Technote 1155

JNI Tips: Building Your Native-Method Libraries For MacOS

By Jens Alfke
Apple Java Team

CONTENTS

JNI (Java Native Interface) is a cross-platform standard for implementing Java methods in native code. MRJ 2.0 and 2.1 support JNI; however, some developers run into trouble getting their JNI libraries to work correctly on Mac OS. For the most part these are build- and installation-related issues. This Technote is designed to put you at ease.

I'm Superunsatisfied!

You just got your JNI-based code to compile and link ... but MRJ still throws an `UnsatisfiedLinkError` when you try to use it. If you're porting code that runs on other platforms, this can be pretty frustrating. Fortunately the problem and the fix are usually pretty simple. The most common problems are:

- You're not exporting the method implementations
- Your library's code-fragment name is wrong
- Your library file isn't located where MRJ can find it

Let's look at each of these in detail.

Back to top

Exporting The Right Stuff

For MRJ to find the right functions to call in your library, they need to be *exported.* The linker needs to know which functions to export when it links the library. The documentation for your development system (probably either MPW or CodeWarrior) will describe in detail how to set up exports. There are basically two ways to do it:

Use a .exp file. One way to set up exports is via a separate ".exp" file, which is a text file that contains the names of the exported functions, one per line. If using CodeWarrior, you add the .exp file to your project, then go to the "PPC PEF" panel of the Project Settings dialog and set the "Export" pop-up to "Use '.exp' File." If using MPW's `ppclink` tool, you use the "`-@export`" flag.

If you use a .exp file, make sure that you spell all the function names correctly (pasting them in from your source files is the best bet) and be sure to keep the file in sync when you add, remove, or rename native methods -- *or if any parameters change.*

Use #pragma export. The other method -- which I prefer -- is to use a C/C++ "`pragma`" to mark functions for export. The line "`#pragma export on`" tells the compiler to mark any function declarations it sees for export; the line "`#pragma export reset`" turns off this mode. One convenient way to use this is to turn on export mode while including the JNI-generated header file that declares all of your methods:

```
// MyNativeClass.c
#pragma export on
#include "MyNativeClass.h"  // this is the header that java.h generated
#pragma export reset

...function bodies follow...
```

Another equivalent way to mark the functions for export is to use the standard `JNIEXPORT` and `JNICALL` macros defined in `<jni.h>`; `JNIEXPORT` has the same functionality as turning on "`#pragma export`." It looks like this:

```
// MyNativeClass.c
#include "MyNativeClass.h"  // this is the header that java.h generated

JNIEXPORT jint JNICALL Java_com_foo_MyNativeClass_myMethod(JNIEnv* e) {
    ...
}
```

You then need to tell the linker to export marked functions. If using CodeWarrior, go to the "PPC PEF" panel of the Project Settings dialog and set the "Export" pop-up to "Use #pragma." If using MPW's `ppclink` tool, export-marked functions will automatically be exported.

The advantage of using the second technique is that it always exports the correct functions, even if the set of native methods change. All you have to do is re-run java.h (or recompile your Java classes, if you're using CodeWarrior's option to emit JNI headers) and the list of exported functions will automatically be updated next time you build your native library.

Back to top

Naming The Library

The second major mistake developers make is incorrectly naming the library . The name of the library should be exactly the same as the name you pass to `System.loadLibrary`. (Metrowerks' Java VM required prepending "`java_`" to the library name; MRJ does not.)

Moreover, the library name is *not* the same thing as the library's filename. The filename is ignored; it's the library name (or "fragment name") that counts. In CodeWarrior, you set it in the "PPC PEF" project settings panel. In MPW's `ppclink`, use the "`-fragname`" flag.

Note:
MRJ 2.0 and some of the early-access releases of MRJ 2.1 had problems handling non-ASCII characters (such as ácçëñtèd letters or Symbols®) in library names, and would map them to the wrong character. This problem has been fixed in MRJ 2.1.

Back to top

Positioning The Library

Finally, even if your library is built perfectly, it won't do much good if MRJ can't find it. The library file needs to be on the *CFM search path*. Basically, this means that it needs to be:

1. Inside the application itself
2. In the same folder as the application
3. In the Extensions folder
4. In a folder directly contained in the Extensions folder, *e.g.*, the MRJ Libraries folder. *This is not recursive* -- sub-folders of MRJ Libraries will not be searched. A common mistake is to put the library in the MRJClasses folder.

"The application", of course, can mean a JBindery-generated application, a browser like Microsoft Internet Explorer, or Apple Applet Runner. (Note that if you're running your code directly from JBindery without saving the settings as an app, then "the application" means JBindery itself).

If your native code is being used by a specific application, then the usual place to put the library is into the same folder as the app. If your native code is part of a shared Java library that's installed in the MRJClasses folder, then the native library should go into the MRJ Libraries folder (not the MRJClasses folder, for reasons described above).

Embedding the code fragment within the application itself is tempting, especially since -- if combined with a Virtual File System -- you can boil your whole app down to a single file. Since the data fork is already in use by the VFS, the best place to put the code fragment is into a resource. You'll have to tell the linker to generate a code resource, then copy the resource into the app and add a new entry to the app's `'cfrg'` resource to point to your code fragment.

Back to top

Debugging The Loading Process

If you have a pesky native library that just flat-out refuses to load even after you've used all the previous information to troubleshoot it, here's a way to debug some of the loading process. (You must have MacsBug installed to do this. Technote 1154, *Debugging Java Code With MacsBug,* has an introduction to MacsBug for Java developers.)

1. Before your app tries to load the native library, break into MacsBug by holding down the Command (cloverleaf) key and pressing the Power key.
2. Enter the command:
   `tvb GetSharedLibrary ';dm r3 pstring'`
   then enter "`g`" to continue.
3. At the next point where MRJ tries to load a native library, you will hit the breakpoint and drop into MacsBug. There should be a line reading "`PowerPC TVector Break At GetSharedLibrary`" in the MacsBug console, followed on the next line by the name of the library being loaded. MRJ loads a lot of libraries when it starts up, so this may not be the library you're looking for. If not, enter "`g`" to continue and go back to step 3.
4. If it *is* your library, double-check that the spelling of the library name is exactly as you specified it to the IDE or linker. If not, you need to make them match. Do the last step, below, then rebuild and try again...
5. Enter "`r8`" to display the value of this register -- you'll need it later.
6. Enter "`gtp lr`" to step over the `GetSharedLibrary` call.
7. Enter "`error r3`" to see the status returned. If this is zero (`noErr`), the library successfully loaded and initialized. This would imply that your problems are due to missing exports.
8. If the error is nonzero, it's probably in the CFM error range, `-28xx`. The error command gives you a brief description, and you can get more info by consulting *Inside Macintosh: PowerPC System Software*. In particular, the most common error is `-2804` (`fragLibNotFound`): your library or some library it imports could not be found.
9. The error may refer to your shared library or to another library that it imports. Remember the "`r8`" command you entered in step 5? Enter "`dm XXXX pstring`" now, where "`XXXX`" is the hex register value returned by the "`r8`" command. The string displayed is the name of the library that caused the error.
10. Finally, enter "`tvc`" to clear the breakpoint and then "`g`" to continue.

Back to top

Further References

- Sun Microsystems. *The Java Tutorial Continued*. Addison-Wesley, 1998.Includes a good JNI tutorial that's also available online.
- Gordon, Rob. *Essential JNI*. Prentice-Hall PTR, 1998. Excellent description of JNI, with lots of examples and a handy reference section at the end. Unfortunately, the platform-specific details cover only Windows and Unix, but this Technote should help you over the few trouble spots.
- Apple Computer. *Inside Macintosh: PowerPC System Software*. Addison-Wesley, 1994. Contains a detailed description of the Code Fragment Manager (CFM) and how it loads shared libraries. (The entire book is available online in PDF form.)
- Apple Computer. *Building and Managing Programs in MPW*. Documents MPW tools, including `ppclink`, and build processes.

Back to top

Downloadables

 Acrobat version of this Note (49K).

Back to top

Acknowledgments

Thanks to Nick Kledzik for reviewing this Technote and suggesting additional tips, and to kelly jacklin for extra JNI info.

To contact us, please use the Contact Us page.
Updated: 01-March-99

Technotes | Contents
Previous Technote | Next Technote