

# Java in Embedded Systems

## Why Java?

Although platform independence has been hailed as Java's greatest strength, it is equally important to note that it is easier to produce bug-free software in Java than in C or C++. Java was designed from the ground up to produce code that is simpler to write and easier to maintain. Though they based their language on the syntax of C, the developers of Java eliminated many of that language's most troublesome features. These features sometimes make C hard to understand and maintain, and they frequently lead to undetected programming errors. Here are just a few of the improvements:

- All of Java's primitive data types have a fixed size. For example, an int is always 32-bits in Java
- Automatic bounds-checking prevents the programmer from writing or reading past the end of an array
- All test conditions must return either true or false. Common mistakes, such as `if (x = 3)`, are detected at compile-time, thus eliminating one entire set of bugs
- Built-in support for strings and string manipulation allows statements like "Hello, " + "world!"

In addition, Java is an object-oriented language similar to C++. This forces software developer to structure their data and functions into logical units called classes. Encapsulation, polymorphism, and inheritance are all available and are used extensively in the built-in class libraries. Java simplifies inheritance by eliminating multiple inheritances and replacing it with interfaces. It also adds many new features that are not available in C++, most notably:

- Automatic garbage collection simplifies dynamic memory management and eliminates memory leaks
- A built-in threads library makes applications written in Java more portable by providing a consistent thread and synchronization interface across all operating systems
- An integrated exception mechanism organizes software exceptions into a logical class hierarchy and does not allow programmers to ignore them

Unfortunately, it is not possible to do everything in Java that you may be accustomed to doing in C. In particular, Java does not allow manipulation of pointers (which are replaced by references). However, much as assembly code can be called from other languages, it is possible to call C/C++ and assembly functions from Java. This technique is known as "native methods". Native methods allow device drivers and other software that manipulates memory or hardware registers directly to be written in another language, without forsaking the benefits of encapsulation.

## Java development tools

Before we talk about using Java in embedded systems, it is helpful to have a basic understanding of the Java development paradigm. The relationship between the various Java development tools is illustrated in Figure 1 and described in the following paragraphs.

In Java, the compiler is called a *class compiler*. A class compiler compiles high-level Java code into a set of assembly-like instructions known as Java bytecodes. The bytecodes are the machine language of Java's "virtual machine." The virtual machine idea is central to Java's platform independence. A virtual machine can be any platform-hardware or software-that is capable of understanding and executing Java bytecodes. There are currently three such platforms: Java Virtual Machines, Just-in-Time Compilers, and Java Processors.

The Java Virtual Machine (VM) is a piece of software that translates Java bytecodes to the processor's native opcodes as they are executed. There are many implementations of the Java VM available, but all of them support the same set of bytecodes defined in the Java Virtual Machine Specification. Versions of the Java VM are available for many common hardware platforms. In an embedded system, the Java VM is essentially a set of threads that execute on top of a real-time operating system (RTOS).

Interpreted languages are portable, but slow. Just-in-Time (JIT) compilers were invented to speed the execution of Java programs by speeding up the language translation process. A JIT compiler is a drop-in replacement for the Java VM. The only difference is that it keeps a copy of all previously translated code for potential reuse. That way, the same section of code never has to be reinterpreted. Only new bytecodes that have not been previously executed need to be translated. A typical program with loops and repeated function calls should execute about 10 times faster under a JIT compiler. That makes the speed of programs written in Java comparable to those written in C++.

To speed the execution of Java bytecodes even further, Sun Microsystems has announced a line of Java processors. These are hardware realizations of the Java virtual machine. In other words, each Java processor is a microprocessor with a set of opcodes that are identical to the Java bytecode standard. To eliminate the need for native methods, these chips will have a set of extended bytecodes for accessing memory directly. It is reasonable to expect that these so-called JavaChips will execute Java bytecodes as quickly as any processor can execute its own assembly language. However, since the programs themselves will be written in the high-level Java, it is more realistic to expect performance comparable to C.

## Embed Java toolkit

The ideal embedded development kit for Java would integrate the above tools with existing cross-compilers, assemblers, debuggers, and real-time operating systems. The parts of the software written in Java could be coded and tested on the host platform, even before the target hardware becomes available. In addition, Ahead-of-Time Compilers would be available to further speed the execution of Java programs.

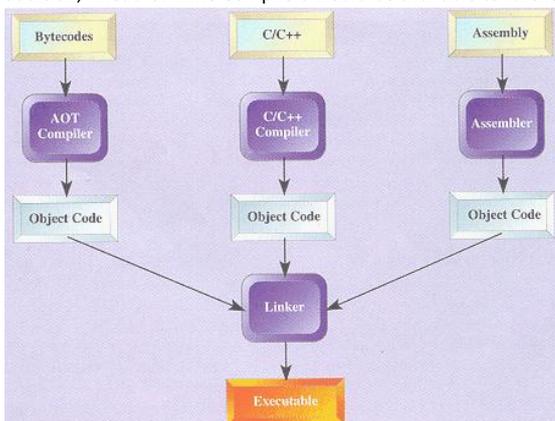


Figure 2 Using Java without a virtual machine

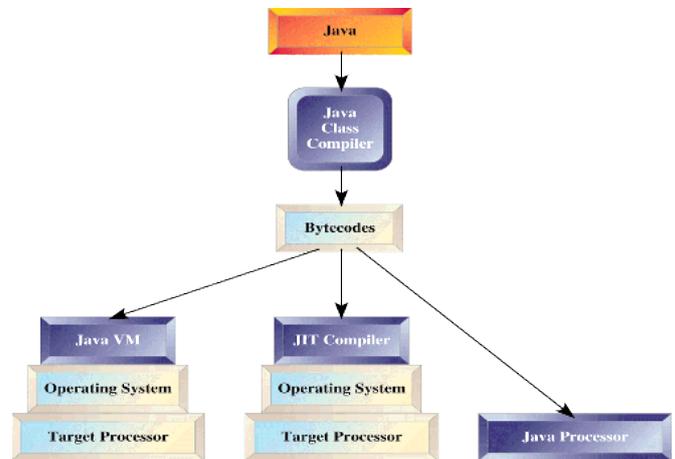


Figure 1 Java development tools

An Ahead-of-Time (AOT) compiler is a tool for converting Java bytecodes into the native object code of the target processor. With an AOT compiler, it would be possible to write software in Java but eliminate the run-time overhead of the Java VM or JIT compiler. The virtual machine would in this case be replaced by a smaller piece of software responsible only for garbage collection and exception handling-possibly provided with the RTOS. Java bytecodes that have been compiled with an AOT compiler could be linked to C/C++ and assembly programs using existing tools. A development scenario involving all three languages is shown in Figure 2.

Given the current set of bytecodes, it is unlikely that an entire embedded software project could be written in Java. For one thing, Java has no means of accessing memory or hardware registers directly (the downside of having no pointers). Unless bytecode extensions are introduced or JavaChips somehow supplant traditional embedded processors, there will always need to be device drivers and other pieces of supporting software written in C and assembly.

In a mixed language environment, developers will need an integrated development environment that understands Java's native methods. Ideally, such a tool would handle the details of mixing Java and other languages, support integrated version control and other niceties, and include a powerful multilingual debugger.

The debugger is perhaps the most critical feature—as it is in any development environment. It must be able to switch easily between Java, C/C++, and assembly language modules. In addition, it should be able to seamlessly communicate with: (1) a Java virtual machine running on the development workstation (for simulation), (2) a virtual machine running on the target (for debugging), or (3) an emulator installed on the target hardware (for real-time tracing). All three of these configurations are illustrated in Figure 3.

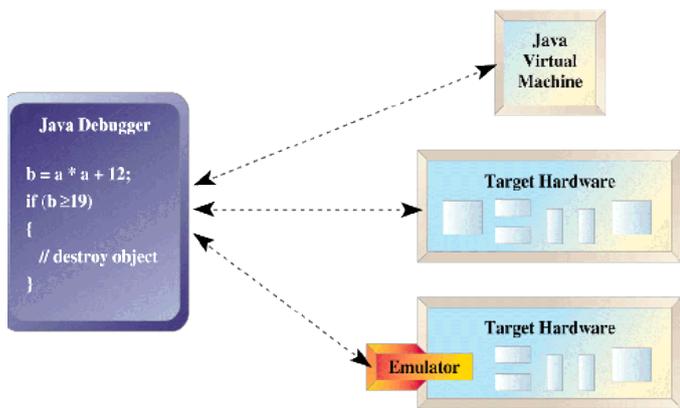


Figure 3 The ideal Java debugging environment

**What tools are currently available?**

Although many of the individual components are available today, no single package fills all of the above requirements. JavaChips and AOT compilers will not be ready for some time, and JIT compilers and remote debuggers are not yet available for embedded systems. However, many of the other pieces are beginning to come together.

In particular, Java class compilers are widely available from multiple vendors. They are inexpensive (frequently less than \$100 per seat) and can be run on PC, Mac, and UNIX workstations. These same class compilers are used to develop commercial software and Web applets. As a result, the development environments are more sophisticated than most of the tools embedded developers are currently using.

Implementations of the Java VM are available for several real-time operating systems, including Integrated System's pSOS. In addition, Java VM's are under development at Wind River Systems (VxWorks) and Microtec (VRTX). These are three of the most prominent RTOS vendors, so it is reasonable to expect other vendors will follow suit.

Unfortunately, the development of JIT compilers for embedded systems is lagging behind. Although they may be eventually made obsolete by AOT compilers, it would be nice to have them now. We are pleased to note that there

is at least some movement toward the introduction of AOT compilers. MetaWare is in the process of developing the first such compiler for 32-bit x86 processors. We feel that the development of AOT compilers for other common embedded processors is crucial to Java's success as an embedded language.

In the area of Java debuggers, there is also good news. A Java Debugger API has been established by Sun and many of the Java development suites include remote debuggers. The current generation of debuggers can communicate over TCP/IP with any virtual machine that supports the debugger API. However, they cannot yet communicate with an embedded target over a serial interface.

**Does this make sense?**

Embedded developers usually have a different set of criterion for evaluating a language than other software developers. Even if Java is the best language for developers of commercial applications and Web applets, it may still not be appropriate as an embedded language. Among the most important requirements for embedded languages are deterministic behavior, small memory footprint, and efficient execution.

The key issue in real-time systems is deterministic behavior. Unfortunately, the current generation of garbage collectors is inherently non-deterministic. Moreover, garbage collection is an integral part of the Java language. Any variables that are not primitive types are objects. Because garbage collection cannot be eliminated from the language, several groups are working to create deterministic garbage collectors. Developers of real-time systems will not want to use Java until such alternatives become available.

Unfortunately, Java's current memory requirements are an order of magnitude too large for many embedded systems. In fact, systems based on typical 8-bit processors may not even have sufficient address space. The Java VM and core libraries alone require about 200K of code space. This requirement is in addition to those of the underlying operating system, the application itself, and a rather large heap for Java objects (the larger the heap, the more deterministic the garbage collection becomes). For the foreseeable future, it probably only makes sense to use Java in systems based on 16 or 32-bit processors with at least 1MB of RAM.

Java is not the most efficient language either. For now, embedded developers wanting to use Java must make do with a Java VM. Unfortunately, that means slow execution—sometimes less than 10% as fast as a similar program written in C. In the long run, performance will become less of an issue as embedded JIT compilers, AOT compilers, and JavaChips begin to appear.

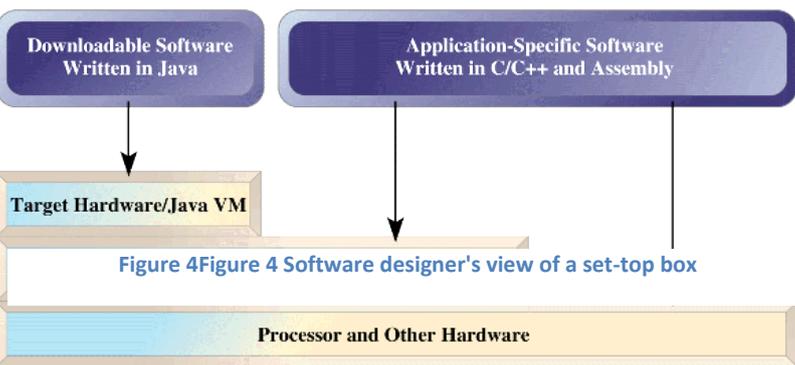


Figure 4 Software designer's view of a set-top box

Figure 4.

In this scenario, the application-specific software would be developed using current cross-compilers, assemblers, debuggers, and other tools. The RTOS and Java VM are commercial software products that would be linked with the application-specific software and stored in ROM. At the time of manufacture, the Java VM would not have any programs to execute; it would just wait idly for Java bytecodes to arrive over the network. The Java bytecodes would be created in a separate development environment, using only a Java class compiler. Once ready, the Java bytecodes could be stored on a server, where they would be available to any set-top box on the network.

**Can't wait?**

For the time being, embedded developers should probably adopt a wait-and-see attitude toward Java. If you really want to use Java today, a Java VM must offer sufficient performance for your application. That probably means most of the software will be written in another language. In the most likely scenario, small Java applications would be downloaded over a network and executed by the local Java VM.

In order to support a Java VM, an embedded system should include a commercial RTOS and have access to significant amounts of RAM and ROM. In addition, software with real-time requirements should continue to be written in C or assembly. If you need your code to be executed efficiently, you should probably wait for AOT compilers and/or JavaChips to become available. In the meantime, stick with C or C++. Java is an excellent programming language, and it will be a pleasure to use it for embedded development. However, it does not make sense to use Java until the proper tools become available.

Despite these deficiencies, however, there are some situations in which Java may already be appropriate. In fact, any sort of embedded device that connects to a network (or the Internet) is a good candidate for Java. Consider a set-top box. In such a device, most of the software would still be written in C/C++ and assembly. That software would control the basic operation of the device: receiving data from the satellite or cable network, decoding the enclosed video and audio streams, and sending images and sounds to the television.

With a Java VM in the set-top box, new functionality could be added to the device after it is manufactured. For example, Java programs could be downloaded over the satellite or cable network while browsing the Internet or during an interactive TV session. These programs would be downloaded over the network in bytecode form. Once received, they could be executed by the Java VM, as shown in