model for changing a property list is as follows:

- Create a copy or "instance" of the desired pre-defined property type, using the H5P_CREATE call. This will return a property list identifier. Please see the Reference Manual entry for H5P_CREATE, for a comprehensive list of the property types.
- With the property list identifier, modify the property, using the H5P APIs.
- Modify the object feature, by passing the property list identifier into the corresponding HDF5 object function.
- Close the property list when done, using H5P_CLOSE.

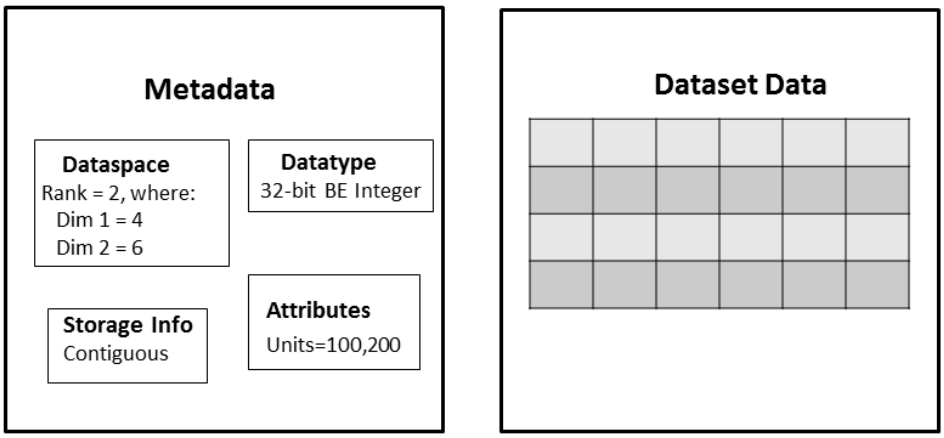
Dataset Storage Layout

Description of a Dataset

The Creating a Dataset tutorial topic defines a dataset as a multidimensional array of data elements *together with supporting metadata*, where:

- The array of elements consists of the raw data values that a user wishes to store in HDF5.
- The supporting metadata describes that data. The metadata is stored in the dataset (object) header of a dataset.

Datatype, dataspace, attribute, and *storage layout information* were introduced as part of the metadata associated with a dataset:



Dataset Storage Layout

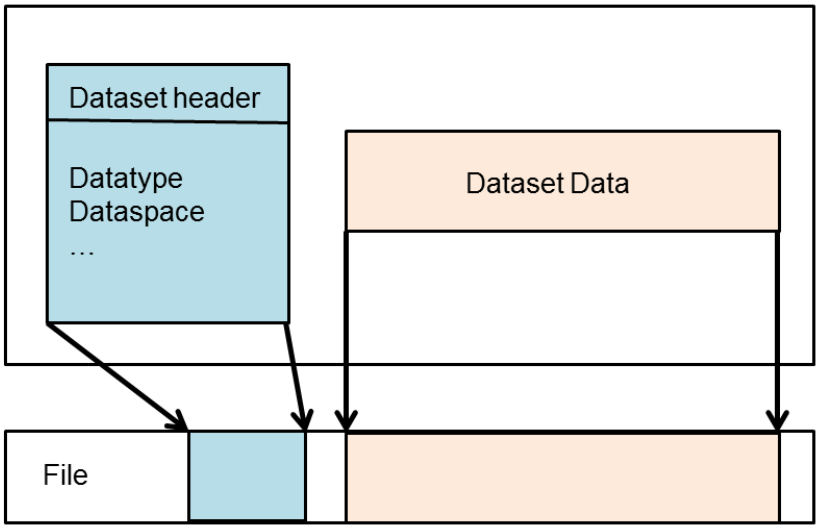
The storage information, or storage layout, defines how the raw data values in the dataset are physically stored on disk. There are three ways that a dataset can be stored:

- contiguous
- chunked
- compact

See the H5Pset_layout/H5Pget_layout APIs for details.

Contiguous

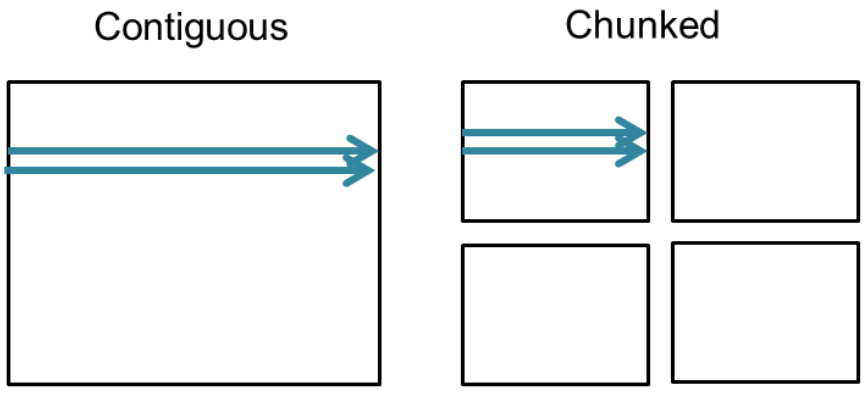
If the storage layout is contiguous, then the raw data values will be stored physically adjacent to each other in the HDF5 file (in one contiguous block). This is the default layout for a dataset. In other words, if you do not explicitly change the storage layout for the dataset, then it will be stored contiguously.



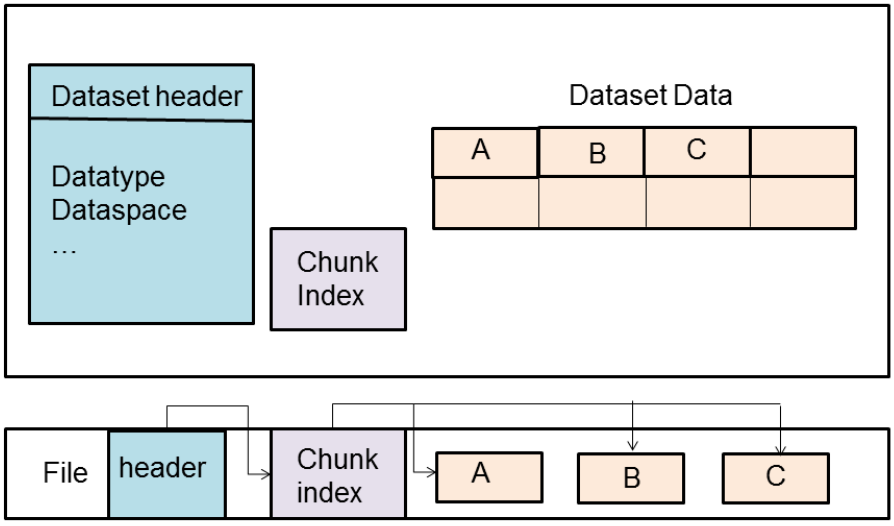
Contiguous Storage Layout

Chunked

With a chunked storage layout the data is stored in equal-sized blocks or chunks of a pre-defined size. The HDF5 library always writes and reads the entire chunk:



Each chunk is stored as a separate contiguous block in the HDF5 file. There is a chunk index which keeps track of the chunks associated with a dataset:



Chunked Storage Layout

Why Chunking ?

Chunking is required for enabling compression and other filters, as well as for creating extendible or unlimited dimension datasets.

It is also commonly used when subsetting very large datasets. Using the chunking layout can greatly improve performance when subsetting large datasets, because only the chunks required will need to be accessed. However, it is easy to use chunking without considering the consequences of the chunk size, which can lead to strikingly poor performance.

Note that a chunk always has the same rank as the dataset and the chunk's dimensions do not need to be factors of the dataset dimensions.

Writing or reading a chunked dataset is **transparent** to the application. You would use the same set of operations that you would use for a contiguous dataset. For example:

```
H5Dopen (...);
H5Sselect_hyperslab (...);
H5Dread (...);
```

Problems Using Chunking

Issues that can cause performance problems with chunking include:

- Chunks are too small.

  If a very small chunk size is specified for a dataset it can cause the dataset to be excessively large and it can result in degraded performance when accessing the dataset. The smaller the chunk size the more chunks that HDF5 has to keep track of, and the more time it will take to search for a chunk.

- Chunks are too large.

  An entire chunk has to be read and uncompressed before performing an operation. There can be a performance penalty for reading a small subset, if the chunk size is subtantially larger than the subset. Also, a dataset may be larger than expected if there are chunks that only contain a small amount of data.

- A chunk does not fit in the *Chunk Cache*.

  Every chunked dataset has a chunk cache associated with it that has a default size of 1 MB. The purpose of the chunk cache is to improve performance by keeping chunks that are accessed frequently in memory so that they do not have to be accessed from disk. If a chunk is too large to fit in the chunk cache, it can significantly degrade performance. However, the size of the chunk cache can be increased by calling H5Pset_chunk_cache.

It is a good idea to:

- Avoid very small chunk sizes, and be aware of the 1 MB chunk cache size default.
- Test the data with different chunk sizes to determine the optimal chunk size to use.
- Consider the chunk size in terms of the most common access patterns that will be used once the dataset has been created.

Compact

A compact dataset is one in which the raw data is stored in the object header of the dataset. This layout is for very small datasets that can easily fit in the object header.

The compact layout can improve storage and access performance for files that have many very tiny datasets. With one I/O access both the header and data values can be read. The compact layout reduces the size of a file, as the data is stored with the header which will always be allocated for a dataset. However, the object header is 64 KB in size, so this layout can only be used for very small datasets.

Programming Model to Modify the Storage Layout

To modify the storage layout, the following steps must be done:

- Create a Dataset Creation Property list. (See H5P_CREATE)
- Modify the property list.

  To use chunked storage layout, call: H5P_SET_CHUNK
  To use the compact storage layout, call: H5P_SET_LAYOUT

- Create a dataset with the modified property list. (See H5D_CREATE)
- Close the property list. (See H5P_CLOSE)

For example code, see the HDF5 Examples page. Specifically look at the Examples by API. There are examples for different languages. See *Read/Write Chunked Dataset* and *Read/Write Compact Dataset*.

Changing the Layout after Dataset Creation

The dataset layout is a Dataset Creation Property List. This means that once the dataset has been created the dataset layout cannot be changed. The h5repack utility can be used to write a file to a new with a new layout.

Sources of Information

Chunking in HDF5 (See the documentation on Advanced Topics in HDF5)
See "Properties and Property Lists" in the HDF5 User's Guide.

Extendible Datasets

Creating an Extendible Dataset

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

HDF5 requires you to use chunking to define extendible datasets. This makes it possible to extend datasets efficiently without having to excessively reorganize storage. *(To use chunking efficiently, be sure to see the advanced topic, Chunking in HDF5.)*

The following operations are required in order to extend a 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.
3. Create the dataset.

4. Extend the size of the dataset.

Programming Example

Description

The *Create an extendible (unlimited dimension) dataset* example shows how to create a 3 x 3 extendible dataset, write to that dataset, extend the dataset to 10x3, and write to the dataset again.

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages.

Remarks

- An unlimited dimension dataspace is specified with the H5S_CREATE_SIMPLE call, by passing in H5S_UNLIMITED as an element of the *maxdims* array.
- The H5P_CREATE call creates a new property as an instance of a property list class. For creating an extendible array dataset, pass in H5P_DATASET_CREATE for the property list class.
- The H5P_SET_CHUNK call modifies a Dataset Creation Property List instance to store a chunked layout dataset and sets the size of the chunks used.
- To extend an unlimited dimension dataset use the H5D_SET_EXTENT call. Please be aware that after this call, the dataset's dataspace must be refreshed with H5D_GET_SPACE before more data can be accessed.
- The H5P_GET_CHUNK call retrieves the size of chunks for the raw data of a chunked layout dataset.
- Once there is no longer a need for a Property List instance, it should be closed with the H5P_CLOSE call.

Compressed Datasets

Creating a Compressed Dataset

HDF5 requires you to use chunking to create a compressed dataset. *(To use chunking efficiently, be sure to see the advanced topic, Chunking in HDF5.)*

The following operations are required in order to create a compressed dataset:

- Create a dataset creation property list.
- Modify the dataset creation property list instance to enable chunking and to enable compression.

- Create the dataset.
- Close the dataset creation property list and dataset.

For more information on compression, see the FAQ question on Using Compression in HDF5.

Programming Example

Description

The *Create a chunked and compressed dataset* example creates a chunked and ZLIB compressed dataset. It also includes comments for what needs to be done to create an SZIP compressed dataset. The example then reopens the dataset, prints the filter information, and reads the dataset:

See HDF5 Introductory Examples for the examples used in the Learning the Basics tutorial. There are examples for several other languages.

Remarks

- The H5P_SET_CHUNK call modifies a Dataset Creation Property List instance to store a chunked layout dataset and sets the size of the chunks used.
- The H5P_SET_DEFLATE call modifies the Dataset Creation Property List instance to use ZLIB or DEFLATE compression. The H5Pset_szip call modifies it to use SZIP compression. There are different compression parameters required for each compression method.
- SZIP Limitations:
    - SZIP compression can only be used with atomic datatypes that are integer, float, or char. It cannot be applied to compound, array, variable-length, enumerations, or other user-defined datatypes. The call to H5D_CREATE **will fail** if attempting to create an SZIP compressed dataset with a non-allowed datatype. The conflict can only be detected when the property list is used.
    - There are restrictions for use of SZIP by commercial users. See the documents at SZIP Compression in HDF5 for further information on SZIP, including the *SZIP copyright notice*.

Discovering the Contents of an HDF5 File

Discovering what is in an HDF5 file

HDFView and h5dump are standalone tools which cannot be called within an application, and using H5Dopen and H5Dread require that you know the name of the HDF5 dataset. How would an application that has no prior knowledge of an HDF5 file be able to determine or discover the contents of it, much like HDFView and h5dump?

The answer is that there are ways to discover the contents of an HDF5 file, by using the H5G, H5L and H5O APIs:

- The H5G interface (covered earlier) consists of routines for working with groups. A group is a structure that can be used to organize zero or more HDF5 objects, not unlike a Unix directory.
- The H5L interface consists of link routines. A link is a path between groups. The H5L interface allows objects to be accessed by use of these links.
- The H5O interface consists of routines for working with objects. Datasets, groups, and committed datatypes are all *objects* in HDF5.

Interface routines that simplify the process:

- H5L_ITERATE traverses the links in a specified group, in the order of the specified index, using a user-defined callback routine. (A callback function is one that will be called when a certain condition is met, at a certain point in the future.)
- H5O_VISIT / H5L_VISIT recursively visit all objects/links accessible from a specified object/group.

Programming Examples

Using H5Literate, H5Lvisit and H5Ovisit:

Under HDF5 Examples you will find the Examples by API, where examples of using H5Literate and H5Ovisit/H5Lvisit are included.

The *Recursively Traverse a File with H5Literate* example traverses a file using H5Literate.
The *Recursively Traverse a File with H5Ovisit/H5Lvisit* example traverses a file using H5Ovisit and H5Lvisit.