Embedded.com

Toward a Smaller Java

By Michael Barr, Embedded.com
Jun 14, 2002 (9:06 AM)
URL: http://www.embedded.com/story/OEG20020613S0026

Sun's K Virtual Machine gives embedded developers a virtual kick in the pants. KVM makes it possible to run Java programs on any 16-bit processor with 128KB of memory.

Sun's unveiling of the K Virtual Machine (KVM) at JavaOne 1999 was not particularly noted by most of the attendees, who are largely in the business of writing software for desktop and server applications. However, the significance of this announcement to the developers of software for embedded systems cannot be underestimated.

Prior to its announcement of KVM, Sun had steadfastly held to the "write once, run anywhere" principle still much associated with Java. Make software portable to any computing platform, Java's proponents said, and its value will increase manyfold. True enough. But that argument overlooks the simple fact that the majority of the world's computing platforms are inside a heterogenous mix of products with more limited resources than general-purpose computers.[1]

Embedded systems typically have only the minimum CPU horsepower, ROM, RAM, and display capabilities necessary to get the job done. Even cell phones, PDAs, and set-top boxes-high-end embedded systems by almost any standards, and platforms frequently described as "ideal for Java"-are far more resource constrained than desktops. Trying to run a Java program written for a desktop computer on one of these systems is a fool's errand.

The problem with the "write once, run anywhere" mantra is that it doesn't acknowledge the reality of resource constraints or offer would-be Java programmers any other option. Java 2 Micro Edition (J2ME) and the K Virtual Machine do both.

J is for Java

Although platform independence has often 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. And, though they based their language on the syntax of C, the creators of Java eliminated many of that language's most troublesome features. These features sometimes make C/C++ programs hard to understand and maintain, and 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, no matter what processor lies underneath.
- Automatic run-time bounds-checking prevents the program from writing or reading past the end of an array.
- All test conditions must return a Boolean result. Common C/C++ programming mistakes, such as while (x = 3), are thus detected at compile time, eliminating an entire class of bugs.

In addition, Java is an object-oriented language, which allows software developers to encapsulate new data types and the functions that manipulate them into logical units called classes. Encapsulation, polymorphism, and inheritance (the three pillars of object-oriented programming) are all available and are used extensively in the built-in class libraries. Java simplifies inheritance

by eliminating multiple inheritance and replacing it with interfaces. It also adds new features that are not available in C++, most notably:

- Automatic garbage collection simplifies dynamic memory management and eliminates memory leaks.
- Built-in language support makes multithreaded applications written in Java more portable than those written in other languages, by providing a consistent thread and synchronization API across all operating systems.

Virtual reality

The virtual machine concept is not unique to Java; it arises from the marriage of two simpler ideas. First, that the work of computer programmers everywhere would be much easier if a single processor architecture was used in every system. And, second, that simulation of one processor by another is always possible and, given sufficient computing power, often reasonable.

The creators of Java designed both a new language and a mythical processor on which all of the programs written in that language would run. Since this processor did not actually exist, they termed it the Java Virtual Machine. The written specification of this "processor" (called The Java Virtual Machine Specification and available in book form[2]) describes a full set of machine-language instructions and their behaviors and reads much like the Programmer's Guide for a real processor. The instruction set recognized by the virtual machine is called the Java bytecodes.[3]

Since the Java processor did not actually exist in hardware (at the time, anyway) it was necessary to simulate it in software. Toward that end, sun developed the first Java Virtual Machine (JVM). The first JVM was a bytecode interpreter, which translates each Java bytecode into one or more of the opcodes of the underlying physical processor, at runtime. An interpreter like this retranslates a bytecode each time it is fetched from memory. Obviously, this slows down the execution of the Java program (think of it as requiring additional processing power). Consider the impact of reinterpretation on a simple for() loop and its inner statements.

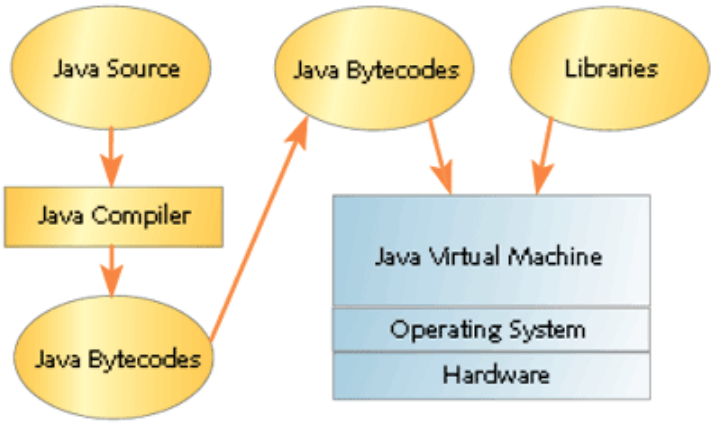

Figure 1: The Java development and execution infrastructure

The overall development and execution infrastructure for a Java program is shown in Figure 1. A program written in the Java programming language is compiled into a set of bytecodes. Those bytecodes are then loaded and executed by a JVM. If the program makes calls to other Java classes, the bytecodes for those classes will likewise be loaded, dynamically linked, and executed. Some of these libraries (java.lang, java.math, java.io, and so on) are intended to be a built-in part of any standard Java execution environment, much as the C standard library is a part of the ANSI C standard.

The principal advantage of the virtual machine concept is portability. Programs written in Java can be executed on any processor for which a simulator (JVM) exists. If the behavior of the virtual machine is defined well enough to ensure consistency across all platforms, applications will produce the same result in each such computing environment.

Getting personal

Of course, portability is about more than just the JVM. A mechanism for executing Java bytecodes on any processor is one-half of application portability; a common set of class libraries is the other.

Java's class libraries are not unlike C's standard libraries. If you've ever used strcmp() or strlen() in a program, you were relying on the standard C library to be linked with your application. Similarly, if you want to manipulate strings in Java, you will need a class library called java.lang to be made part of your runtime environment.

In order to promote and encourage application portability, Sun initially defined several standard runtime platforms (groups of class libraries). Sun referred to these standard platforms as Java Application Environments. The following application environments were defined:

- Standard Java-the full set of class libraries included in Sun's JDK. These classes are appropriate for desktop workstations and servers and may expect significant hardware and operating system resources.
- EnterpriseJava-a superset of the above class libraries, for use in "enterprise" servers and systems.
- PersonalJava-a (not-quite proper) subset of the Standard Java class libraries more appropriate to set-top boxes, PDAs, network computers, and other "networked embedded systems" with a fairly large amount of processing power and memory.

Divergent libraries were necessary because of the enormous growth of the "standard" libraries as Java was initially embraced and adapted by the Internet and enterprise programming communities. See Table 1.

Table 1: Growth of Java's standard libraries

JDK version	Year released	Packages	Classes	Interfaces
1	1996	8	172	40
1.1	1997	23	391	113
1.2	1998	62	1,287	305

The intention of these three standard application environments was to allow Java application developers to easily specify the types of platforms on which their program would run. For example, a program written for use in a PersonalJava-compatible PDA could also be run on a PersonalJava-compatible set-top box.

Embedded Java

Of course, one of the big problems with using Java in embedded systems is size, a requirement which is at direct odds with virtual machines and large standard libraries.

Sun's original JVM-and one they still license for use in all manner of systems-consumes 512KB of ROM. Included in that figure are the basic elements of bytecode execution: bytecode interpreter, garbage collector, dynamic class loader, and bytecode verifier. Not included are the underlying operating system, your application's bytecodes, and any supporting native code. Even the Java class libraries-required to achieve true portability-are not included in this sizable memory requirement. If you set out to use Sun's JVM and want to include a set of class libraries conforming to the PersonalJava API, you'd soon wind up with a system requirement of about 2MB of ROM.

Several vendors make more embeddable JVMs (smaller and faster, with more deterministic garbage collection, and so on), but you still generally need at least 1MB of ROM to achieve compatibility with the PersonalJava API.

If you're willing to discard portability altogether, you can reduce the ROM requirements significantly. The trade-off between portability and ROM usage can be made in one or both of two places: the JVM or the class libraries. The first Sun-supported approach was to shrink the libraries. That was what they termed the EmbeddedJava API.

Not a true application environment, like the three mentioned earlier, EmbeddedJava simply duplicates the list of class libraries from PersonalJava but makes every library class, method, and data member optional. In other words, if your particular application doesn't require java.net (no networking), don't include those classes in your build. This will obviously reduce application portability (for example, you can no longer run a network time synchronization program), but if that matters less to you than ROM space, you have that option.

The idea is that you write your application to the PersonalJava API, compile it to bytecodes, then run it through additional development steps called filtering and code compaction. A command-line tool called JavaFilter reads your application's bytecodes and produces a list of all the methods and data members of the class libraries on which it depends. Another tool, called JavaCodeCompact, then produces a minimal ROM image for that application by culling elements of the full set of class libraries that aren't used.

An application written for one EmbeddedJava platform may or may not run on another EmbeddedJava platform. That will depend on the library dependencies of the particular application and the precise set of libraries present in ROM on the new system. In most cases, they won't match up properly.

K is for kilobytes

EmbeddedJava was definitely a step in the right direction for developers working in constrained environments. And it was the first evidence that the folks at Sun were beginning to understand that Java wouldn't be very successful in the embedded systems market without some concessions on the portability front. However, EmbeddedJava didn't go far enough.

To truly minimize the ROM requirements for a Java execution environment, developers need to be able to fiddle with the JVM itself, to reduce features in exchange for memory.

The K Virtual Machine (KVM) provides that necessary flexibility. Built from the ground up with efficiency, minimizing RAM and ROM usage, and modularity in mind, KVM is a remarkable piece of software. It is provided in source code form, through Sun's Community Source Licensing Program,[4] and can be customized in many ways.

With KVM, it is possible to run Java applications on 16-bit processors running at just 25MHz and in systems with as little as 128KB of ROM to spare. This is a significant step forward for embedded Java. Sun's support for this approach officially opens the door to its many competitors in the embedded market to provide their own smaller JVMs.

Under the microscope

Shortly after the announcement of KVM, which I'll have more to say about later, came the release of Java 2 Micro Edition (J2ME). Three Java editions-the new name for the standard APIs, starting in Java 2-were created to promote the upward compatibility of applications written in Java. J2ME offers a way to recapture some of the benefits of "write once, run anywhere" that were lost in the fragmented set of platforms that the introduction of EmbeddedJava heralded.

With every platform having a potentially unique set of library features in ROM, the rules for EmbeddedJava meant that almost no application code would be portable. J2ME promises some of the same size reduction with a corresponding increase in portability.

The key is subsetting. If the platform that an application was written for contains a subset of the library features on desktop and server systems, then it can be run on those too. And if, for example, the very same library subset is present on two different manufacturer's cell phones, then an application written for either could be run on both.

The three Java editions are called "Enterprise" (J2EE), "Standard" (J2SE), and "Micro" (J2ME). The approximate goal is to have each library set be a subset of the previous. However, J2ME is not quite a true subset of J2SE. Just as PersonalJava added a few additions and twists to the StandardJava API, so does J2ME. These are mostly in the areas of more limited and diverse display capabilities and user input methods, and greater timer granularity.

Configurations and profiles

Within the J2ME libraries, there are several types of subsets. The first category of these are termed configurations. The currently defined configurations are Connected Device Configuration (CDC) and Connected Limited Device Configuration (CLDC).

CDC is a set of libraries meant for use in devices that plug into the wall, are always on, and have semi-permanent connections to the Internet.

CLDC, which is a proper subset of CDC, is the library set that will be of most interest to the embedded community. CLDC focuses on small devices that are battery-powered and have slow and spotty connections to the Internet, if they have such connections at all. In other words, typical embedded devices with limited resources.

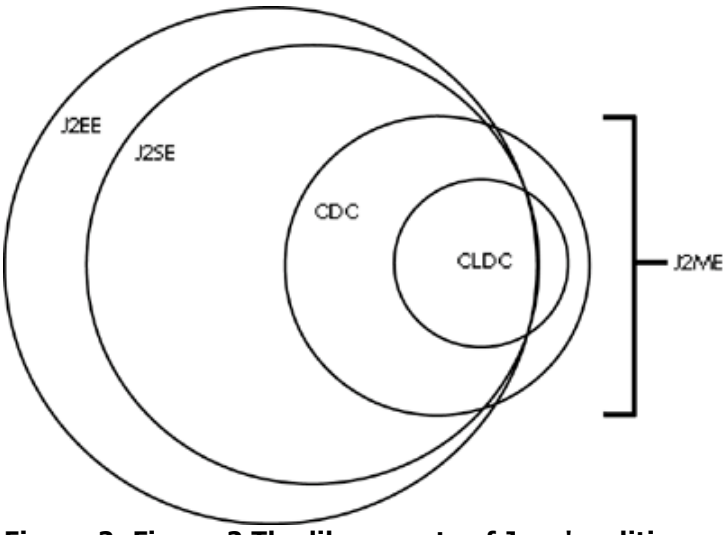

Figure 2: Figure 2 The library sets of Java's editions

Figure 2 shows how the libraries of the two J2ME configurations are related to the libraries of the other Java editions. The CDC and CLDC classes are not a proper subset of the J2SE classes.

For maximum portability of applications, the designers of an embedded system that would run Java applications should select either CDC or CLDC as their library configuration.

Building on the configuration's library framework are one or more optional profiles. These are extensions of the standard libraries specific to a particular class of embedded system. For example, the Mobile Information Device Profile (MIDP) adds class libraries specific to mobile devices like cell phones and PDAs. Any embedded system with a CLDC configuration and MIDP profile in ROM could run applications written for any other platform with those same libraries.

This allows a more reasonable, trade-off--centered approach toward portability. It is certainly much more likely that the owner of a cell phone from one manufacturer would want to run a Java utility, application, or game written for another brand of cell phone, than that the owner of a logic analyzer would want to run Java code originally written for use on a microwave oven. To gain such portability, cell phone manufacturers might be willing to add a few libraries (and a corresponding amount of ROM) that microwave oven manufacturers would not.

J2ME's configurations and profiles are defined by technical partnerships involving Sun and its customers, under the guidelines of the Java Community Process.[5]

Spotless

KVM was first envisioned as a Java virtual machine for Palm handhelds, and it is in that environment that embedded developers can best exeriment with and appreciate it and the J2ME libraries.

The MIDP for Palm OS website (java.sun.com/products/midp4palm/) is the place to start. The code and tools you'll find there help Palm OS users run Java applications that are available for other MIDP-compliant devices, such as mobile phones and pagers.

The full CLDC and KVM distribution, which targets the Windows, Solaris, and Linux platforms, is available at java.sun.com/products/cldc/. This code has been designed for easy portability. A helpful and well-written "KVM Porting Guide" is included. And all of the CLDC library code is written in Java and supplied in bytecode form, which KVM can execute once you get it up and running on your target platform.

The first thing to know about KVM is that it's written entirely in ANSI C. It is done this way to make it easy to target any 16- or 32-bit processor for which a C cross compiler exists. Other than implementing a couple of platform-specific functions in C, there's not much to the porting process.

But wait! How can KVM execute platform-independent Java bytecodes on your specific processor without some platform-specific assembly code? Easily! All the authors of KVM had to do was code the equivalent of each Java bytecode in C:

```
switch (bytecode)
{
case 0x01:
...;
case 0x02:
...;
...
};
```

Your compiler does the rest for them automatically-by turning each C snippet into one or more native opcodes. This may not produce the most efficient translation from each bytecode to the native opcode(s) of your specific processor, but it does make the KVM code extremely portable.

The only platform-dependent functions you'll write to port KVM are concerned with allocating big chunks of memory for heap expansion, deallocating the same, returning the current system time, and handling user interface issues. There are also some (optional) hooks for doing platform-specific stuff during VM startup and exit.

As provided, the source code has been tested on Solaris (using GCC or Sun DevPro C 4.2), Windows (Visual C++ 6.0), Linux (GCC), and Palm OS (CodeWarrior Release 6 for Palm). Makefiles and platform-dependent code for each of these platforms are provided for you.

In the next section, I will refer only to the Palm OS port, which I experimented with myself. I began with the code for KVM 1.0.3 and used Metrowerks CodeWarrior, running on a Windows 2000 development host, to tweak the Palm OS port to my liking.

Size matters

The size of the KVM executable is heavily influenced by the Java language features selected when it is compiled. Configuration options include:

- Floating-point support can be enabled or disabled.
- Internal debug logging and tracing code can be included or excluded.
- The maximum heap size can be set.
- A runtime bytecode caching technique ("fast bytecodes"), which improves performance 15% to 20% can be turned on, at the cost of several kilobytes of extra code.
- Either big- or little-endian data mode can be selected.

- Class prelinking/preloading can be turned on or off.
- Some behaviors of the garbage collector can be tweaked.
- Optional non-CLDC classes, like graphics libraries, can be included or excluded.

Included in the KVM download are pre-built KVM.prc and KVMutil.prc files, which are required for download to a Palm OS device before Java applications can be run there. Somewhat to my surprise, these default files together took up almost 400KB of memory on my Palm V. But that seems to be the maximum, with all options and CLDC libraries included. (The MIDP libraries are not included in this figure.)

I was able to quickly build a much smaller standalone KVM for the Palm (about 60KB), simply by turning off floating-point support, enabling prelinking/preloading, disabling "fast bytecodes," and "ROMizing." The latter means going through an automated, EmbeddedJava-style pruning of the class libraries at build time, to include in memory only those classes, methods, and data members that are actually used by the applications you want to run.

Typical Java applications for the Palm-like games and calculators-load more slowly (one to two seconds) but otherwise execute as responsively as native Palm apps. (It appears the KVM restarts each time a new application is run.) In addition, most of the Java apps I played with were pretty small-about 5KB to 15KB each.

Write once, run anywhere?

Overall, I am impressed by the J2ME concept and the KVM implementation. Some initial concerns about garbage collection were addressed in release 1.0.3. And, though the new collector is not deterministic, it is far more responsive and respectable than its predecessor.

One remaining complaint concerns the threading model, which has yet to be improved in any update. Under KVM, each Java thread executes a fixed number of bytecodes before being preempted by the scheduler. This overly simplistic round-robin approach needs to be replaced by a priority-based preemptive scheduler to make KVM more than just a platform for experimentation.

Additionally, the virtual machine enhancements suggested in the Real-Time Specification for Java**[6]** would greatly enhance the value of KVM to real-time developers. Some of the proposals address determinism, others garbage collection and thread scheduling. Perhaps now that KVM's source code is out there, a helpful individual or group will come along and make these changes for the benefit of all.

In summary, KVM represents something close to a JVM embedded programmers could live with. There are still some issues of efficiency and determinism to deal with, but those are not of universal concern. By putting the trade-offs of size vs. portability in the hands of the developers-and in the process, minimizing insistence on portability at all cost-Sun has given us a very nice way to run Java programs on simpler processors with far less memory than ever before. And J2ME's configurations and profiles provide a flexible roadmap for achieving a desired level of portability at a more reasonable cost.

Michael Barr is the editor in chief of *Embedded Systems Programming.* He is a lecturer at the University of Maryland and a long-time presenter at the Embedded Systems Conferences. He wrote *Programming Embedded Systems in C and C++* (O'Reilly and Assoc., 1999). Michael received his BS and MS degrees in electrical engineering from the University of Maryland. You can e-mail him at mbarr@cmp.com.

Endnotes

1. "If you round off the fractions, embedded systems consume 100% of the worldwide production of microprocessors." -Jim Turley, processor industry analyst (*ESP,* May 1999)
Back

2. Lindholm, Tim and Frank Yellin. *The Java Virtual Machine Specification.* New York: Addison-Wesley, 1999.
Back

3. The term bytecode derives from the fact that each instruction is just one byte wide.
Back

4. The long and short of Sun's Community Source Licensing Program is that developers can have full access to the source code for the libraries and virtual machine while they are doing research or creating a product. They can even make changes to that code that suit their system's needs. However, before they start shipping a product that includes that software, they need to enter into a licensing agreement with Sun. For more information, read www.sun.com/software/communitysource/faq.html.
Back

5. www.jcp.org
Back

6. www.rtj.org
Back