JHI5 Design Notes

Description of the Java HDF5 Interface (JHI5)

Very Important Change: Version 3.0 (and above) of the JHI5 packages all HDF library calls as "hdf.hd5flib". Note that the "ncsa" has been removed. Source code which used earlier versions of the JHI5 should be changed to reflect this new implementation.

What it is

The **Java HDF5 Interface (JHI5)** is a Java package (*hdf.hdf5lib*) that ``wraps around" the HDF5 library.

There are over 460 functions in the HDF5 library (version 1.8). Ninety three of the functions are not supported in JHI5. Most of the unsupported functions have C function pointers, which is not currently implemented in JHI5. For a complete list of unsupported functions, please see unsupported functions.

Note: The JHI5 does not support HDF4 or earlier. See the JHI.

The JHI5 may be used by any Java application that needs to access HDF5 files. It is extremely important to emphasize that *this package is not a pure Java implementation of the HDF-5 library.* The JHI5 calls the same HDF5 library that is used by C or FORTRAN programs. (Note that this product cannot be used in most network browsers because it accesses the local disk using native code.)

The Java HDF5 Interface consists of Java classes and a dynamically linked native library. The Java classes declare native methods, and the library contains C functions which implement the native methods. The C functions call the standard HDF5 library, which is linked as part of the same library on most platforms.

The central part of the JHI5 is the Java class *hdf.hdf5lib.H5*. The *H5* class calls the standard (*i.e.*, `native' code) HDF5 library, with native methods for most of the HDF5 functions.

How to use it

The JHI5 is used by Java classes to call the HDF5 library, in order to create HDF5 files, and read and write data in existing HDF5 files. For example, the HDF5 library has the function **H5Fopen** to open an HDF5 file. The Java interface is the class *hdf.hdf5lib.H5*, which has a method:

```
static native int H5Fopen(String filename, int flags, int access );
```

The native method is implemented in C using the Java Native Method Interface (JNI). This is written something like the following:

```
JNIEXPORT jint JNICALL Java_hdf_hdf5lib_H5_H5Fopen ( JNIEnv *env, jclass class, jstring hdfFile, jint flags, jint access)  {  /* ...convert Java String to (char *) */  /* call the HDF library */  retVal = H5Fopen((char *)file, (unsigned)flags, (hid_t)access );  /* ... */ }
```

This C function calls the HDF5 library and returns the result appropriately.

There is one native method for each HDF entry point (several hundred in all), which are compiled with the HDF library into a dynamically loaded library (*libjhdf5*). Note that this library must be built for each platform.To call the HDF `***H5Fopen***' function, a Java program would import the package '***hdf.hdf5lib.****'*, and invoke the method on the class '***H5***'. The Java program would look something like this:

```
import hdf.hdf5lib.*; {  /* ... */  try {  file = H5.Hopen("myFile.hdf", flags, access );  } catch (HDF5Exception ex) {  //...  }  /* ... */ }
```

The ***H5*** class automatically loads the native method implementations and the HDF-5 library.

JHI5 Design Notes

1. Overview

The **Java HDF5 Interface (JHI5)** is a Java package (**hdf.hdf5lib**) that ``wraps" HDF5 library. For general information about HDF5 file format and library, please see the HDF5 Home Page. The JHI5 may be used by any Java application that needs to access HDF5 files. This product cannot be used in most network browsers because it accesses the local disk using native code.

What it is

A central part of the JHI5 is the Java class ***hdf.hdf5lib.H5***. The H5 class calls the standard (*i.e.*, `native' code) HDF5 library, with native methods for most of the HDF5 functions. In general, there is one native method call for each function in the HDF5 API, with similar arguments. Consult the HDF5 reference manual for details of the C API.

The Java HDF5 Interface consists of Java classes and dynamically linked native libraries. The Java classes declare native methods, and the library contains C functions which implement the native methods. The C functions call the standard HDF5 library, which is linked as part of the same library on most platforms. The Java HDF5 Interface also translates between Java and C arrays, and converts error codes from the HDF5 C library to Java Exceptions.

Intended purpose

The Java HDF5 Interface is intended to be the standard interface to access the HDF5 library from Java programs. The JHI5 is a foundation upon which application classes can be built. All classes that use this interface package should interoperate easily on any platform that has HDF5 installed.

It is likely that most Java programs will not want to directly call the HDF5 library. More likely, data will be represented by Java classes that meet the needs of the application. These classes can implement methods to store and retrieve data from HDF5 files using the Java HDF5 library.

It is important to note that, unlike serialized Java objects, files written with the JHI5 are completely compatible with HDF5 files in any language. Using the JHI5, Java programs can exchange data with C, C++, and Fortran programs.

What It Isn't

It is extremely important to emphasize that *this product is **not** a pure Java implementation of the HDF5 library.* The JHI5 calls the same HDF5 library that is used by C or FORTRAN programs.

The JHI5 is *not* a high level model of data or storage. The JHI5 is the Java interface to the HDF5 library API. High level object models implemented in Java will use the JHI5 to store and retrieve objects using HDF5.

The JHI5 is *not* a persistent object store for Java objects. It would be possible to implement an object store with the JHI5, but the interface does not provide any special support for storing and retrieving Java objects.

2. How to use the JHI5

How it works

The JHI5 is used to call the HDF-5 library. The Java application will make essentially the same calls as a C program. The HDF-5 library is accessed through the Java class ***hdf.hdf5lib.H5***.

For example, the HDF5 library had the function **H5Fopen** to open an HDF5 file. The Java interface is the class ***hdf.hdf5lib.H5***, which has a method:

```
public native int H5Fopen(String name, int flags, int access_id) throws HDF5LibraryException,
NullPointerException;
```

The native method is implemented in C using the Java Native Method Interface (JNI). The native method implementation is written something like the following:

```
JNIEXPORT jint JNICALL Java_hdf_hdf5lib_H5_H5Fopen (JNIEnv *env, jclass class, jstring name,
jint flags, jint access_id) { hid_t status; char* file; jboolean isCopy; if (name == NULL) {
/* exception -- bad argument? */ nullArgument( env,"H5Fopen: name is NULL"); return -1; } file
= (char *)(*env)->GetStringUTFChars(env,name,&isCopy); if (file == NULL) { /* exception -- out
of memory? */ JNIFatalError( env,"H5Fopen: file name not pinned"); return -1; } status =
H5Fopen(file, (unsigned) flags, (hid_t) access_id );
(*env)->ReleaseStringUTFChars(env,name,file); if (status < 0) { /* throw exception */
libraryError(env); } return (jint)status; }
```

Note that this C function calls the HDF5 library and returns the result appropriately.

Essentially, there is one native method for each HDF5 entry point (but please read "What does not work", below), which are compiled with the HDF5 library into a dynamically loaded library (***libhdf5***). Note that, while the Java classes may be used on any platform without recompilation, the JHI5 C library must be built for each platform.

How to call it

To call the HDF5 `***H5Fopen***' function, a Java program must import the package '***hdf.hdf5lib.****', and invoke the '***H5Fopen***' method. The Java program would look something like this:

```
import hdf.hdf5lib.*; /* ... */ try { // access an HDF5 file using the HDF5 library file =
H5.H5Fopen("myFile.h5", HDF5Constants.H5F_ACC_RDWR, HDF5Constants.H5P_DEFAULT); } catch
(HDF5Exception ex) { System.err.println(e); } /* ... */
```

The **H5** class automatically loads the native method implementations and the HDF5 library itself.

Parameter passing conventions

The Java HDF5 interface follows the HDF5 C interface as closely as possible. However, Java does not support pass-by-reference parameters, so all parameters that must be returned to the caller require special treatment for Java. In general, such parameters are passed as elements of an array. For example, to return an integer parameter, the Java native method would be declared to use an integer array. For instance, the C function

```
void foo( int inVar /* IN */, int *outVar /* OUT */ )
```

would be implemented in the JHI5 as:

```
native void foo( int inVar, int []outVar )
```

where the value of '*outVar*' would be returned as '*outVar[0]*'.

Data conversion and copying

The Java HDF5 Interface translates data between C data types and Java data types. For instance, when a Java program reads a two dimensional array of floating point numbers (**float [][]**), the HDF5 native library actually returns an array of bytes. The Java HDF5 Interface converts this to the correct Java object, and returns that to the calling program. This process uses the Java Core Reflection package to discover the dimensions and type of the array, and then calls native code routines to convert the bytes from native (C) order to the correct Java object(s).

The Java program can read and write multidimensional arrays of Java numeric types, there is no need for the calling program to convert the Java data types. This data conversion clearly adds overhead to the program, and for that reason some programs may prefer to manage this conversion. In this case, the Java program itself may convert data to and from an array of **byte**, which is passed directly to HDF5.

```
// the following call will read data from the HDF5 file into the // Java array of Float,
converting from C into the appropriate Java // objects.

Float[][] myData = new Float[100][200];

H5.H5Dread( dataset, memtype, memspace, filespace, myData );
```

The automatic conversion provided by the Java HDF5 Interface assures correct interoperability with any HDF5 file from any language. If the application program manages the conversion itself, the application must be responsible for assuring that the bytes written will produce the results expected when read by a C program.

Some types of data must be used with care. Please see the "What may not work", below.

HDF5 Constants

The HDF5 API defines a set of constants and enumerated values. Most of these values are available to Java programs via the class ***HDF5Constants***. In the example above, the parameters for the ***H5Fopen()*** call include two numeric values, ***HDFConstants.H5F_ACC_RDWR*** and ***HDF5Constants.H5P_DEFAULT***. As would be expected, these numbers correspond to the C constants ***H5F_ACC_RDWR*** and ***H5P_DEFAULT***.

The HDF5 API defines a set of values that describe number types and sizes, such as *H5T_NATIVE_INT* and *h5size_t*. These values are determined at run time by the HDF5 C library. To support these constants, the Java class ***HDF5CDataTypes*** looks up the values when initiated. The values can be accessed as public variables of the Java class, such as:

```
int data_type = HDF5CDataTypes.JH5T_NATIVE_INT;
```

The Java application uses both types of constants the same way, the only difference is that the ***HDF5CDataTypes*** may have different values on different platforms.

Error handling

The HDF5 error API (H5E) manages the behavior of the error stack in the HDF5 library. This API is omitted from the JHI5. Instead, errors are converted into Java exceptions. This is totally different from the C interface, but is very natural for Java programming.

The exceptions of the JHI5 are organized as sub-classes of the class ***HDF5Exception***. There are two subclasses of ***HDF5Exception: HDF5LibraryException*** and ***HDF5JavaException***. The sub-classes of the former represent errors from the HDF5 C library, while sub-classes of the latter represent errors in the JHI5 wrapper and support code.

The super-class ***HDF5LibraryException*** implements the method '***printStackTrace()***', which prints out the HDF5 error stack as described in the HDF5 C API ***H5Eprint()***, followed by the standard Java stack trace. This may be used by Java exception handlers to print out the complete error stack.

For more information about JHI5 exceptions see, Exceptions.

3. What definitely works

The JHI5 Interface supports the writing and reading of multi-dimensional arrays of numbers. Any Java number type can be used (both intrinsic and object types, i.e., both **float** and **Float**), and **Strings.** The Java HDF5 Interface must translate data between C data types and Java data types and vice versa. For instance, when a Java program reads a two dimensional array of floating point numbers (**float [][]**), the HDF5 native

library actually returns an array of bytes in C order. The Java HDF5 Interface converts this to the correct Java object(s), and returns the Java object to the calling program. Similarly, the Java program will pass an array of Java numbers, which is converted to an array of bytes in correct C order, and then passed to HDF5.

This process uses the Java Core Reflection package to discover the shape and type of the array, and then calls native code routines to convert the bytes from native (C) order to the correct Java object(s). This data conversion is invisible to the calling program, and assures complete data compatibility with other languages. Obviously, there may be a performance penalty imposed by this data manipulation, which will depend on the platform and the implementation of the Java Virtual Machine.

The JHI5 supports creating HDF5 files, creating HDF5 Groups, Datasets, Datatypes, and Dataspaces. Attributes can be created for Groups and Datasets. Similarly, objects can be accessed from any HDF5 file.

The JHI5 also supports creating and accessing of arrays of compound data types (i.e., structured records of heterogeneous data type). Java objects and HDF5 Compound Datatypes are not completely compatible, but it is usually possible to map between them on a case by case basis. See "What may not work", below and Using Compound Datatypes.

The JHI5 supports reading and writing selections, including hyperslabs, repeated blocks (strides), and point selections. These work as expected for Java arrays of any dimension and any number type. HDF5 "chunking" works as expected. Compression can be used, providing that the appropriate compression libraries are linked with the HDF5 library. (See "What may not work", below.)

4. What does not work

The current implementation of the Java HDF5 interface provides most of the functions of the HDF5 library. However, some features of HDF5 could not be supported for Java.

C function arguments

Several HDF5 APIs require pointers to functions as parameters, through which the user application passes a code to be called by the HDF5 library. This feature cannot be used by Java programs, and are omitted from the JHI5. In a few cases, we have provided alternative methods to provide important missing functions.

The HDF5 library supports extensions to add new compression schemes. GZIP is provided as a standard part of the HDF5 library, but other compression methods may also be linked and used. To use a compression method, a C program must "register" it with the library, passing a pointer to the "driver" which implements the compression scheme.

In the current implementation of the Java HDF Interface, there is no way to "register" compression libraries from a Java program. The consequence of this is that, in the event that data is compressed by a C program with an alternative compression scheme, a Java program will not be able to access that data.

Parallel APIs

The HDF5 parallel APIs are not implemented in the JHI5. Precisely how MPI-IO can and should be used for Java applications is still under study. The parallel API includes:

1. *H5Pset_mpi*
2. *H5Pget_mpi*
3. *H5Pset_xfer*
4. *H5Pget_xfer*

Variable Length Array

The HDF5 Variable Length APIs are not fully supported by the JHI5. The following functions are not implemented:

1. *H5Pset_vlen_mem_manager*
2. *H5Pget_vlen_mem_manager*

5. Warnings: What May or May Not Work

The HDF5 library is very general and flexible, and the library is correspondingly complex. The Java HDF5 Interface provides the key features of the HDF5 library, but some features are only partially supported, or must be used with care.

Strings

Java stores strings as objects, representing the string as a variable length of Unicode characters. In the initial releases, HDF5 supports strings as fixed or variable length arrays of characters, and understands how C and Fortran store strings. Additional support for strings, including Unicode may appear in later versions of the HDF5 format and library.

The JHI5 attempts to map Java **String** objects to and from HDF5 strings by converting to and from C strings (zero terminated arrays of ASCII characters). For most purposes, this works as expected. However, it should be realized that it is possible to write Java **Strings** to HDF5 in many ways--e.g., as zero-terminated arrays, as zero padded arrays, as arrays of 16-bit Unicode characters, etc. These may or may not be easy for a reader to realize that it is intended to be read as a **String**, or what handling may be required. In any case, strings written by other languages may

require careful processing in order to construct the appropriate Java **String**. For instance, strings written from Fortran will typically be fixed length arrays of ASCII, padded with zeros. These can be read into a Java **String** object, and the padding can be removed by the '*trim()*' method.

As a general rule, users are advised to use care when reading and writing strings.

User Defined Datatypes

The HDF5 format and library allows applications to define almost any kind and combination of numerical data types. For instance, user programs can control the byte order, number of bits (e.g., 5-bit integers), and the layout of floating point numbers. The JHI5 fully supports these features, allowing the definition and discovery of such user defined datatypes. However, the Java language only supports a few standard number types, and it is up to the Java application programmer to create appropriate object types to represent user defined datatypes, and to convert between the stored representation and whatever Java objects are implemented.

Data Types

HDF5 allows programs to specify the storage format of data when it is written to disk and when it is read from disk to memory, and handles conversions as needed. The Java language itself specifies a single standard layout for numeric data, which may be different from the "native" machine formats. however, the Java Native Interface converts between the 'native' machine layout and Java. For example, the JNI routine *GetFlo atArrayElements* converts a Java **float[]** object to an appropriate C array of floats, copying and transforming the data if necessary. (See the JNI d ocumentation.)

As a consequence, Java programs are always writing and reading native C data to and from the HDF5 library. This means that the datatypes ***'H5 T_NATIVE_*'*** work essentially the same as for a C program on the same platform.

Programmers should be aware that when data is read (written) into the Java HDF5 Interface, it may actually be transformed twice: the JNI may swap the bytes from Java to native C (native C to Java), and then the HDF5 library may swap the bytes again from native C to the specified storage order (storage to C). The HDF5 library may also convert the data, e.g., from **short** to **int** or **int** to **float,** when such a transformation is specified. This can be confusing because while the layout of the Java numbers is the same on all platforms, the HDF5 Datatype is specifying the relationship between the native values and the HDF5 file, which depends on the platform.

Fortunately, in most cases the datatypes for C can be used in Java to get the correct results. E.g., data from a Java Integer can be written as ***'H5 T_STD_32BE'***, and read back as ***'H5T_NATIVE_INT'***. This will give correct results on all platforms in either C or Java. The one exception to this is Java **Long** and **long** integers. These are defined to be 64-bit integers in Java, but may be different sizes on different C environments--in fact, a 'long' is often 32-bits long. The HDF5 type ***'H5T_NATIVE_LONG'*** refers to the size and byte order of a '**long**' according to the C compiler used at the time the library is compiled, which is typically 32-bits.

To correctly read and write Java long or Long data, the HDF5 data type must be **H5T_STD_64BE** or **H5T_STD_64LE**, i.e., 64-bit big endian or little endian. Either may be selected, but an appropriate byte order must be specified to read the data correctly. However, in the current implementation this data cannot be read with the C type **H5T_NATIVE_LONG**. To read on a platform with the same byte order, the data must be read with the same H5T type. To read on a platform with a different byte order, the data must be read with the opposite byte order.
In general, *64-bit data should be used with caution, and results checked carefully to make sure that data is correct*.

Compound data types

The HDF5 format and library support complex structured data types, including arrays of "compound data types", i.e., arrays of structured records. Compound data is stored as packed arrays of bytes, as C stores 'struct' data. The Java language does not use a flat memory model; data--including arrays--are stored as objects. An array of Java objects is not necessarily stored as a contiguous block of storage, and contains information besides the content of the variables. For these reasons, it is very difficult to automatically map between HDF5 compound data types and Java objects.

The JHI5 supports defining, writing, and reading compound data in HDF5 files through two mechanisms. First, an array of compound data can be read or written as appropriately packed arrays of bytes. A Java application may create an array of bytes of the appropriate size, fill it with data to match the intended layout (i.e., as a C struct would be laid out), and pass that to the HDF5 library. Data may be read similarly, and the Java application must interpret the bytes to create the intended Java objects. *In this case, it is up to the Java programmer to create the correct bytes which correspond to how the C compiler would lay out the structure in memory.*

Alternatively, data may be written and read by individual 'elements', i.e., by the fields of the compound data type. For example, for an HDF5 compound data type with one 'int' and one 'float' element, the 'int' data dan be read into a Java array of **ints**, and the float element can be read into a second array of Java **floats**.

See "Using Compound Datatypes" for a detailed example.

Unsupported Functions

H5Ddebug	Low priority
H5Dgather	Function pointer
H5Dscatter	Function pointer

H5Eget_auto1	Function pointer
H5Eget_auto2	Function pointer
H5Epush1	Function format with list
H5Epush2	Function format with list
H5Eset_auto1	Function pointer
H5Eset_auto2	Function pointer
H5FDalloc	Function pointer
H5FDclose	Function pointer
H5FDcmp	Function pointer
H5FDflush	Function pointer
H5FDfree	Function pointer
H5FDget_eoa	Function pointer
H5FDget_eof	Function pointer
H5FDget_vfd_handle	Function pointer
H5FDquery	Function pointer
H5FDread	Function pointer
H5FDregister	Function pointer
H5FDset_eoa	Function pointer
H5FDtruncate	Function pointer
H5FDunregister	Function pointer
H5FDwrite	Function pointer
H5Fget_file_image	Function pointer
H5Fget_mdc_config	Function pointer
H5Fget_vfd_handle	Function pointer
H5Fset_mdc_config	Function pointer
H5Giterate	Function pointer
H5Iregister	Function pointer
H5Iregister_type	Function pointer
H5Iobject_verify	Function pointer
H5Iremove_verify	Function pointer
H5Isearch	Function pointer
H5Lcreate_ud	Function pointer
H5Lregister	Function pointer
H5Lunpack_elink_val	Function pointer
H5Pcreate_class	Function pointer
H5Pget_elink_cb	Function pointer
H5Pget_multi_type	Low priority

H5Pget_type_conv_cb	Function pointer
H5Pget_vlen_mem_manager	Function pointer
H5Pinsert1	Function pointer
H5Pinsert2	Function pointer
H5Piterate	Function pointer
H5Pregister1	Function pointer
H5Pregister2	Function pointer
H5Pset_driver	Function pointer
H5Pset_dxpl_mpio_chunk_opt	Function pointer
H5Pset_dxpl_mpio_chunk_opt_num	Function pointer
H5Pset_dxpl_mpio_chunk_opt_ratio	Function pointer
H5Pset_dxpl_mpio_collective_opt	Function pointer
H5Pset_elink_cb	Function pointer
H5Pset_filter_callback	Function pointer
H5Pset_multi_type	Low priority
H5Pset_type_conv_cb	Function pointer
H5Pset_vlen_mem_manager	Function pointer
H5Tfind	Function pointer
H5Tregister	Function pointer
H5Tunregister	Function pointer
H5Zregister	Function pointer