Java Native Interface for HDF Version 3

This document explains the general principles of using the JNI to link C libraries and the specific requirements for linking to HDF4 and HDF5. The JNI specification and code is standard for all platforms, but compiling and runtime linking are different for each virtual machine and platform. These issues are complex and not well documented.

---

1. Overview

These packages "wrap" the HDF libraries using the Java Native Method Interface (JNI). There are two wrappers, one for the HDF4 library (JHI) and one for the HDF5 library (JHI5).

Package	Description
Java HDF Interface (JHI)	Classes and JNI code to access the **HDF4** library.
Java HDF5 Interface (JHI5)	Classes and JNI code to access the **HDF5** library.

It is extremely important to emphasize that ***these packages are not a pure Java implementation of the HDF libraries***. The JHI or JHI5 calls the same HDF libraries that are used by C or FORTRAN programs.

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

Each Java HDF Interface is intended to be the standard interface for accessing HDF files from Java programs. The JHI is a foundation upon which application classes can be built, such as the HDF Object package.

---

2. The Java Native Interface (JNI)

The Java language defines its own virtual machine and instruction set. This design makes Java easy to program and very portable. However, Java programs cannot use code written in any language except Java. In order to provide a minimal level of interoperability with other languages, the Java Native Interface (JNI) defines a standard for invoking C and C++ subroutines as Java methods (declared 'native'). The Java Virtual Machine (JVM) implements a platform-specific mechanism to load C or C++ libraries, and to call the native methods on behalf of the Java programs.

So, to access a C library from a Java program, four tasks must be completed:

- Declare appropriate 'native' methods in one or more Java classes
- Write 'glue' code for each native method to implement the JNI C or C++ standard
- Compile the JNI glue (C or C++) to create a JNI library
- Deploy the Java classes and the JNI library on the system

When the Java program runs, the JNI glue and C library must be in the 'java.library.path' path.
The JNI standard defines the syntax for the Java native methods, and the C and C++ for the glue layer. This standard is universal, the same code should work on any platform.

Compiling the JNI is specific to platform, compiler, and Java Virtual Machine. The challenge is to create a C library that can be dynamically loaded by the JVM. On each platform, specific flags must be used with the C or C++ compiler, to create a library that the Java VM can successfully load. The same C code is used on all platforms, but the build process is different for each platform.

The rules for deployment are roughly standard across all platforms, i.e., the JNI libraries must be in the library load path for the Java application. However, the precise behavior of the Java VM is not standardized and, indeed, varies across platforms, and even different versions of a JVM on the same platform.

2.1. Design Limitations of the JNI

Mixing arbitrary C code with the execution of a Java Virtual Machine (JVM) is a difficult challenge, and the JNI necessarily has limitations.

First, the JVM is a multi-threaded program. The C code in a C library will be executed in a thread, with multiple threads active. The C code must be thread safe, and in some cases may need to synchronize to prevent disastrous bugs. However, *the precise requirements on the C code depend on the implementation of the specific Java VM*. Not only is there no general specification for the C code, the thread-safety requirements are different for different Java VMs--and in most cases there is no way to know what the C code must do.

Second, the JNI layer is vulnerable to errors because neither Java nor C can check the correctness of Java-to-C interface. Also, any errors in the

C code can create grave and mysterious crashes in the JVM. In particular, if the C code overwrites memory that it shouldn't, it can corrupt the state of the JVM, producing indecipherable errors in the Java program.

Third, the JVM dynamically links to C or C++ libraries. This process is similar to the way that dynamic libraries are implemented for C programs, but the *Java VM does not necessarily follow the same rules as the C runtime*. The precise requirements on the library and behavior of the loader is specific to the JVM, and different platforms exhibit different behavior. Furthermore, in order to be used throughout the JNI, the C library (and all its dependencies) must be compiled and linked with very specific C compiler and linker settings.

2.2. Resolving References in JNI libraries

The JVM is dynamically linked to C or C++ libraries. This process is similar to the way that dynamic libraries are implemented for C programs, but the *Java VM does not follow the same rules as the C runtime*.

The native library is linked when a native method is invoked (i.e., on demand) or when the library is specifically invoked (i.e., the first access to the *hdf.hdflib.HDFLibrary* or *hdf.hdf5lib.H5* class) with a call to *System.loadLibrary("hdf")* or *System.loadLibrary("hdf5")*. If successful, the link step is invisible to the application and user.

This link step will fail if:

- the library is not found in the paths configured at runtime
- the library is not compatible with the JVM
- a required symbol cannot be resolved

If the link fails, the application will receive an exception, such as *java.lang.UnsatisfiedLinkError* or *java.lang.NoClassDefFoundError*.

Note that it may be difficult to determine the cause of the link failure: it might be a path problem, it might be a code generation problem (e.g., incompatible settings for libhdf.so), or it might be a missing or mis-matched symbol.

The JVM usually does not implement dependency checking and dynamic loading for the C libraries that it loads. Specifically, if the JNI library has unresolved symbols, e.g., to another dynamic library, the JVM may or may not load that library. And if multiple C libraries with mutual dependencies are manually loaded into the JVM, they may or may not be linked to resolve the references in the C code. In most JVMs, these dependencies are *not* implemented.

Also, some JVMs check all references at load time, rejecting the library if any symbols are unresolved. Other JVMs will load a library with unresolved symbols, and fail only when one of the missing symbols is referenced. In other words, the linking exception may occur differently with different JVMs.

---

3. The HDF Libraries

The HDF Java products are built on top of the JNI, to implement native interfaces to the HDF4 and HDF5 libraries. Essentially, there is a single Java object for the HDF4 library (*hdf.hdflib.HDFLibrary*), with a native method for each entry point to HDF4. A second Java object (*hdf.hdf5lib.H5*) wraps the HDF5 library.
The HDF4 and HDF5 libraries present significant challenges for the JNI. These libraries are large, complicated C programs, with hundreds of entry points, complex arguments, C-dependent behaviors (e.g., C subroutines with function pointers), and dependencies on multiple external libraries. Because of these challenges, the HDF JNI interface is not ideal, and can be difficult to compile, and can be difficult to run.

Furthermore, the HDF Native Interface is not available on some platforms because the technical mysteries have not been successfully resolved.

3.1. Thread Safety

The Java VM is a multi-threaded application, and therefore the HDF library will be called from a thread in a multi-threaded program. It is far from clear what all the implications of this fact may be, but both HDF4 and HDF5 are not thread-safe, so calls to the library should be serialized.

The HDF5 library has a "thread safe" option. *This option is irrelevant for the JNI and should not be used.* The HDF5 thread safe option depends on P-threads, but the JVM uses it's own thread package, which usually isn't p-threads. More important, it is impossible to correctly synchronize from inside the HDF5 library.

Following the JNI specification, the recommended procedure is to use Java synchronization to create a monitor to serialize access to the library. This is done in the Java code by declaring the methods *'synchronized'*. This design essentially puts a single lock around the HDF library. This is equivalent to the HDF5 thread safe option, except it is implemented by the JVM at a higher level in the call stack.

3.2. The HDF JNI Layer

The HDF4 and HDF5 libraries have hundreds of entry points, with many types of arguments, including all numeric types, Strings, and pointers to arrays. The HDF JNI layer must translate all parameters between Java and C, invoke the C API, and translate return values and errors to Java.

The overriding principle in this layer is that Java uses *objects,* and C uses *memory structures*. The memory layout of a Java object *cannot be accessed* through Java or the JNI. The translation can only be done by calling special C routines that return C data structures.
The HDF JNI layer is basically a series of calls that:

1. translate each input parameter from Java objects to C data structures
2. call HDF
3. translate the return values from C data structures into Java objects
4. clean up
5. return

The translation of data between Java and C sometimes requires allocation of memory and a copy of the data. The details depend on the implementation of the specific JVM. In general, Arrays, Strings, and C structures are copied, although the JVM may be able to optimize some of the copies.

These data copies may be very significant for the HDF native interface: when reading and writing large blocks of data, the data array may be in memory twice (in a Java object and in a C array), and data may be copied between the two copies. This can be a very serious performance penalty.

The C library returns error codes. The errors are detected in the JNI layer and Java exceptions are raised. This behavior is very natural. The calling Java program must surround the calls to the HDF libraries with appropriate try/catch expressions.
The HDF libraries define a large number of constants to be used as parameters to the API. The HDF JNI layer exports a set of Java constants that are mapped to the required C values. The Java program calls the native method hdf.hdf5lib.J2C to get the correct value for the C constants.

3.3. Compiling, Linking, External Dependencies

While the Java and C code is portable, the build process is platform and JVM specific. The build process is constrained by the dynamic loading of the JVM. For each platform, the JVM will load libraries built with certain compilers, with certain parameters. In general, the library must be a "dynamic" library, which must contain the JNI entry points (i.e., the C implementations of the 'native' methods), along with the code to implement the call.
In the case of the HDF libraries, this means that the JNI layer, the HDF library, and all libraries that HDF depends on must be loaded. Figure 1 gives a sketch of the dependencies. A Java program will access HDF4 through the class hdf.hdflib.HDFLibrary. When this Java class is loaded, the HDF4 JNI library is loaded. This library must contain:

- the JNI entry points, one for each native method
- the HDF4 library (libhdf, libmfhdf)
- the external compression required by HDF (libz, libjpeg, libsz)
- possibly other system libraries, e.g., libm

Similarly, HDF5 is accessed through hdf.hdf5lib.H5, which loads the HDF5 JNI:

- the JNI entry points, one for each native method
- the HDF5 library (libhdf5)
- the external compression required by HDF (libz, libsz)
- possibly other system libraries, e.g., libm

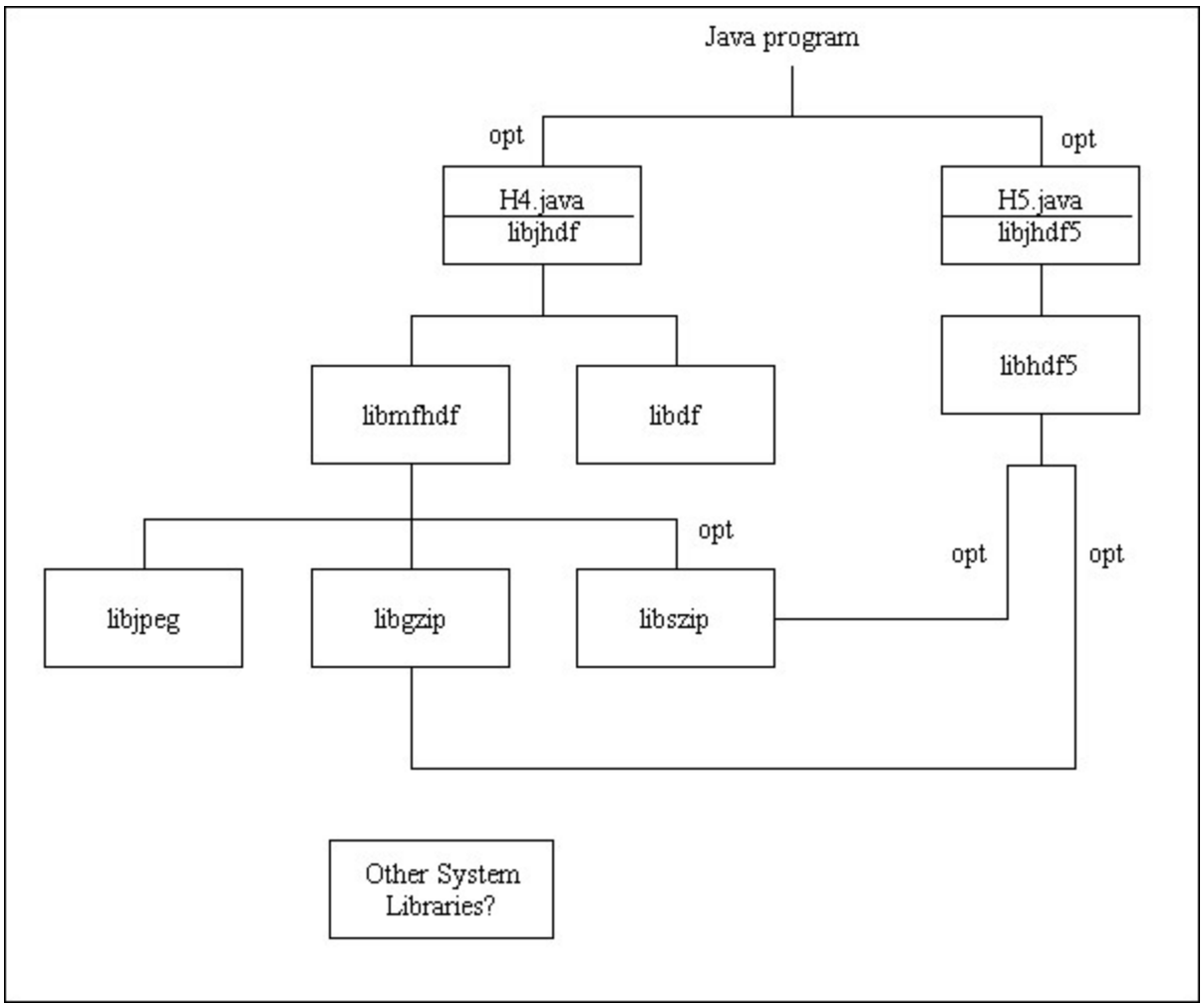

Figure 1. The HDF libraries and the libraries they depend on.

In general, the Java VM will load only one library (e.g., libhdf.so). This library must be built in a particular way, depending on the JVM. Note that this usually means that all the dependent code must also be built conformantly. This may or may not be possible in all cases. If one of the components cannot be built, the HDF JNI cannot be supported on that platform.
On windows, solaris, linux, and mac, the build procedure is to use a static version of the dependent libraries (libdf.a, libz.a, etc.), and then output a shared library. E.g., for solaris the HDF4 build process is:

- Build the JNI implementation classes, *.o
- link *.o with static libraries: libhdf.a, libmfhdf.a, libz.a, libjpeg.a, libszlib.a
- output a shared library, libjhdf.so

Note that it does not work to build with shared libraries: libhdf.so, etc.. The reason is that the file loaded by the JVM (libjhdf.so) must have all the symbols resolved, it cannot require dynamic linking to other libraries. (In other words, you have to build a dynamic library that does not depend on any other dynamic libraries.

Deployment, runtime

The Java HDF has two parts, the Java classes and the C libraries. The Java classes are executed with a Java VM, they must be found in the CLASSPATH just as any Java classes. The C libraries are loaded when the Java classes for the JNI are initialized. The C libraries must be in the library search path for the JVM, e.g., LD_LIBRARY_PATH.

The most common problem is difficulty running the code after compilation. This is usually caused by problems with the class path or library path. When one of the jar files is missing from the class path, the program will encounter a "Class not found" exception. An "unsatisfied link" error will occur if the library with the native implementation is not found in the LD_LIBRARY_PATH.