JHDF5 (HDF5 for Java) 19.04

Introduction

HDF5 is an efficient, well-documented, non-proprietary binary data format and library developed and maintained by the HDF Group. The library provided by the HDF Group is written in C and available under a liberal BSD-style Open Source software license. It has over 600 API calls and is very powerful and configurable, but it is not trivial to use.

SIS (formerly CISD) has developed an easy-to-use high-level API for HDF5 written in Java and available under the Apache License 2.0 called JHDF5. The API works on top of the low-level API provided by the HDF Group and the files created with the SIS API are fully compatible with HDF5 1.6/1.8/1.10 (as you choose).

Table of Content

Simple Use Case

Assume you want to write a `float[]`. All you have to do is:

```
float[] mydata = new float[1000];
...<fill mydata>...
IHDF5SimpleWriter writer = HDF5Factory.open("myfile.h5");
writer.writeFloatArray("mydata", mydata);
writer.close();
```

That's it. If you want to read your data, do:

```
IHDF5SimpleReader reader = HDF5Factory.openForReading("myfile.h5");
float[] mydata = reader.readFloatArray("mydata");
...<use mydata>...
reader.close();
```

There is a lot more functionality available in the API, but you don't have to learn it until you need it. For details about the library, including some simple example programs, see the javadoc.

Overview of the library

The full functionality of the library is available as a hierarchical, "quasi-fluent" API designed along the data types that JHDF5 can handle. It is explained in some detail in the javadoc of IHDF5Reader and IHDF5Writer.

The hierarchical ("quasi-fluent") API provides the full functionality. It is designed along the data types supported by JHDF5.

- `file()`: File-level information and operations, contains e.g. `close()` and `flush()` method.
- `object()`: Object-level information, where "objects" can be data sets, links, groups or data types, following the concept of an HDF5 object. Here you can find methods like `getGroupMemberInformation(String, boolean)` and createGroup(String) for creating and browsing groups and methods like `tryGetSymbolicLinkTarget(String)` and `createExternalLink(String, String, String)` for working with symbolic links.
- `bool()`: Reader and writer methods for boolean data sets, including bit fields.
- `int8()` / `int16()` / `int32()` / `int64()`: Reader and writer methods for signed integer data sets, where “8”, “16”, “32”, “64” denotes the number of bits of the integer type.
- `uint8()` / `uint16()` / `uint32()` / `uint64()`: Reader and writer methods for unsigned integer data sets, where “8”, “16”, “32”, “64” denotes the number of bits of the integer type. The methods will always read signed integers, as Java only knows signed integer types. If you need unsigned integers, you need to convert them using one of the methods in `UnsignedIntUtils`.
- `float32()` / `float64()`: Reader and writer methods for float data sets, where “32”

stands for 32bit floating point numbers, “64” for 64bit floating point numbers.

- `time()` / `duration()`: Reader and writer methods for time stamp (or date) and for time duration data sets.
- `string()`: Reader and writer methods for string data sets.
- `enumeration()`: Reader and writer methods for enumeration data sets.
- `compound()`: Reader and writer methods for compound data sets.
- `opaque()`: Reader and writer methods for data sets that are "black boxes" to HDF5 which are called "opaque data sets" in HDF5 jargon. Here you can also find methods of reading arbitrary data sets as byte arrays.
- `reference()`: Reader and writer methods for HDF5 object references. Note that object references, though similar to hard links on the first glance, are specific in HDF5 and have quite some differences in that it refers to the actual data block rather than to a metadata entity. All reader methods support transparently resolving references as returned by the reader methods of this interface.

When the HDF5 file is opened in read-only mode, read-only versions of each of these interfaces will be returned with the read-only subset of the methods.

Numeric Data Types

The reader and writer interfaces for numeric data types `int8()`, `int16()`, `int32()`, `int64()`, `uint8()`, `uint16()`, `uint32()`, `uint64()`, `float32()`, `float64()` all provide the same methods, just with different primitive Java data types.

The interfaces allow reading and writing of

- Scalar data sets (`write() / read()`),
- **One-dimensional array** data sets (`writeArray() / readArray()`),
- **Multi-dimensional array** data sets (`writeMDArray() / readMDArray()`),
- Scalar attributes (`getAttr() / setAttr()`),
- One-dimensional array attributes (`getArrayAttr() / setArrayAttr()`),
- Multi-dimensional array attributes (`getMDArrayAttr() / setMDArrayAttr()`).

There are methods for **matrices** (e.g. `float[][]`), but these are no special data types, but mere convenience types for two-dimensional arrays.

One- and multi-dimensional array data sets can be written and read in full or **block-wise**. The position of a block in the file can be either specified by “offset” (method suffix `WithOffset`) or by “block number” which is “offset x block size” (no suffix). The “natural blocks” are given by the chunk size that was used when creating the data set. The method `getArrayNaturalBlocks()` can be used to iterate over all natural blocks of a data set.

The library supports writing and reading **slices** of multi-dimensional data sets. A slice has one or more dimensions fixed to a given value (“bound indices”). Assume you have a 5-dimensional array with dimension $\{ i_0 = 0..9, i_1 = 0..9, i_2 = 0..9, i_3 = 0..9, i_4 = 0..9 \}$ (indices 0 – 4) and request reading a slice with bound indices $\{ i_1 \leftarrow 8, i_3 \leftarrow 0 \}$. This will return you a 3-dimensional data with free indices i_0, i_2, i_4: $\{ i_0 = 0..9, i_1 \leftarrow 8, i_2 = 0..9, i_3 \leftarrow 0, i_4 = 0..9 \}$. Slices and blocks can be combined, that is you can write or read a block of a slice of a data set (using methods like `writeSlicedMDArrayBlock()`).

Compound Data Types

Compound types in HDF5 are types that can combine multiple elementary values of a different elementary type, the so-called “members”. For example a compound type can have members of type `String`, a `float` and a `boolean`. Each of the values has a name (or key) that is used to access it. In memory, HDF5 compound types can be represented by simple Java classes of the *Data Transfer Object* type. Actually it is very simple to create an HDF5 compound type from a Java class like so:

```
HDF5CompoundType<MyData> type =
   writer.compound().getInferredAnonType(MyData.class);
```

One can then write an object of `MyData` into the file like this:

```
writer.compound().write("ds_name", type, myData);
```

Alternatively, the compound type can be created on-the-fly:

```
writer.compound().write("ds_name", myData);
```

A compound of this type can be read like so:

```
MyData cpd = reader.compound().read("name", MyData.class);
```

You can also write compound data from and read them into maps, lists or arrays by specifying instead of a DTO class a `HDF5CompoundDataMap.class`, `HDF5CompoundDataList.class` or `Object[].class`, respectively. This can be used to create and read compound structures that are only known at runtime. An example is:

```
HDF5CompoundDataMap cpd =
     reader.compound().read("name", HDF5CompoundDataMap.class);
```

System requirements

The minimum version of Java required is Java 8. The library has been tested with Opened / Oracle Java 8 and 11.

JHDF5 contains the native HDF5 library in version 1.10.x and other native libraries. For these native libraries, minimum requirements apply for the Operating System and CPU Model they can run on.

The following systems have been tested (in braces systems that should work):

- Linux x86 and x86_64: CentOS 7 and Ubuntu 16.04 (other modern Linux distributions should work)
- Linux ARMv7l: Caspian Stretch (2018-06-27) on a Raspberry Pi 3 Model B+ (older Caspian releases and Raspberry Pi generations and models should work)

- Apple MacOS X x86_64: MacOSX 10.13 (MacOSX 10.11 an 10.12 should work)
- Microsoft Windows 10 64 bit (Windows Server 2016 should work)

FAQ

What does HDF5 stand for?

HDF5 stands for *Hierarchical Data Format* v5.

It is an efficient, well-documented, non-proprietary binary data format and library developed and maintained by the HDF Group.

What about HDFx with x < 5?

There has never been HDF1, HDF2, or HDF3, but there is HDF4, see http://www.hdfgroup.org/products/hdf4/index.html. If you don't have files in HDF4 format, you have probably no need to use it.

What does the *Hierarchical* stand for in HDF?

You can use *groups* in HDF5 which are pretty much the same as directories in a file system. You specify a path to a data set in an HDF5 file the same way you would specify a path to a file in a Unix file system, that is "`/path/to/dataset`". If you don't use slashes, your data sets will end up in the root group ("/"). Thus the path "`dataset`" is equivalent to "`/dataset`".

Why would I want to use it?

Some good reasons are:

1. It is faster and more memory efficient than any ASCII based format.
2. It has a well-defined on-disk format.
3. It is portable and has been ported to many platforms.
4. It can be read and written using an Open Source library.
5. There is a dedicated group of people who is committed to ensure that you can still read data in that format in 50 years from now and this group has some customers with big pockets that have a vested interest in that, too. NASA, e.g., is using HDF5 for keeping the data of its long-term Earth Observing System.

Can I read and write data that are larger than the available RAM of my JRE?

Yes, you can. There are methods that allow you to do block-wise I/O. In order to create such a data set, you need to use methods like `IHDF5Writer.int64().createArray(String,long,int)`. For writing, you use

`IHDF5Writer.int64().writeArrayBlock(String, long[], int)`, for reading, you use `IHDF5Reader.int64().readArrayBlock(String, int, long)`.

Is JHDF5 thread-safe?

Yes. It is safe to use JHDF5 concurrently from multiple threads, even when accessing the same file or data set.

HDF5 sounds like a "file system within a file". How does it compare?

Yes, an HDF5 file is pretty much a Unix-style file system within a single file. Since you know the file system terms, it's handy to know how the HDF5 terms translate to file system terms, so here is the mapping table:

HDF5	file system
data set	file
group	directory
attribute	extended attribute (aka `xattr`)
hard link	hard link
soft link	soft / symbolic link
external link	a soft link to another file system that can be accessed without any additional operations to make the file system accessible

Are there important HDF5 concepts that do not translate to the file system analog?

Yes, "data type", "data space" and "storage layout" are such concepts. You don't need to know them to get started, but once you want more control over how the data are stored in the HDF5 file they become important.

What is a *Data Type*?

A *Data Type* is the HDF5 meta data about the data it writes to disk. While of course all that is written to disk ends up as a sequence of bytes, it is relevant to know whether these bytes constitute a string or a float value. Even when you know they are float data, it is important to know whether they are single or double precision and whether they are saved as *little endian* or *big endian* in order to interpret them correctly. HDF5 keeps this kind of information in the *Data Type* in a form that is independent of the hardware platform, the operating system and the programing environment you may use to write or access the data.

What is an *Opaque Data Type* and a *Tag*?

Data Types in HDF5 are supposed to be self-explaining, i.e. an array of bytes is supposed to be just that: an array of integer numbers which are small enough to fit in a byte. On the other hand, of course all binary data can be expressed as an array of bytes, though that data then have a "hidden structure". In order to mark such data types that have an unknown internal structure, HDF5 uses an *Opaque Data Type*, which has an internal representation as an array of bytes. Each *Opaque Data Type* has a *Tag* (which is just a string) to identify it. Note that the knowledge of the meaning of such an opaque type (the "hidden structure") needs to be made available separately; the *Tag* merely helps identifying it.

What is a *Data Space*?

In most cases, data in HDF5 are stored as an array of some kind. An HDF5 *Data Space* contains the information about the dimensions of this array, that is how many axes the array has and what is the extent of the array along each of these axes.

What is a *Storage Layout*?

Even when a *Data Space* completely defines the array structure from a logical point of view, there are various ways to store the data in the file: they can be kept either close to its meta data (called "compact" *Storage Layout*), in a separate place in the file as one big block (called "contiguous" *Storage Layout*) or in several chunks of some size (called "chunked" *Storage Layout*). All these storage layouts have different capabilities and performance characteristics and suit different use cases.

The API tries to automate the decision between the contiguous and the chunked *Storage Layout*, based on whether you want to be able to extend the data set later on and whether you want to compress (deflate) the data set or not. The choice for the compact storage layout, however, is left to the developer (see e.g. `IHDF5Writer.float32().writeArray(String,float[],HDF5FloatStorageFeatures)`. This *Storage Layout* is known to be very efficient for small data sets that you do not need to extend later on.

Can I read only exactly the same type that I wrote? (Numeric Conversions)

While you need to distinguish the basic types like strings, numeric values, enumerations, compounds, you can let the HDF5 library perform conversions for you between numerical values. Note that you may lose precision when doing so and that you can get an overflow (which will lead to an exception). You should also keep in mind that conversions are potentially time-consuming operations and only use them when really needed.

What are *Type Variants*?

While the *Data Type* describes how the values need to be interpreted on a low level (e.g. endianness), the data can still be interpreted in different ways. In most cases, this interpretation is too domain specific to put them into a generic library. In some cases,

however, it does make sense. Consider the case of a `long` value that should be interpreted as a time stamp (number of milliseconds since the start of the epoch). It seems reasonable to refer to this as a "timestamp" even when it ends up as a `long` value in the file. These are the *Type Variants* defined:

- `TIMESTAMP_MILLISECONDS_SINCE_START_OF_THE_EPOCH` – Time stamp in milliseconds since midnight, January 1, 1970 UTC (aka "start of the epoch")
- `TIME_DURATION_MICROSECONDS` – Time interval in micro seconds
- `TIME_DURATION_MILLISECONDS` – Time interval in milli seconds
- `TIME_DURATION_SECONDS` – Time interval in seconds
- `TIME_DURATION_MINUTES` – Time interval in minutes
- `TIME_DURATION_HOURS` – Time interval in hours
- `TIME_DURATION_DAYS` – Time interval in days
- `ENUM` – Enumeration, used when applying a scaling filter to an enumeration
- `NONE` – Denotes “no type variant”
- `BITFIELD` – Bit field, used when applying a scaling filter to a bit field.

More time variants may be added at a later point in a backward-compatible way.

Is there an easy way to convert a directory with its content from a file system to an HDF5 file and vice versa?

Yes, there is one. It is called `h5ar` (the "JHDF5 Archiver"). Just type

```
$ bin/h5ar
```

in the jhdf5 installation directory, or

```
$ java -jar sis-jhdf5-h5ar-cli-<version>.jar
```

and you'll get the usage information.

Note that with the HDF5 Archiver you don't have much control over how data end up in the archive: each file will be one data set and all data sets are plain byte arrays.

What are the options to provide the native libraries?

The native libraries can be either provided as *files* on the file system or as resources in a jar file that is in the class path. In the later case the resources will be copied to the file system before being linked. The native library loading tries the following methods:

1. Use a specific Java property `native.libpath.<libname>` for a library `<libname>`. The name of the library needs to be fully specified, e.g. for *jhdf5*: `-Dnative.libpath.jhdf5=/home/joe/java/native/jhdf5.so`. Note that the library provided needs to match the current platform that the program is running on.

2. Use a naming schema that looks for a library compatible with the platform the program runs on at a path that starts with `native.libpath`. This needs to point to a directory that has a structure like the one in subdirectory `lib/native/jhdf5`. Thus if you have unzipped the distribution to `/home/joe/jhdf5`, setting the JVM property `-Dnative.libpath=/home/joe/jhdf5/lib/native` will work.

3. There are versions `native.caching.libpath.<libname>` and `native.caching.libpath` which use the file or directory provided as a cache for the library version found in the jar file(s). Thus if the library file does not yet exist or is out-of-sync with the version on the classpath, the latest version from the classpath will be unpacked to the file system before loading the libraries from there, otherwise the version on the file system will be used (libraries are compared by size and CRC32 checksum).

4. Use a library packaged in a jar file by putting the jar file on the class path. Internally this uses the same mechanism for resolving a suitable library as methods 2 and the unpacking mechanism of method 3, but copies the library to a temporary file location where it is removed when the program finishes.
   This is the simplest way to use the library. If you are confused by the explanations given above, this may be what you want to use.

5. If none of the methods above work out, the fall back is to use the JVM default way of loading JNI libraries via the Java system call `System.loadLibrary()`. This may require the Java property `java.library.path` to be set and it may require the library to follow a platform specific naming convention.

If you need to trace how the library loaded is determined and what library file is finally loaded, set the property `-Dnative.libpath.verbose=true` and you will get debug output to `stderr` on library resolution and loading.

What does JHDF5 need `libunix.so` for? Isn't HDF5 platform agnostic?

Yes, HDF5 is written in a quite platform agnostic way and runs e.g. on Microsoft Windows. `libunix.so` is used only by JHDF5 Archiver, not the JHDF5 Core Libraries. Even for the Archiver to work they are not strictly needed. If the JHDF5 Archiver doesn't find `libunix.so` on a system (or `libunix.jnilib` on MacOS X system),which is always the case on Microsoft Windows systems, then it will not be able to retrieve and set file permissions and ownerships, but other than that the archiver will still work.