

## HDF5 Groups

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

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

## HDF5 Datasets

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

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

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

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

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

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

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

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

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

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

The datatypes that are supported in the current implementation are:

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

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

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

#### Examples of Native Datatypes and Corresponding C Types

Example	Corresponding C Type
H5T_NATIVE_CHAR	signed char
H5T_NATIVE_UCHAR	unsigned char
H5T_NATIVE_SHORT	short
H5T_NATIVE_USHORT	unsigned short
H5T_NATIVE_INT	int
H5T_NATIVE_UINT	unsigned
H5T_NATIVE_LONG	long
H5T_NATIVE_ULONG	unsigned long
H5T_NATIVE_LLONG	long long
H5T_NATIVE_ULLONG	unsigned long long
H5T_NATIVE_FLOAT	float
H5T_NATIVE_DOUBLE	double
H5T_NATIVE_LDOUBLE	long double
H5T_NATIVE_HSIZE	hsize_t
H5T_NATIVE_HSSIZE	hssize_t
H5T_NATIVE_HERR	herr_t
H5T_NATIVE_HBOOL	hbool_t

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

*Named datatypes*. Normally each dataset has its own datatype, but sometimes we may want to share a datatype among several datasets. This can be done using a *named* datatype. A named datatype is stored in the file independently of any dataset, and referenced by all datasets that

have that datatype. Named datatypes may have an associated attributes list.

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

Properties of a dataspace consist of the *rank* (number of dimensions) of the data array, the *actual sizes of the dimensions* of the array, and the *maximum sizes of the dimensions* of the array. For a fixed-dimension dataset, the actual size is the same as the maximum size of a dimension. When a dimension is unlimited, the maximum size is set to the value H5P\_UNLIMITED. (An example below shows how to create extendible datasets.)

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

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

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

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

*Compact* storage is used when the amount of data is small and can be stored directly in the object header.

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

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

## HDF5 Attributes

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

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

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

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

### Hyperslab selection

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

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

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

- start: a starting location for the hyperslab. In the example start is (0,1).
- stride: the number of elements to separate each element or block to be selected. In the example stride is (4,3). If the stride parameter is set to NULL, the stride size defaults to 1 in each dimension.
- count: the number of elements or blocks to select along each dimension. In the example, count is (2,4).
- block: the size of the block selected from the dataspace. In the example, block is (3,2). If the block parameter is set to NULL, the block size defaults to a single element in each dimension, as if the block array was set to all 1s.

## Naming conventions

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

- **H5F**: File-level access routines.
- Example: H5Fopen, which opens an HDF5 file.
- **H5G**: Group functions, for creating and operating on groups of objects.
- Example: H5Gset, which sets the working group to the specified group.
- **H5T**: DataType functions, for creating and operating on simple and compound datatypes to be used as the elements in data arrays.
- Example: H5Tcopy, which creates a copy of an existing datatype.

- **H5S**: DataSpace functions, which create and manipulate the dataspace in which the elements of a data array are stored.
- Example: H5Screate\_simple, which creates simple dataspaces.
- **H5D**: Dataset functions, which manipulate the data within datasets and determine how the data is to be stored in the file.
- Example: H5Dread, which reads all or part of a dataset into a buffer in memory.
- **H5P**: Property list functions, for manipulating object creation and access properties.
- Example: H5Pset\_chunk, which sets the number of dimensions and the size of a chunk.
- **H5A**: Attribute access and manipulating routines.
- Example: H5Aget\_name, which retrieves name of an attribute.
- **H5Z**: Compression registration routine.
- Example: H5Zregister, which registers new compression and uncompression functions for use with the HDF5 library.
- **H5E**: Error handling routines.
- Example: H5Eprint, which prints the current error stack.
- **H5R**: Reference routines.
- Example: H5Rcreate, which creates a reference.
- **H5I**: Identifier routine.
- Example: H5Iget\_type, which retrieves the type of an object.

```
#include "hdf5.h"
#define H5FILE_NAME "SDS_row.h5"

int main (int argc, char **argv) {
    /* HDF5 APIs definitions */
    hid_t file_id; /* file and dataset identifiers */
    hid_t plist_id; /* property list identifier( access template) */
    herr_t status;

    int mpi_size, mpi_rank;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Info info = MPI_INFO_NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &mpi_size);
    MPI_Comm_rank(comm, &mpi_rank);

    /* Set up file access property list with parallel I/O access */
    plist_id = H5Pcreate(H5P_FILE_ACCESS);
    H5Pset_fapl_mpio(plist_id, comm, info);

    /* Create a new file collectively. */
    file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

    /* Close property list. */
    H5Pclose(plist_id);
}
```

```

/* Close the file. */
H5Fclose(file_id);

MPI_Finalize();

return 0;
}

```

## Writing Regularly Spaced Columns in C

In this example, you have two processes that write to the same dataset, each writing to every other column in the dataset. For each process the hyperslab in the file is set up as follows:

```

count[0] = 1;      count[1] = dimsm[1];
offset[0] = 0;    offset[1] = mpi_rank;
stride[0] = 1;    stride[1] = 2;
block[0] = dimsf[0]; block[1] = 1;

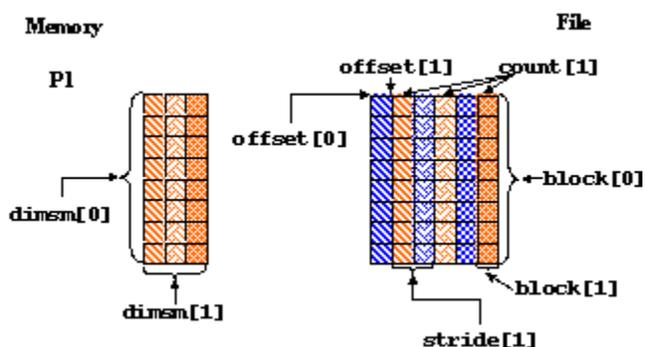
```

The *stride* is 2 for dimension 1 to indicate that every other position along this dimension will be written to. A *stride* of 1 indicates that every position along a dimension will be written to.

For two processes, the *mpi\_rank* will be either 0 or 1. Therefore:

- Process 0 writes to even columns (0, 2, 4...)
- Process 1 writes to odd columns (1, 3, 5...)

The *block* size allows each process to write a column of data to every other position in the



dataset.

Below is an example program for writing hyperslabs by column in Parallel HDF5:

```

#include "mpi.h"
#include "hdf5.h"
#include "stdlib.h"

```

```

#define H5FILE_NAME    "SDS_col.h5"
#define DATASETNAME    "IntArray"
#define NX      8      /* dataset dimensions */
#define NY      6
#define RANK    2

int main (int argc, char **argv) {

    hid_t    file_id, dset_id;          /* file and dataset identifiers */
    hid_t    filespace, memspace;      /* file and memory dataspace identifiers */
    hsize_t  dimsfs[2];                /* file dataset dimensions */
    hsize_t  dimsm[2];                 /* memory dataset dimensions */
    int      *data;                    /* pointer to data buffer to write */
    hsize_t  count[2];                 /* hyperslab selection parameters */
    hsize_t  stride[2];
    hsize_t  block[2];
    hsize_t  offset[2];
    hid_t    plist_id;                 /* property list identifier */
    int      i, j, k;
    herr_t    status;

    int mpi_size, mpi_rank;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Info info = MPI_INFO_NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &mpi_size);
    MPI_Comm_rank(comm, &mpi_rank);

    if (mpi_size != 2) {
        printf("This example to set up to use only 2 processes \n");
        printf("Quitting...\n");
        return 0;
    }

    /* Set up file access property list with parallel I/O access */
    plist_id = H5Pcreate(H5P_FILE_ACCESS);
    H5Pset_fapl_mpio(plist_id, comm, info);

    /* Create a new file collectively and release property list identifier. */
    file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
    H5Pclose(plist_id);

    /* Create the dataspace for the dataset. */
    dimsfs[0] = NX;
    dimsfs[1] = NY;
    dimsm[0] = NX;
    dimsm[1] = NY/2;
    filespace = H5Screate_simple(RANK, dimsfs, NULL);
    memspace = H5Screate_simple(RANK, dimsm, NULL);

```

```

/* Create the dataset with default properties and close filespace. */
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
                  H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
H5Sclose(filespace);

/* Each process defines dataset in memory and writes it to the hyperslab in the file. */
offset[0] = 0;
offset[1] = mpi_rank;
stride[0] = 1;
stride[1] = 2;
count[0] = 1;
count[1] = dimsm[1];
block[0] = dimsf[0];
block[1] = 1;

/* Select hyperslab in the file. */
filespace = H5Dget_space(dset_id);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, stride, count, block);

/* Initialize data buffer */
data = (int *) malloc(sizeof(int)*(size_t)dimsm[0]*(size_t)dimsm[1]);
for (i=0; i < dimsm[0]*dimsm[1]; i=i+dimsm[1]) {
    k = 1;
    for (j=0; j < dimsm[1]; j++) {
        data[i + j] = (mpi_rank + 1) * k ;
        k = k * 10;
    }
}

/* Create property list for collective dataset write. */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace, plist_id, data);
free(data);

H5Dclose(dset_id);
H5Sclose(filespace);
H5Sclose(memspace);
H5Pclose(plist_id);
H5Fclose(file_id);

MPI_Finalize();

return 0;
}

h5dump:

HDF5 "SDS_col.h5" {
GROUP "/" {

```

```
DATASET "IntArray" {  
  DATATYPE H5T_STD_I32LE  
  DATASPACE SIMPLE { ( 8, 6 ) / ( 8, 6 ) }  
  DATA {  
    (0,0): 1, 2, 10, 20, 100, 200,  
    (1,0): 1, 2, 10, 20, 100, 200,  
    (2,0): 1, 2, 10, 20, 100, 200,  
    (3,0): 1, 2, 10, 20, 100, 200,  
    (4,0): 1, 2, 10, 20, 100, 200,  
    (5,0): 1, 2, 10, 20, 100, 200,  
    (6,0): 1, 2, 10, 20, 100, 200,  
    (7,0): 1, 2, 10, 20, 100, 200  
  }  
}  
}
```