JHDF5 (HDF5 for Java) 14.06

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 (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.

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.

System requirements

The minimum version of Java required is Java 6. The library has been tested with JREs 6 and 7 from Oracle.

As JHDF5 contains native libraries, minimum requirements exist also for the Operating System and CPU.

The following systems are known to work:

- Linux x86 and x86_64: Redhat Enterprise Linux 5 and 6; other modern Linux distributions are expected to work as well
- Apple MacOS X x86: MacOS 10.6
- Apple MacOS X x86_64: MacOS 10.6, 10.7, 10.8 and 10.9
- Microsoft Windows x86: Windows XP, Windows Vista, Windows 7, Windows 8, Windows Server 2003 and 2008
- Microsoft Windows x86_64: Windows XP Professional x64 Edition, Windows Vista, Windows 7, Windows 8, Windows Server 2003 x64 Edition, Windows Server 2008R2

Support for the following systems is deprecated and will be removed in one of the next versions:

- Apple MacOS X x86: MacOS 10.6
- Apple MacOS X x86_64: MacOS 10.6
- Microsoft Windows x86: Windows XP, Windows Vista, Windows Server 2003
- Microsoft Windows x86_64: Windows XP Professional x64 Edition, Windows Vista, Windows Server 2003 x64 Edition

So, essentially the minimum requirement for Apple Mac OS X will be increased to Mac OS X 10.7 and the minimum requirement for Microsoft Windows will be increased to Windows 7. This is the last version of JHDF5 supporting JRE 6. The next version will require JRE 7.

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.

What does "batteries included" mean?

In the distribution, there is a directory `lib/batteries_included` which contains a jar file `sis-jhdf5-batteries_included.jar`. This jar file is what you should use to get started as it hides the complexity of finding a suitable native library from the user.

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.float().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 *Compound Types* and how can I create and use them?

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);
```

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.sh
```

in the jhdf5 installation directory, or

```
$ java -jar sis-jhdf5-batteries_included.jar
```

and you'll get a help text for it (there is currently no written documentation for `h5ar`).

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.

Can I use JHDF5 to run HDFView?

Yes. The jar files of JHDF5 can be used as a drop-in replacement of `jhdf5.jar` from HDF-Java 2.10.1 as released by The HDF Group. (In fact JHDF5 contains the low-level library JHI5 from HDF-Java 2.10.1 and will run any Java program that builds on JHI5). If you have copied `sis-jhdf5-batteries_included.jar` into the `lib` directory of HDF-Java, you can start HDFView like this:

```
$ java -cp \
lib/sis-jhdf5-batteries_included.jar:\
lib/jhdfobj.jar:lib/jhdf5obj.jar:lib/jhdfview.jar -Xmx1000m \
ncsa.hdf.view.HDFView
```

Why are there so many jar files in the distribution?

JHDF5 has dependencies on other, lower-level libraries. They can either be packaged together with the JHDF5 library, or they can be provided separately. Depending on your use-case, one or the other may be more convenient. Thus, the distribution contains both types of jar files. For a detailed description, see file CONTENT in the distribution.

The jar files that do not contain their dependencies packaged are:

- `sis-jhdf5-core.jar` – the base JHDF5 Library
- `sis-jhdf5-tools.jar` – currently holds the JHDF5 Archiver `h5ar` (may be

extended to also have other tools in a later version)

The jar files that do contain their dependencies packaged are:

- `sis-jhdf5.jar` – all Java classes, but no native libraries
- `sis-jhdf5-batteries_included.jar` – all Java classes and all native libraries for the supported platforms.

What are the options to provide the native libraries?

The native libraries can be either provided as *files* or java *resources*. In the later case the resources will be copied to a temporary file before being linked. The native library loading tries the following methods:

1. Use a specific Java property `native.libpath.<libname>` for each library `<libname>`. The name of the library needs to be fully given, e.g. `-Dnative.libpath.jhdf5=/home/joe/java/native/jhdf5.so`.

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`. Note that this needs to point to a directory that has a structure like the one in subdirectories `lib/native/jhdf5` and `lib/native/unix`. Thus if you have unzipped the distribution to `/home/joe/jhdf5`, setting `-Dnative.libpath=/home/joe/jhdf5/lib/native` will work.

3. Use a library packaged in a jar file and provided as a resource (by putting the jar file on the class path). Internally this uses the same directory structure as method 3., but packaged in a jar file so you don't have to care about it. Jar files with the appropriate structure are `sis-jhdf5-batteries_included.jar` and `lib/nativejar/*.jar` (one file for each platform).
This is the simplest way to use the library. If you are confused by the explanations given above, this is what you want to use.

4. Use the default way of loading JNI libraries via `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.

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 apart from that it will work.