Systronix White Paper -- Why Use Java?

Bruce Boyes, Systronix Inc.
http://www.systronix.com
http://www.PracticalEmbeddedJava.com

First Draft

JCX is an approximate acronym for "Java Control System". It is intended for use in industrial, educational, and robotic applications. JCX is not just one technology, rather, it is a system architecture built upon multiple open standards. JCX is programmed in native-execution, realtime Java. We believe Java is the biggest news to come to the embedded space since C arrived over 20 years ago.

Everyone has heard of Java, but it is still a relative newcomer to small embedded systems, especially realtime systems. Some are incredulous upon hearing that our industrial and robotic systems are programmed in Java. This paper is written to answer the most common questions about embedded Java, and specifically native-execution Java.

This document and many others are available online at http://www.jcx.systronix.com

Why consider Java in the first place?

Java is a modern, object-oriented language based on open, public standards. Java is much more standardized and has a much richer collection of core functions than any other general-purpose computer language. Many models of the so-called real world lend themselves well to representation by objects.

Objects can give your programs a degree of modularity which makes them easier to understand and maintain. It's possible to provide users of your code with a consistent, public API (Application Programming Interface) while keeping the implementation details private.

Most developers are aware that Java applications can execute with little or no change on multiple hardware platforms. This is not really the main benefit of Java - robustness and reliability is. Still, portability of code is a compelling benefit. When you consider embedded systems with their typically unique interfaces to hardware devices, and often peculiar user interfaces, you can imagine that such portions of the program will not be so easily portable, even in Java, and this is true. Even there, Java code can be written using techniques such as abstract classes and the class.forname method so that your application can dynamically bind to the correct hardware-specific runtime support it needs. We use this technique to support or SBX2 LCD and keypad drivers on hardware as radically different as SaJe and TINI.

There are excellent, free Java development tools such as the IBM Eclipse IDE (Integrated Development Environment). There is a huge community of Java programmers, many of whom make their work available for use by others.

What makes Java better than C/C++?

- Java is not magical, it just uses about 30-year newer technology than C. When C first appeared, networking did not exist. Objects were unknown. Systems with 32 KBytes of memory were considered large. All this has changed in the past 30 years. So you would hope that programming languages would also change to keep up.
- The core Java API is very rich and includes standard packages for serial I/O, Ethernet and internet access, security, graphics, sound, and other typical functions. This means, for example, that Java programs which use the javax.comm package can literally execute unchanged on multiple Java platforms. We do this routinely on such different platforms as JStamp, JStik, SaJe, TStik/TINI, SNAP, and the PC. C/C++ has no such standard support, and C programmers have never been able to enjoy this level of portability.
- Java has a consistent set of proven and stable APIs for TCP/IP networking, and has had from day one. No other language can claim that. I don't know about you, but one thing that drives me batty about C is the plethora of non-standard, arbitrary add-on packages for serial and network I/O. How many of these do you want to learn in your life?
- Java has a very robust, mature security model which is has been pounded on by a huge community. You can't say the same for Vendor X's custom C-language TCP/IP stack and security provisions.
- Java has built-in exception handling, which C lacks completely. It's possible to circumvent Java's good intentions but at least you have to consciously do so. C gives you no help at all.
- Java programs are potentially much more reliable. C has essentially no runtime error checking, memory allocation is all manual, etc. Java does memory management automatically, bounds-checks array access, etc so it handles the big problems for you.
- Programmer productivity is at least 2X greater with Java. You can get a lot done in a short amount of time with Java because it has such a rich library of functions already built into the language. There is a huge amount of open source, free software which you can probably leverage -- and most of the time it will actually compile and run on *your* hardware!
- Java code can be tested and debugged on a platform such as a PC, then moved with little or no changes to an embedded system.
- A good programmer can write a greater quantity of more robust, re-usable code in Java than the same programmer using other language tools. Sure, a bad programmer can write lousy code in any language, so you still have to use your brain with Java. It's not a panacea.

Java programs can be better - a LOT better than C

In Java, memory management is automatic, and many classes of bugs (such as buffer overruns) and stray pointers are impossible in Java. Java has excellent code documentation and archiving tools built in (though it's up to the programmer to take advantage of them). Java has extensive exception handling built into the language, and

you generally must go to some effort to circumvent it.

These factors make it possible for a competent Java programmer to write highly readable, maintainable, and robust programs much more rapidly and easily than any other general purpose programming language. Such benefits do not come automatically - programmers must be familiar with these Java capabilities and know how to properly apply them.

Isn't Java slow? (Not today)

Java compilers, interpreters, and runtime implementations have come a long way in the last 5 years. The execution of well-written Java code can now be on a par with well-written C code. Most Java code is executed by a JVM (Java Virtual Machine) which can be an interpreter, a JIT (Just-In-Time) compiler, or an adaptive optimizing engine such as HotSpot.

It's important to compare Java programs to *similarly robust* programs in C or assembly. This is because C and assembly code have zero built-in safety features such as array bounds checking, or automatic memory management. C and assembly programmers are responsible for writing their own safety features or using proprietary third-party libraries to achieve the same ends. Many C and assembly programmers ignore exception handling (eventually to their own peril). Code which executes quickly but is unreliable is no bargain. There are a whole class of well-known C and assembly bugs which are difficult to find and fix. These bugs are virtually impossible in Java. By the time you add equivalent safety and robustness code to C and assembly programs, you will find that most or all of the claimed C and assembly performance advantage has evaporated.

Anyone in charge of a schedule and budget will agree that software development time, reliability and maintainability are the main factors driving the true lifecycle cost of software. Typically 60% of a product's lifecycle cost is the cost of maintenance and upgrades. In an embedded system, this total cost far outweighs the cost of hardware or even the cost of initial development.

It's possible to write slowly-executing programs in any language, of course, and perhaps easier in Java since memory management is handled automatically. Careless object creation and destruction in Java can be transparent to the programmer, but can have a huge effect on performance. It's important for Java programmers to be trained in efficient use of objects. Actually, the same holds true for any object-oriented language.

Native-execution Java is blazing fast

Native- or direct- execution Java chips are built with Java as their native instruction set, so they execute Java as quickly as other controllers execute equivalent instructions in C or assembly code. There's nothing magical about creating a processor which executes Java byte codes natively.

Our native execution Java systems offer deterministic real-time performance with very

low power consumption. In fact, JStamp uses less power than many PIC-class control modules, yet has many times the memory and far greater computational performance.

JStik can execute approximately 20 million Java byte codes per second and can switch thread contexts in less than one microsecond. The JStik HSIO (High Speed Input Output) expansion bus can burst data at up to 50 megabytes per second. JStamp can execute 3 million Java byte codes per second or throttle down and run on a 9-volt transistor radio battery for over 40 hours.

Isn't Java hard to learn?

Java syntax is based on C so if you are familiar with C you have a big head start on learning Java. Understanding objects and applying them appropriately is not trivial. Neither is learning the common Java packages and idiosyncrasies. All of this will probably take a year or more to master, assuming you use Java daily and study some of the many excellent Java reference texts.

Is there a Dark Side to Java? If so, what is it?

Java is not a panacea for all programming problems. Sloppy programmers can circumvent many of Java's safety features and write bad code in Java. The programmer is still the most important link in the software development chain (and it's unlikely that any computer language or development tool will ever completely replace the human brain).

Java is a high-level language whereas C is a high level language with low-level constructs such as direct memory access and manual memory allocation. Other aspects of Java are more highly abstracted than C or C++. This abstraction can get in your way when you are writing interfaces to actual hardware. For example, if you are writing code (using javax.comm) to interface to an RF Modem, you need to know when transmitted data is actually leaving the serial port hardware so that you can switch the modem from transmit to receive mode. The abstraction of Java makes such tasks difficult.

…to be continued at http://www.PracticalEmbeddedJava.com/