Java for embedded systems

An introduction to using Java in embedded systems

Summary
Java is great for downloading applications over a network, but what about for embedded systems? The same features that lend themselves to networkability, fit well in the process-control field, too. This applet demonstrates some real-time principles. *(2,000 words)*

By David L. Ripps

Java is a high-level language with which to write programs that can execute on a variety of platforms. So are C, C++, Fortran and Cobol, among many others. So the concept of a portable execution vehicle is not new. Why, then, has the emergence of Java been trumpeted so widely in the technical and popular press?

Why is Java different from other languages?

Part of Java's novelty arises from its new approach to portability. In previous High-level languages, the portable element was the source program. Once the source program is compiled into executable form for a specific instruction set architecture (ISA) and bound to a library of hardware-dependent I/O, timing and related operating system (OS) services, portability is lost. The resultant executable form of the program runs only on platforms having that specific ISA and OS. Thus, if a program is to run on several different platforms, it has to be recompiled and relinked for each platform. And if a program is sent to a remote target for execution, the sender must know in advance the exact details of the target to be able to send the correct version.

With Java, source statements can be compiled into *machine-independent*, "virtual instructions" that are interpreted at execution time. Ideally, the same virtual code runs in the same way on any platform for which there is an interpreter and OS that can provide that interpreter with certain multithreading, file, graphical, and similar support services. With portability moved to the executable form of the program, the same code can be sent over the net to be run without prior knowledge of the hardware characteristics of the target. Executable programs in the Java world are universal.

In principle, portability could have been achieved in the C or C++ world by sending the source program over the net and then having the compilation and linkage done as a pre-step to execution. However, this approach would require that the target system have sufficient CPU speed and disk capacity to run the sophisticated compilers and linkers required. In the future, network platforms may not have the facilities to run even a simple compiler.

Is that all?

Java is not just a new concept in portability. The Java language evolved from C and C++ by locating and eliminating many of the major sources of program error and instability. For example, C has an element know as a *pointer* that is supposed to contain the address at which a specific type of information is stored. However, the pointer can be set to literally any address value, and by "casting" a programmer can trick the compiler into storing any type of information at the arbitrary pointer address. This is convenient if you write error-free code, and a snake pit if you don't. Java does not have pointers.

Equally important, Java has built-in support for multiprogramming. C and its immediately descendent C++, were designed to express a single thread of computing activity. There was no inherent support for multiple program threads executing simultaneously (on multiple CPUs), or in parallel (timesharing a single CPU). Any such facilities had to be supplied by an external multiTasking operating system. There are several good programs of this type readily available, such as MTOS-UX from Industrial Programming. However, the services provided are all vender-specific. Nether ANSI nor any of the various committees set up to hammer out a universal set of OS services ever produced a single, universally-accepted standard. There are in fact, several proposed standards, so there is no standard.

Java bypasses the problem by building multithreading and the data synchronization it requires directly into the source program. You still need an OS to make this happen, but, the semantic meaning of the OS actions is standardized at the source level.

A standard at last

Java then has all of the technical requisites to become the standard programming language for programs to be distributed over the net. And with a well-supported campaign spearheaded by Sun Microsystems, Java is becoming the *de facto* working standard. Will Java supersede C as the language of choice for new programs in general? With network programming likely to play an increasingly larger part the overall programming field, I think so.

Java for embedded systems

Embedded or real-time systems include all those in which timing constrains imposed by the world outside of the computer play a critical role in the design and implementation of the system. Common areas for embedded systems are machine and process control, medical instruments, telephony, and data acquisition.

A primary source of input for embedded systems are random, short-lived, external signals. When such signals arrive, the processor must interrupt whatever else it is doing to capture the data, or it will be lost. Thus, an embedded program is most often organized as a set of individual, but cooperating threads of execution. Some threads capture new data, some analyze the new data and integrate it with past inputs, some generate the outgoing signals and displays that are the products of the system. Currently, most embedded programs are coded in C, with critical parts possibly in assembler.

Putting the issue of execution efficiency aside, some of the major problems of C for embedded systems are:

- The permissiveness of C operations, which can lead to undisciplined coding practices and ultimately to unstable execution.

- The absence of universal standards for multithreading, shared data protection, and intra-thread communication and coordination, which can make the program hard to transfer to alternate platforms. But, these are just the problems that Java solves. Since many programmers will have to learn Java because of its importance to the net, it will be natural for Java to supplant C in the embedded world.

The use of Java may be different, however. We anticipate that Java programs for embedded applications will differ from net applets in at least five major ways. Embedded applications will be:

- compiled into the native ISA for the target hardware.
- capable of running in the absence of a hard or floppy disk, and a network connection.
- supported by highly tailored, thus relatively small run-time packages.
- able to execute on multiple processors, if needed for capacity expansion.
- contain significant amounts of legacy C code, at least during the transition from C to Java.

No need to interpret an embedded Java program

In an embedded system (where the current ISA is always known) there is little benefit for Java's universal executable format. And there is a significant time penalty for execution by interpretation. The Java source is likely to be compiled into the ISA-specific instructions of the target hardware, as is currently done with C. At least in the final product that goes out to the field.

During program development, especially when doing initial element testing, it may be very convenient to debug individual Java components by executing them on any available Java-capable platform, such as a PC or SPARC workstation. There will soon be a large variety of debugging tools that exploit the speed and extensive facilities available on a PC or workstation And during early testing, execution efficiency is of little importance.

We expect that in most cases the embedded program that finally goes out into the field will have been compiled into native code. As a result, the program not need contain the Java interpreter and libraries of class code unless those classes are actually used by the application. The result is a smaller and faster program compared with the standard applet approach.

Nevertheless, some embedded programs may benefit from retaining the capacity to dynamically load new Java source code. Those are applications that must be altered partially or totally while in the field. One example is a process control applications for which the process is changed from time to time to make alternate products.

Special hardware for embedded applications

Java programs currently execute on PCs or workstations, which of course have a hard disk. Thus, the Java interpreter expects to be able to load any needed class libraries and code components from a disk.

In contrast, many embedded systems are housed within a highly-constrained box, and have to operate in the harsh factory floor environment. Lack of space, power or cooling, or excessive vibration may preclude a standard hard or floppy disk As a result, we anticipate that embedded applications will be structured as a small section of initialization code that starts the Java programs from a ROM disk. (A ROM disk behaves like a hard disk, but uses read-only-memory as the medium.) Any needed temporary data files will be stored in RAM disks.

Java applets may be called upon to give voice to sound files and sight to arbitrary still or moving picture files. Thus, the Java runtime operating system must provide hardware support for a wide variety of potential audio-visual operations. The Java applet environment places open-ended requirements on its OS.

Embedded applications are generally not so free wheeling; by the time the application is ready for the field, the OS requirements and hardware environment should be known. Thus, the OS needed for an embedded Java applications can be tailored to that specific application, by including only what is known to be needed. The saving in memory size and maintenance effort can be enormous.

Mixed systems: multiple languages, multiple CPUs

While we expect Java to supersede C as the primary programming language for embedded systems in the near future, there is still an enormous number of lines of C code in operation. Companies will have to work with that code for many years as the transition to Java runs its course. Many systems will have to be a mixture of legacy C code and Java enhancements.

It is not trivial to integrate an overall application with some components written in Java and others in C or assembler. Part of the problem arises from security issues: How can Java guarantee the security of the system if execution disappears into "unknown" regions of code? Furthermore, the danger is compounded if the non-Java code were to make OS support service calls, especially calls that alter the application’s threading and data-protection aspects. Java expects to be the sole master of such matters.

Thus we see that mixed language systems may have to exist, but this is not going to be easy. Similarly, there may be problems with multiple CPUs.

Current CPUs are fast, and get faster with each new generation. Yet, there are some embedded applications for which a single CPU still does not have enough power to keep up with a worst-case burst of external input. Such systems require multiple CPUs working together to complete the required processing. Even if the system can handle current work loads, the next version may not.

Do we have a problem?

When you combine the desire to write in Java, with the need to execute on unique, system-specific hardware, possibly with mixed source languages and multiple CPUs, you introduce a major obstacle. You are not likely to get an off-the-shelf Java OS from Sun Microsystems.

Many companies that have previously offered their own proprietary real-time OS are now developing a Java OS, or are seriously considering such an offering. My own company, Industrial Programming, is currently using its experience with embedded multithreading/multiprocessor operating systems to create a new system that will handle applications written in both Java and C. And as the case with its traditional product, MTOS-UX, the OS is transparent to the number of tightly-coupled CPUs that are executing the application code. If one CPU is not enough, you can add more without altering the application.

About the author
David Ripps has been in the computer field for over thirty years. He is currently Vice President of Industrial Programming, Inc. (Jericho, NY). His functions there include the technical supervision of the MTOS-UX product line. Among the technical areas in which David has worked are: realtime operating systems, realtime applications and computer simulation of physical processes. He is the author or co-author of three books and numerous technical articles. David's formal education is in Chemical Engineer, with a B ChE from Cornell and M ChE plus PhD from NYU. David lives in New York City with his wife, Sylvia, and their children, Elana and Samara.