



Device Drivers and System-Level Programming in Java

Chris McKillop
Software Engineer
QNX Software Systems Ltd.
cdm@qnx.com

Introduction

In the not-too-distant past, engineers who wrote device drivers, operating systems, and other system-level software did most of their work in assembly. Today, those same engineers use C — even though hand-tuned assembly can still run faster and smaller than the code generated by many C compilers. So why the change? Because experience has shown that the portability, speed of development, and built-in runtime support (the C Standard Library) of C far outweigh the relatively small cost of using it. As it turns out, much the same can be said for Java. In fact, system-level software written in Java can offer an even better cost-benefit ratio, making Java a viable language for both current and future embedded designs. In this paper, we walk through the steps of writing a device driver in Java, from hardware initialization to client interfacing. We also address key issues, such as realtime performance and portability, and identify which types of applications benefit the most from Java device drivers.

Driving Hardware

What, exactly, is system-level programming? For most developers, it means programming at the lowest level in the system, talking directly to, and controlling, hardware. Device drivers represent a specific type of programming at this level. Typically, they interface with both the operating system (OS) and the hardware to give application-level programs access to devices in some standard fashion. (An upfront warning: some types of device drivers can't be done in Java. For example, under many OSs, drivers are either loaded as shared objects or statically linked with the kernel itself. You couldn't use Java to write a device driver for such an OS since the Java Virtual Machine runs outside of the kernel.)

Most drivers have three main sections:

- **Hardware initialization** — To “drive” the hardware means talking to it, which in turn requires direct access to it, through either mapped memory addresses or specific in/out instructions.
- **Interrupt handling** — To set up an interrupt handler, the driver may need to talk to CPU-specific hardware. In most cases, however, it interfaces with interrupt services provided by the OS. Using those services, the driver can register an interrupt handler that is invoked or unblocked whenever the desired interrupt fires.
- **Client interfacing** — Interfacing with clients is the trickiest part, and varies from system to system. On many systems, the client has to make system calls to the kernel, requesting to interact with the specific device. Thankfully, that isn’t the only way to interact with clients.

Before exploring these three areas in detail, we need to cover some key Java concepts — in particular, the Java Native Interface, or JNI.

JNI

The Java Standard Library provides a lot of functionality. However, it doesn’t always provide the features needed to interface directly to the underlying system. Consequently, Sun introduced the JNI, which lets you safely and easily call native functions loaded from a shared library. These functions can be passed classes as parameters, interact with those classes, throw exceptions, and behave (from the perspective of the Java program calling the native function) just like any other Java function. To become a native method, a function within a class simply uses the keyword “native” coupled with a static declaration. For example:

```
package com.qnx.examples;
public class NativeTest {
    public static native int test();
}
```

The native method *test()* can now be invoked as *NativeTest.test()*. In response, the Java Virtual Machine (JVM) will try to find the symbol `Java_com_qnx_examples_NativeTest_test` in its process space. As you can imagine, however, running the above example alone will cause an exception (specifically, a `java.lang.UnsatisfiedLinkError`), as it is unlikely that any of the default shared libraries loaded by the JVM will provide that symbol. Consider, therefore, the following example:

```
package com.qnx.examples;
public class NativeTest {
    static {
        System.loadLibrary("mynative");
    }
    public static native int test();
}
```

The *static* section added to the `NativeTest` class is invoked before any other static functions. The `System.loadLibrary()` call, meanwhile, tells the JVM to load the `mynative` library. This library is system dependent — on UNIX, for instance, the JVM would load `libmynative.so`; on Windows, `mynative.dll`. In order for `test()` to function, the `mynative` library must include at least the following function:

```
#include <jni.h>

JNIEXPORT jint JNICALL
Java_com_qnx_examples_NativeTest_test(JNIEnv *env, jclass obj)
{
    return -1;
}
```

Every JNI function takes at least two parameters: a pointer to the JVM’s environment in order to interact with it, and a reference to the instance of the class that is invoking the JNI. Since the `test()` function itself takes no parameters, only those two parameters need to be defined.

If the above source is compiled into a shared library for your system, named appropriately, and placed in a location where the JVM can find it, then `NativeTest.test()` will run and return `-1`. The remaining macros (`JNIEXPORT`, `JNICALL`, ...), which are provided by the `jni.h` header file, hide JVM specifics (for instance, calling conventions) as much as possible, allowing your JNI code to be more portable.

Hardware Initialization: Accessing Physical Memory

Though the ability to extend the JVM with native methods is a powerful feature, there is a drawback: every time you use a native method, you have to “port” that method when moving to a new system. A good design will keep logic and high-level work in pure Java, with the minimum JNI code required to get the work done. Such an approach allows for maximum re-use of natives between projects.

For example, consider the following class, which serves as a container for POSIX system calls. By keeping all of your natives in a single class like this, you can greatly simplify porting between systems:

```
public class POSIX {
    public static int MAP_SHARED = 0x001;
    public static int MAP_PHYS = 0x10000;
    public static int PROT_READ = 0x100;
    public static int PROT_WRITE = 0x200;
    public static int NOFD = -1;
    public static native long mmap64(long addr, long len,
                                     int prot, int flags,
                                     int fildes, long off );
    public static native int munmap(long addr, long len);
    public static native byte in8(long addr);
    public static native void out8(long addr, byte value);
}
```

Using this `POSIX` class as a base, let’s examine how a driver can access hardware for configuration and control. As mentioned earlier, hardware initialization requires gaining access to the hardware. If your

JVM complies with JSR-0001, the Real-time Specification For Java (RTSJ), you can do this easily, using the `RawMemoryAccess` class that the RTSJ defines. Instances of the `RawMemoryAccess` class allow your Java driver to map a specific physical range of memory and then read or write values into that memory area. The driver can, as a result, initialize and control hardware devices.

At the time of writing, only one JVM (jTime from TimeSys) fully implements the RTSJ specification, while another (j9 from IBM) implements portions of it. By using the `POSIX` class, however, you can build a simple class that works much like `RawMemoryAccess`; in fact, this class will allow your code to work regardless of whether your JVM supports the RTSJ. For an example of such a class, see the code sample below.

Note that the RTSJ supports several other access methods, including `get/setShort()` and `get/setInt()`, as well as `setBytes()`, which sets a range of device memory to a given range of byte values. You can add these functions as needed while maintaining portability with a system that has a complete `RawMemoryAccess` class.

```
public class RawMemoryAccess {
    long size;
    long base;
    long vaddr;

    public RawMemoryAccess(Object type, long base, long size) {
        this.size = size;
        this.base = base;
    }

    public long map() {
        vaddr = POSIX.mmap64(0, size,
            POSIX.PROT_READ | POSIX.PROT_WRITE,
            POSIX.MAP_PHYS | POSIX.MAP_SHARED,
            POSIX.NOFD, base);

        return vaddr;
    }

    public void unmap() {
        POSIX.munmap(vaddr, size);
    }

    public long getMappedAddress() {
        return vaddr;
    }

    public byte getByte(long offset) {
        return POSIX.in8(vaddr + offset);
    }

    public void setByte(long offset, byte value) {
        POSIX.out8(vaddr + offset, value);
    }
}
```

Interrupt handling

Compared to memory access, which has a very simple interface, interrupt handling is fairly complex. For starters, some OSs won't let even let you access interrupts. Windows and Linux, for instance, allow only kernel entities to access the interrupt subsystem. In comparison, OSs targeting embedded and real-time markets either make no distinction between application and kernel space (e.g. VxWorks) or provide a suitable API for dealing with user-space interrupts (e.g. QNX[®] Neutrino[®] RTOS).

Even if an OS provides an API that can be accessed via the JNI, you may still have to address timing issues within the JVM. If the JVM conforms to the RTSJ, no problem: you can create real-time threads that run unencumbered by the JVM garbage collector. Without those real-time threads, however, any interrupts managed in Java will be at the mercy of the garbage collector and must, as a result, tolerate latencies that last milliseconds or even seconds. If such latencies are unacceptable, then a solution that leans on the JNI is in order.

Assuming that a pure Java solution is feasible, you could build a JNI container class for interrupt management, as shown in the following code example. Using such a class, you can start a high-priority thread (or `NoHeapRealtimeThread` if the RTJS available) that would attach to the required interrupt, unmask the interrupt, and wait for the interrupt in a processing loop. The thread could then read or write to the hardware on each interrupt and, using a thread-safe data structure, publish any data to lower-priority threads in the system.

```
public class RTOS {
    public static native int InterruptAttach(int irq);
    public static native int InterruptDetach(int id);
    public static native int InterruptWait(int id);
    public static native int InterruptMask(int id);
    public static native int InterruptUnmask(int id);
}
```

For some systems, this almost pure Java solution won't provide the desired performance and latency. If so, you must use a native interrupt handler. During the `RTOS.InterruptAttach()` call, the associated JNI code would create a thread or register a handler function, then return a pointer to a heap-allocated data structure for the ID. Instead of calling `RTOS.InterruptWait()`, the code would call `RTOS.InterruptPopData()` and `RTOS.InterruptPushData()`. The native thread would then read and write from FIFO-style (first in, first out) structures as interrupts fire, unblocking any Java threads that are waiting for data to "pop." See Figure 1.

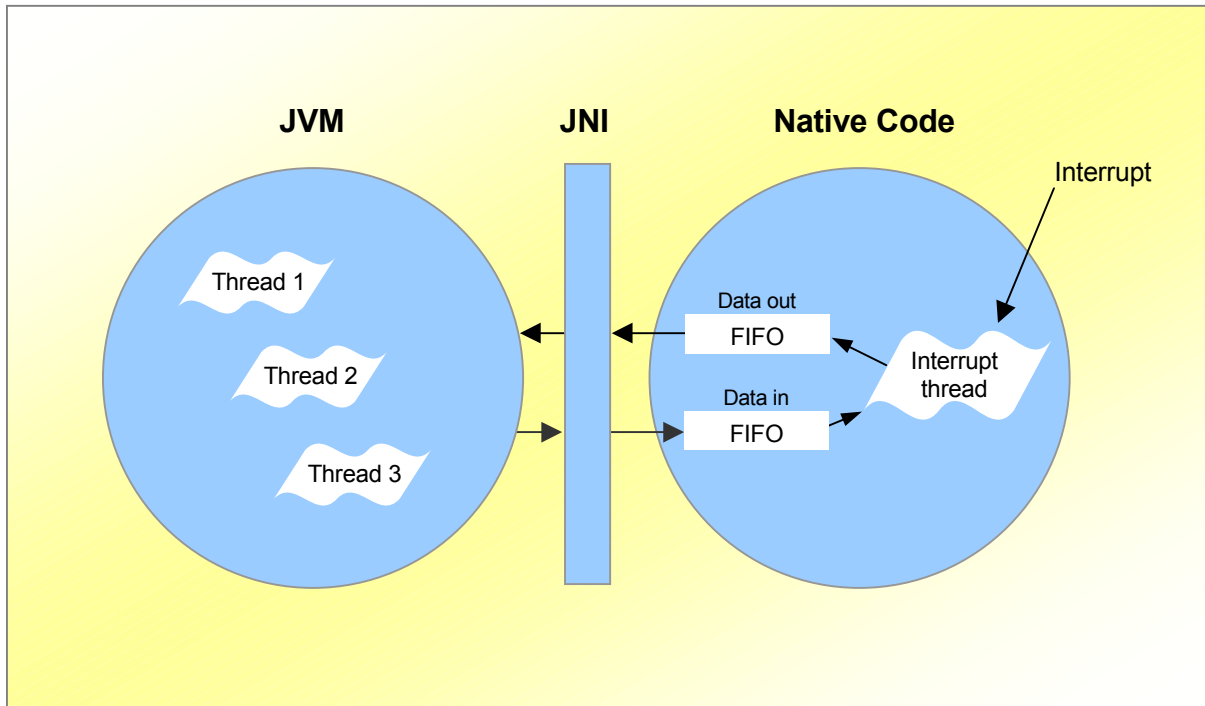


Figure 1 — Using a native interrupt handler.

Client Interfaces

A device driver is of little use if the data it moves around can be accessed only by the driver itself — the driver needs a way to communicate with other parts of the system. On traditional OSs such as UNIX, a client process can access a device by opening a file descriptor to the device — usually by calling *open()* on something in */dev*. When the process subsequently makes a system call that passes this descriptor, the kernel routes the call to the proper device. Obviously, this sort of approach won't work for a driver written in Java. On systems such as the QNX Neutrino RTOS, on the other hand, it's possible for any user-space application, including programs running in a JVM, to register a device and handle the open/read/write actions. Still, this approach isn't always portable. That leaves us with two possible routes for talking to clients — sockets and inner-JVM.

Sockets

Java was built for “The Network.” Any compliant JVM (J2ME and up) can act as a TCP (or UDP) server. This provides an obvious choice for client programming, particularly since most other languages can also talk over TCP/IP. By having a thread in your Java driver act as a TCP daemon on a specific port (or range of ports), you allow the driver to be accessed by any client process on the system (or the network, if needed). Since sockets are, on most systems, file descriptors, you can even use standard functions like *read()* and *write()* to communicate with your driver from C programs.

Inner-JVM

If your entire system is being written in Java, you can provide an interface to your driver that is accessible through Java only. To do this, you could provide facilities to get an `InputStream` and/or an `OutputStream` to your driver for performing I/O. Or, you could simply define a standard `IDriver` interface that all clients would use for talking to any Java driver in the system. The upside to this approach is performance: it bypasses the entire network stack and minimizes data copies. The downside is fragility: if your JNI code or the JVM causes a fault, then your application and your drivers will be taken down all at once.

What's This Good for Anyway?

Obviously, some devices are better suited to system-level work in Java than others. Front-panel/HMI systems, for example, are ideal candidates, particularly since buttons, touch-screens, and video output drivers are easily coupled with application logic. Also, by using Java's extensive built-in GUI APIs, you can set up an emulation environment that runs with a simulated frame buffer in a window, with widgets for the buttons. Such an environment lets you design, build, and debug the application-driver interface even before you have hardware in-house.

Other candidates include any low-interrupt rate, DMA-capable device. Such devices have enough buffering between interrupts for the latency present in the JVM to be a minor issue. Poor candidates for Java drivers include high-interrupt devices with small hardware buffers and low-latency demands — 16550 UARTs and IEEE 1394 adapters, for instance.

Overall, Java has become a tool that embedded designers should consider when building new systems. Compared to C, it offers greater portability, a higher level of design abstraction, and a richer standard library. Moreover, its performance is starting to rival that of many C compilers. And, as we have shown, Java can serve as the basis not only for applications, but for device driver and other system-level work as well.