Java and other Managed Bindings to Kakadu

David Taubman, UNSW

July 21, 2006

1 Brief Summary

The Kakadu software framework is implemented in C++ using a fairly rigorous object oriented design strategy. All classes which are intended for public use are documented by comments in public header files, using a set of comment conventions which allows the "*kdu_hyperdoc*" utility to build high quality HTML documentation describing the classes and their interaction with each other. The same comments may embed special directives which allow "*kdu_hyperdoc*" to construct Java native interfaces to the classes in a meaningful and efficient manner, along with interfaces to other languages, including C# and Visual Basic.

Virtually all high level Kakadu classes and virtually all of their member functions can be presented to Java and other foreign language applications using these tools. Moreover, by following the same conventions, you can get "*kdu_hyperdoc*" to build foreign language bindings to your own C++ classes, which might wrap up certain subsets of the Kakadu functionality efficiently. "*kdu_hyperdoc*" also integrates its documentation of the Java and other foreign language interfaces which it builds with the native C++ HTML documentation.

In addition to providing "downward" calls into the Kakadu system from Java and other languages, "*kdu_hyperdoc*" allows "upward" calls from the native implementation back up into Java, C# or Visual Basic implementations of objects which use the foreign language bindings. In particular, any native Kakadu virtual function which designated as a "callback" function using the "[BIND: callback]" syntax in its description, causes code to be constructed which calls the functions of an implementing class in the foreign language. Amongst other things, this capability is currently used to bring all Kakadu system messages into Java, C# or Visual Basic in an elegant manner.

Since Kakadu emphasizes robustness, considerable attention has been paid to exception handling in building foreign language bindings. Exceptions can be thrown and caught across the Kakadu native interfaces, including when those exceptions are thrown inside callback functions implemented in Java, C# or Visual Basic. The principle source of exceptions is generally the code which handles Kakadu system errors, which can and often should be implemented in Java for Java-based applications, C# for C#-based applications, and so forth.

2 Purpose

One reason for building foreign language interfaces is to avoid platform dependencies in the implementation of certain applications. Although the native Kakadu implementation must be compiled for the platform of interest and shipped as binaries, the vast majority of the Kakadu system (the core system, plus most application specific add-ons) should compile on virtually any platform. However, graphical user interfaces and networking elements of an application must often be implemented differently for different platforms. You can now implement these elements in Java.

Many of our licensees have asked us about Java interfaces to the Kakadu system, because they like the features and efficiency of the Kakadu implementation, but prefer to build systems based around Java. The new Java interfaces are intended to fulfill the needs of these customers.

Other licensees have asked for the Java interfaces to be extended to C# and Visual Basic. These are fully supported from Kakadu version 5.2.

3 Getting Started

First, you will need to compile the foreign language bindings. The process for doing this is described in the file "*Compiling_Instructions.txt*". From Kakadu version 5.2, the build process has been made much more straightforward. In brief, the typical steps are:

1. Compile and run the "*kdu_hyperdoc*" tool. For Unix platforms, this is done by the relevant makefile in "*apps/make*." For Windows platforms, it is done using the .NET or Visual C++ (version 6) worskpace files, respectively named "*apps/apps.sln*" or "*apps/apps.dsw*."

2. Build the interface DLL's or shared libraries – these are built from the code constructed by "*kdu_hyperdoc*." For Unix platforms, this is done by the relevant makefile in "*managed/make*." For Windows platforms, everything is done by the .NET workspace file, "*managed/kdu_managed.sln;*" if you only have Visual C++ (version 6), use "*managed/kdu_managed.dsw*" to build the Java native interfaces, but you will not be able to build C# or Visual Basic interfaces, which are founded on .NET.

3. Make sure all of the DLL's or shared libraries built using the above process are included in the load path. For Unix users this is the path defined by the "`LD_LIBRARY_PATH`" environment variable. For Windows users, the load path is defined by the "`PATH`" environment variable, shared by both executables and dynamic libraries.

4. For Java bindings, you must also compile the "`.java`" files found in the "*java/kdu_jni*" directory (the "*java*" directory is a sibling, rather than a child of your Kakadu source distribution directory). One way to do this is to invoke the standard tool, "`javac`" using a command like

"`javac *.java`"

issued from a shell or command prompt inside the "*java/kdu_jni*" directory. Don't forget to include the resulting class files in the Java load path by adding the "*java*" directory (use its full path name) to the "`CLASSPATH`" environment variable.

You can then go ahead and look through (and build) either of the Java example applications found in the "*managed/java_samples*" directory, and/or the corresponding C# examples found in the "*managed/csharp_samples*" directory. The corresponding Java and C# examples do exactly the same thing. Java examples may be compiled using "`javac`" as usual, whereas the C# examples are compiled for you from the "*managed/kdu_managed.sln*" .NET workspace.

4 Kakadu Calls in Java

After a few initial tips, most of the Java native calls should become pretty self explanatory.

Firstly, since there are no global functions, #defines, or enum types in Java, these are all lumped into a single Java class known as "*Kdu_global*", which contains only static functions and static, final variable declarations. There are not many global functions in Kakadu, but there are quite a few constants, which you will be glad to find available in "*Kdu_global*". Only those constants for which "*kdu_hyperdoc*" was able to find a type binding are included there, but all the constants you need should be of this form, due to the way the Kakadu header files have been written. For example, Kakadu header files define a constant like `HH_BAND` as

```
#define HH_BAND ((int) 3)
```

rather than just

```
#define HH_BAND 3
```

This gives "*kdu_hyperdoc*" the information it needs to include the definition in the "`Kdu_global`" Java class.

The second convention you will want to note is that C++ class names are mapped to Java class names by capitalizing their first letter (I always use strictly lower-case names with underscores for my C++ names).

Similarly, all function names are mapped to Java by capitalizing the first letter in the C++ function. Although this is not a recommended Java convention, it helps to avoid the accidental use of names with reserved meaning in Java. In particular, Java assigns a special meaning to class members with the name "*finalize*", while Kakadu has a number of classes with their own "*finalize*" functions. These become "*Finalize*" in Java.

The Java interfaces are pure function-only interfaces. There are no public member variables, except for the "*native_ptr*" member in every exported class – you should never modify this from Java.

Some functions do not bind well to Java. Among these are the C++ operator overloads, for which there is no equivalent in Java. To overcome these difficulties, some extra functions have been added to some of the Kakadu classes, to provide alternatives which do bind well. Extra functions have also been added to the small number of publically accessible Kakadu classes (or structs) which offer public member variables. These extra functions allow the underlying member variables to be manipulated safely across the Java bindings.

The "*kdu_hyperdoc*" tool maps unsigned 32-bit integers to Java `long`, and maps unsigned 16-bit integers to Java `int`. Signed quantities are mapped in the most natural fashion, with 16, 32 and 64-bit integers mapped to Java's `short`, `int` and `long`, respectively. While these conventions are reasonable, they may prevent the disambiguation of overloaded C++ functions. In these cases, you must explicitly mark one or more of the offending C++ functions as having no Java binding. Do this by including the directive "`[BIND: no-java]`" or "`[BIND: no-bind]`" in the comment description following the function's declaration. This has already been done in all the standard Kakadu header files, but you may wish to use the tool to build Java bindings to your own native classes and functions.

One of the complications which can arise when building Java native interfaces is that object destruction is not explicit in Java. This can damage memory efficient implementations, and also prevents us from controlling the order of destruction, which is very important for certain Kakadu objects. To overcome this difficulty, we explicitly provide "*Native_destroy*" members for all Java objects which are bound to C++ objects that can be destroyed. The "*Native_destroy*" function will destroy the internal C++ representation, after which you should also throw away your reference to the Java object – its "*native_ptr*" member will become `null`. If you override the "*Native_destroy*" function in a derived Java class, be sure to call "*super.Native.destroy*" from your derived version.

One thing which might puzzle you at first is that "kdu_error" and "kdu_warning" objects do not have Java native bindings (you will note the absence of the "`[BIND: reference]`" tag from the descriptions of these classes in "*kdu_message.h*"). This is deliberate, because these objects must never be created on the heap, only on the stack as local variables, and the Java bindings create all objects on the heap. The reason why these objects should never be created on the heap is that their destructors never complete – they either exit the process or throw an exception. Nevertheless, the absence of these functions should cause no problem for you, because we have two new global function calls, "*kdu_print_error*" and "*kdu_print_warning*", which create "*kdu_error*" and "*kdu_warning*" objects internally on the stack and destroy them immediately after delivering the relevant message string. These functions bind well to Java and work well with Java's excellent string construction syntax. You may also find these functions convenient alternatives for your C++ development.

Be sure to refer to the HTML documentation rooted at the "*index.html*" file in the "*documentation*" sub-directory. If you build the interfaces and documentation in the manner recommended in the "*Compiling_Instructions.txt*" file, all Java functions will be documented along with their C++ counterparts. This allows you to rapidly identify which functions are available in Java and which are

not. It also allows you to see how objects, arrays, reference arguments, etc., are passed across the Java interfaces. You will soon get the hang of our argument binding conventions.

5 Kakadu Calls in C# and Visual Basic

Bindings to the popular yet Microsoft-specific languages, C# and Visual Basic, are provided through a set of managed native interfaces, implemented in Microsoft's so-called "Managed Extensions to C++." These are implemented through the DLL "*kdu_mni.dll*," which uses both the core system DLL "*kdu_vXXY.dll*" and an auxiliary DLL "*kdu_aXXY.dll*", where "XX" stands for the current Kakadu version and "Y" is one of "R" (for release) or "D" (for debug). These two DLL's are also shared with the Java native interface DLL, "*kdu_jni.dll*." To access the Kakadu API from your C# or Visual Basic application, you have only to add "*kdu_mni.dll*" to the list of references in the corresponding .NET project. This DLL should appear in the "*bin*" directory which is parallel to your current Kakadu distribution directory. Alternatively, add a reference to the "*managed/kdu_mni/kdu_mni.vcproj*" project.

Actually, there are other Microsoft managed languages other than C# and Visual Basic, which can use the managed Kakadu bindings in "*kdu_mni.dll*." However, we currently provide syntax in the HTML documentation generated by "*kdu_hyperdoc*" only for C# and Visual Basic. Also, to access global variables, you will need to add the "*kdu_constants.cs*" or "*kdu_constants.vb*" file found in "*managed/kdu_mni*" to your project. The "*kdu_hyperdoc*" tool does not currently build corresponding constants files for other languages.

All classes and functions which can be bound to Java can also be bound to C#. In fact, the binding conventions are identical to those for Java, with only the following exceptions.

1. The class names in C# and Visual Basic are formed by prepending a "C" to the native C++ class name, rather than capitalizing the first letter in the class name. Thus, "*kdu_region_compositor*" in C++ becomes "*Kdu_region_compositor*" in Java and "*Ckdu_region_compositor*" in C# and Visual Basic.

2. The function names in C# and Visual Basic are identical to those in C++, without any capitalization of the first letter.

3. All global functions are found inside a class named "*Ckdu_global_funcs*" which is inherited by a class named "*Ckdu_global*" defined in "*kdu_constants.cs*" or "*kdu_constants.vb*," as appropriate. The latter class provides definitions for all exported constants. In this way, "*Ckdu_global*" is completely equivalent to the Java class "*Kdu_global*."

4. Wherever a native Kakadu C++ class or structure provides a public member variable with a corresponding "*get_*`xxx`" or "*set_*`xxx`" function, where

`xxx` stands for the name of the member variable, the and/or set function are bound as property accessors for the purpose of C# and Visual Basic language bindings. This means that you do not call the get or set function directly from C# or Visual Basic; instead, your code simply uses the member variable as though it were directly accessible. Behind the scenes, the relevant get or set function is called for you. For example, if `coords` is an object of type *Ckdu_ coords* and `yval` is an integer, you can directly assign "`yval = coords.x;`" the corresponding code in Java reads "`yval = coords.Get_x();`" Property accessors are clearly identified in the HTML documentation generated by "*kdu_ hyperdoc.*"

5. Kakadu classes which are assigned a "*Native_ destroy*" function in Java (i.e., all objects whose C++ implementation provides a public destructor) are provided with a "*Dispose*" function in the managed bindings used for C# and Visual Basic. The semantics of this function follow exactly the same conventions used by Microsoft for all of its Windows component objects. While similar to the semantics of "*Native_ destroy*" in Java, there are some subtle differences, which are worth explaining:

   (a) The public "*Dispose*" function takes no arguments and is not virtual. This is the function you should call to delete the underlying native Kakadu object when you can't wait until garbage disposal. All Kakadu classes with public destructors, for which bindings are created, are assigned a "*Dispose*" function. Do not override this in any derived class.

   (b) A protected virtual function with the prototype, "*Dispose(bool in_ dispose)*" is also provided. This function gets called with its *in_ dispose* argument equal to true if called from the non-virtual argument-less "*Dispose*" function. During garbage collection, it is called with *in_ dispose* equal to false. This is the function which actually does the disposing. You can override this function in a derived class to perform additional cleanup tasks (typically disposing other Kakadu objects or Windows components). In this case, be sure to pass the call on to the base class's "*Dispose(bool in_ dispose)*" function, with the same value for the *in_ dispose* argument. Also, since the function might be called multiple times, you should take care that any overriding function cannot erroneously try to dispose other objects multiple times. For an example of an overridden "*Dispose(bool in_ dispose)*" function, see the "*Ckdu_ bitmap_ buf*" object in "*KduRender2.cs.*"

6 Building Your Own Foreign Language Bindings

You can put "*kdu_hyperdoc*" to work for you to build Java and other foreign language interfaces to your own classes. Simply include the relevant header file in the list supplied to "*kdu_hyperdoc*" and include the relevant binding directives in comments attached to your C++ class declaration – and optionally its member functions. There are quite a number of subtleties you might come across, but most of them should be pointed out to you by helpful messages from "*kdu_hyperdoc*" if you do something which will cause problems.

Class bindings recognized by "*kdu_hyperdoc*" fall into three categories, signalled by the "`[BIND: reference]`", "`[BIND: interface]`" and "`[BIND: copy]`" directives. The most useful and powerful of these is "`[BIND: reference]`", and you are recommended to use it everywhere, since the other bindings may not be used with derived classes. The "`[BIND: interface]`" option is used in special circumstances to avoid inefficient use of the heap when the C++ class is actually an interface to a separate internal C++ object. Classes specifying this binding must contain no virtual functions whatsoever, may not be derived, may have no destructors, and may contain no member variables other than a single memory pointer. The "`[BIND: copy]`" option is used only for simple C++ objects, whose size is known (no derivation, or virtual functions), which can reasonably be passed by value across C++ functions. The only examples of such classes in Kakadu are "*kdu_coords*" and "*kdu_dims*". As already mentioned, unless you really know what you are doing, you should use the "`[BIND: reference]`" convention.

"*kdu_hyperdoc*" will automatically try to create the relevant foreign language bindings for all member functions of classes which specify one of the bindings mentioned above, except for functions which contain the "`[BIND: no-java]`" (or, equivalently "`[BIND: no-bind]`") directive mentioned earlier. Note, however, that you can supply explicit "*-bind*" arguments to "*kdu_hyperdoc*" (these will usually be in a switch file, loaded with the "*-s*" option) to explicitly identify a restricted set of classes, global functions or even class member functions for which you want bindings to be generated. See the usage statement printed by "*kdu_hyperdoc*" for more on this.

You should be aware of two additional binding directives which can be added to the comments appearing after a function declaration in your header file. These are "`[BIND: donate]`" and "`[BIND: callback]`". The former specifies that the function call will donate ownership of the object from which it is invoked to some other object. This signals to the binding code that the C++ object should not be automatically destroyed if the Java (also C# or Visual Basic) object is subsequently desroyed or garbage collected. As a general rule, the C++ object belonging to any Java binding class which was directly instantiated from Java will be automatically destroyed during garbage collection, unless it has been donated. The same applies to all foreign language bindings.

You can also provide the "`[BIND: donate]`" directive inside comments de-

scribing a specific function argument, in which case it has the reverse semantics. Specifically, in that case the directive indicates that the C++ object embedded in the function argument is being donated to the object whose function is being called – it will no longer be subject to garbage collection or explicit destruction.

The "`[BIND: callback]`" directive has a very considerable impact on the construction of foreign language bindings. It may only be applied to virtual C++ functions (not necessarily pure) of classes with the "`[BIND: reference]`" binding. Moreover, the following limitations currently apply to functions marked with this binding:

1. the C++ implementation of the callback function (if any) must do nothing at all, and any return must be a natural zero (**0** for integer quantities, **NULL** for strings and object references, **false** for booleans, etc.);

2. the callback function must not have array arguments, although strings and object references are allowed.

To do something meaningful, the bound Java, C# or Visual Basic class (as appropriate) which contains a callback function must be derived by you, the developer, providing a suitable implementation of the callback function. For a simple example of this, see the "*Kdu_sysout_message*" object, declared in "*KduRender.java*". For a much more advanced example, see the "*Ckdu_bitmap_compositor*" object in the C# sample, "*KduRender2.cs.*" Note that you ARE allowed to throw exceptions from your foreign language implemention of the callback function.