JAVA FOR REAL-TIME TELEMETRY SYSTEMS

Gilles K/Bidy
Engineering Product Manager
L-3 Communications, Telemetry West
9020 Balboa Avenue
San Diego, CA 92123

ABSTRACT

Because of an ever-increasing need for performance and high predictability in modern real-time telemetry systems, the Java programming language is typically not considered a viable option for embedded software development. Nevertheless, the Java platform provides many features that can easily be applied to embedded telemetry systems that other development platforms cannot match. But obviously, there are pitfalls to be aware of. This paper will present an alternative solution to address today's problems in real-time telemetry systems and will cover the following topics:

- Java development platforms for the embedded world
- Impact on software portability and reusability
- Performance and optimization techniques
- Direct access to hardware devices
- Memory management and garbage collection
- Network-centric component-oriented architecture
- Real-time examples from past experience
- Future developments

KEYWORDS

Java, Embedded Systems, Real-Time, Telemetry

INTRODUCTION

Born in 1995, the Java programming language was primarily conceived and developed as an alternative to traditional high-level languages such as C and C++. It was originally targeted for platforms such as Sun Solaris, Microsoft Windows, and most importantly the World Wide Web through the use of Internet browsers.
Three years later, after a very successful start, it was decided to expand the use of this state-of-the-art programming language to the world of embedded computing, from full-blown multi-processor systems to small devices such as cell phones and PDAs.
However, Java concepts cannot be directly applied to real-time embedded platforms without taking into consideration the constraints of such environments: limited system resources (processing power, memory and I/O interfaces), high reliability, security, etc…
These requirements demand a solution providing both flexibility and scalability. The goal of this paper will be to determine if and how Java can be successfully applied to the world of embedded computing.

THE JAVA™ PLATFORM

Running on top of the existing operating system, the Java platform provides both a development and runtime environment via a virtual machine. A Java program is nothing else but a set of source files compiled to binary machine-independent byte codes that can be executed (or more precisely interpreted) by the virtual machine. The following diagram illustrates a typical java compilation and runtime process:

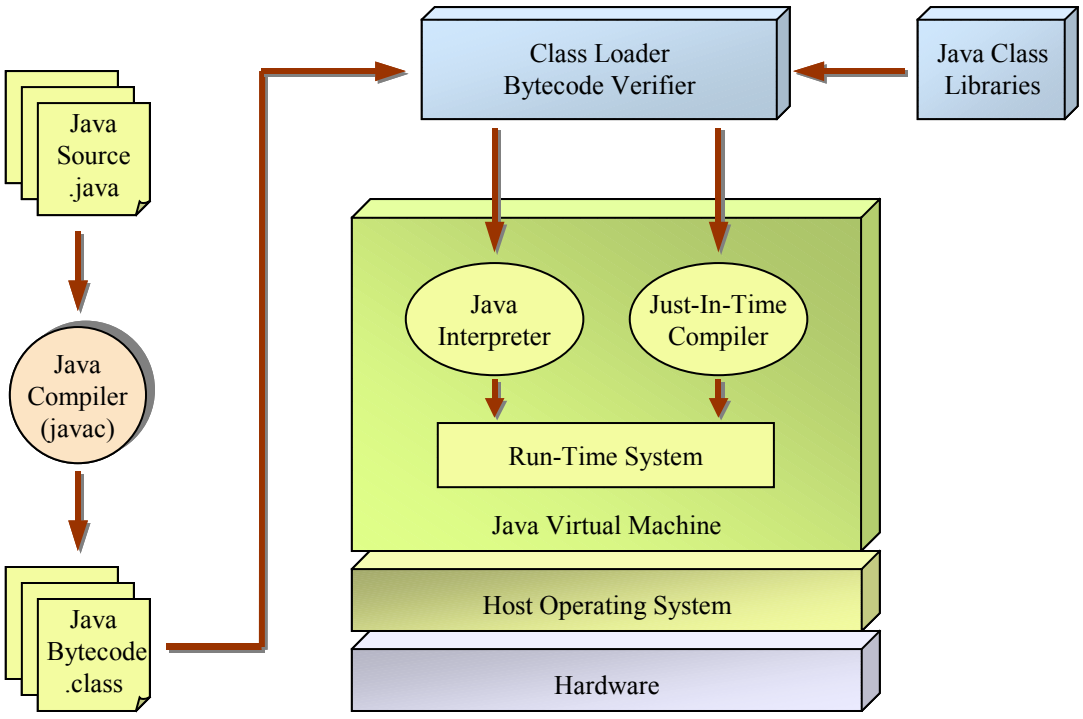

Figure 1 – Java Compilation and Execution Overview

Therefore, Java is not only a high-level object-oriented programming language but also a software development platform that includes a runtime environment. In fact, the Java platform is comprised of multiple layers that can be used together or separately to address specific programming needs:

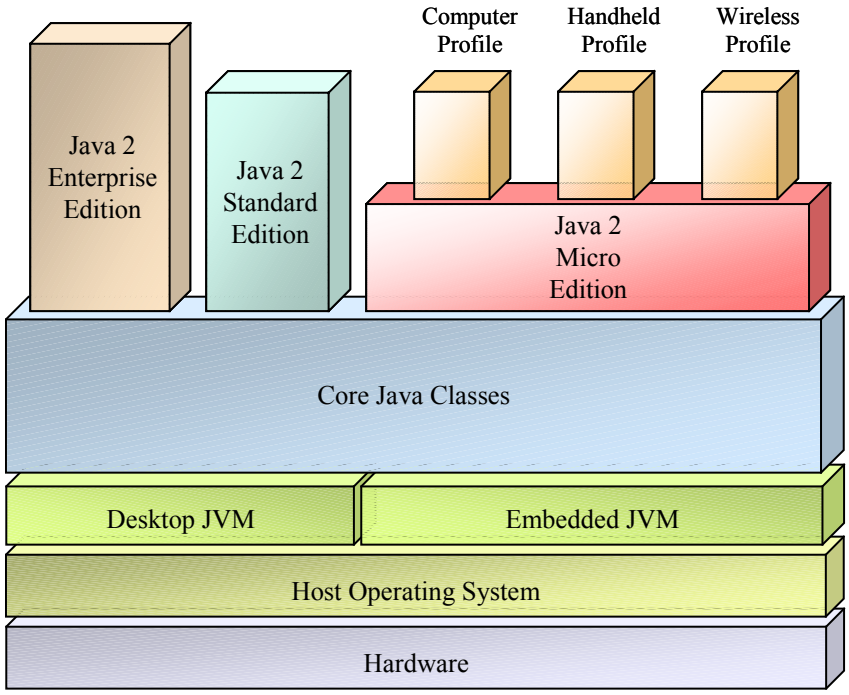

Figure 2 – Java Software Platforms

With the introduction of Java 2 (i.e. JDK 1.2.2) in 1999, Sun Microsystems decided to deploy three different Java platforms:

- The Java 2 Enterprise Edition (J2EE) geared toward e-commerce and business/enterprise software applications
- The Java 2 Standard Edition (J2SE) addressing software needs for desktop and workstation applications
- The Java 2 Micro Edition (J2ME) focused on consumer electronics and more importantly embedded devices.

In 2002, the most up-to-date release of these Java 2 platforms was version 1.4, available as of last April.

As far as embedded development is concerned, an initial specification called Personal Java was released by Sun in 1998. A few vendors including Sun offered implementations for various targets such as PowerPC and Intel processor boards and basically gave life to Java in embedded targets for the first time. It is one of these implementations that L-3 used and deployed on their high-end telemetry processor 2 years ago.

More details will follow on current and future specifications for the J2ME platform in the last section of this paper.

In parallel to the J2ME developments, a few Java experts have created a specification for a real-time environment for Java. This specification as well as preliminary implementations will be discussed later on.

SPOTLIGHT ON JAVA AS A PROGRAMMING LANGUAGE

Developed as a replacement for traditional and proven high-level languages such as C and C++, Java is overall a simple object-oriented language, designed for rapid development through software reusability and robust computing guaranteed by a new memory management scheme.

For the telemetry software engineer, Java provides a high level of reliability and typically shortens development cycles, both criteria being equally important in flight test or space communities.

Core features of the Java language include standard platform-independent data types, absence of pointers (cause of nightmares for C and C++ programmers), advanced error management (including runtime verifications for out-of-bound writing, stack overflow or division by zero situations), dynamic loading and discovery of classes, etc.

In addition to these basic (but essential) programming elements, the Java language supports a plethora of library classes to address such needs as networking, multi-threading, SQL database management, and internationalized user interfaces…

In the context of embedded programming, there is a crucial need for full control of memory and hard drive resources. For that very purpose, it is usually possible to customize an embedded Java virtual machine to only include classes of interest for a particular program. Furthermore, networked configurations can benefit from the dynamic class loading aspect of Java to access certain classes or libraries as needed during execution.

The following diagram illustrates how Java classes can be organized within a virtual machine:

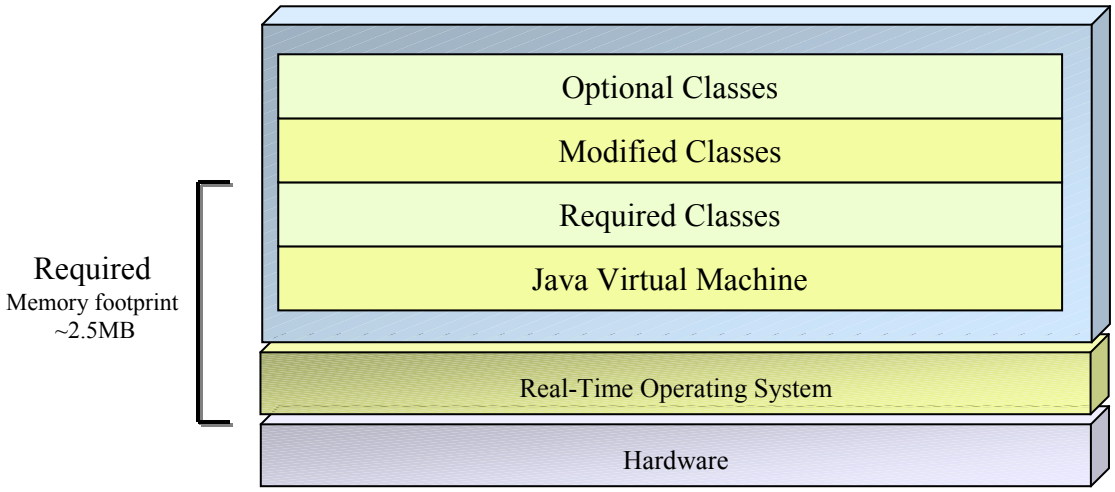

Figure 3 – Java Virtual Machine Customization

SOFTWARE PORTABILITY

One of the advantages that Java brings to the programmer is platform independence. Benefits of this approach for software development cannot be overemphasized. Indeed, once true platform independence is achieved, the programmer can focus on a single body of code and not worry about platform specific details - such as file I/O, native data types or graphic primitives to name a few…

It was not until version 1.2+ that Java provided true, efficient platform independence, including all graphical widgets and primitives.

Platform compatibility issues are now pushed down one level to the virtual machine layer. Conveniently, Java 2 compliant virtual machines have been made available on most commonly used operating systems.

Embedded developers usually face a significant amount of hardware configurations, making it difficult to port an application from one platform to the next. Standard specifications or APIs are non-existent, creating a need for rapid software portability. Although Java may not address all of the embedded programmer's needs, it does provide a portable framework for commonly used sections of code, not requiring hard real-time performance.

For standard telemetry software, L-3 has found that 75% of the embedded C code (sometimes even more) could be ported to Java and become platform-independent as well as network friendly.

The following diagram illustrates a common technique to make the most of Java in an embedded application without affecting real-time performance:

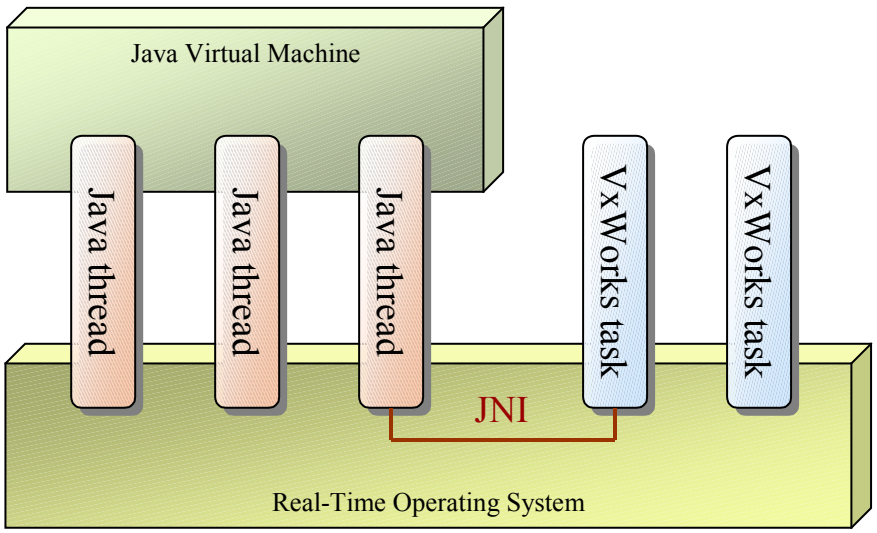

Figure 4 – Java cooperation with native code

The Java Native Interface (JNI) acts as a bridge between native resources and Java classes. It is typically used to access various memory regions or call native functions such as task management routines. JNI will be further discussed in the section dealing with direct access to hardware resources.

CODE REUSABILITY, MYTH OR REALITY?

Source code reusability is definitely not a new concept in the software engineering field. Yet, few programming languages can make such a claim. With loose object encapsulation, C++ supports code reuse, but does not make it easy for the programmer. It is commonly admitted in the programming community that Java somehow breaks the barrier of code reusability and outperforms most object-oriented languages in this regard. There are 3 key aspects of Java that make this possible. First of all, multiple inheritance is not permitted. Developers have to use interfaces to define multiple behaviors for a given class. Second, object encapsulation is strictly enforced in Java, global variables or functions outside of the scope of a class are simply not supported. Finally, Java does not have header files and uses the concept of packages. This new way of grouping a set of classes together into a reusable component greatly facilitates code sharing across applications and projects.

L-3's experience with code reuse with Java has been overall very positive. Previous attempts using C and C++ were questionable, whereas current development projects written in Java have definitely made great use of this feature.

For the embedded programmer, having access to off-the-shelf components ready to be integrated is an invaluable advantage. This allows for a larger feature set, coupled with decreased development cycles.

In the case of a telemetry processing system running a real-time operating system, L-3 was able to reuse basic initialization and control code across most supported modules, resulting in a reduction of development time. The following UML diagram is an example of code sharing across 2 telemetry modules.

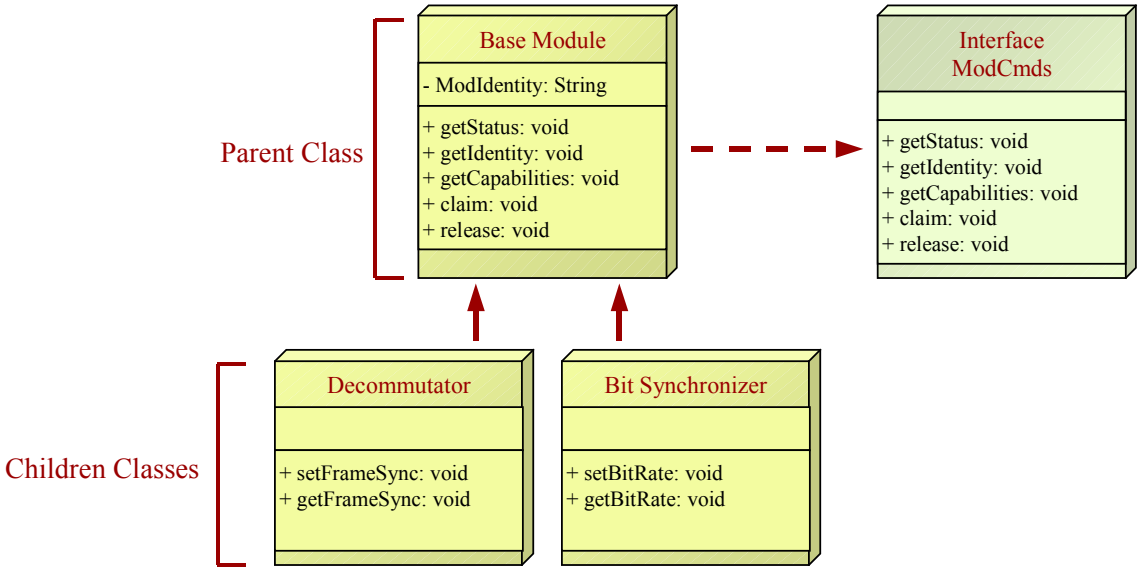

Figure 5 – Code reuse UML example

In the source code, developers only need to reference the original package using the 'import' keyword. Here's a snippet of code for the Decommutator module:

```
// Decommutator Module

import lstar.lfc.module.BaseModule;

public class Decommutator extends BaseModule
{
     // Class definition
}
```

Figure 6 – Code reuse source example

RELIABILITY & SECURITY

Yet another critical aspect of real-time embedded devices, reliability was a major design factor for Java. Many embedded systems perform mission-critical tasks and must never fail, making Java a language of choice for this type of applications. Indeed, Java enforces various safety measures, providing a viable runtime environment for critical tasks.

Java performs strong compile and runtime verifications and can detect casting errors, major runtime errors such as division by zero without crashing the application. The developer must also comply with rules of structured pure object-oriented programming through the use of classes and packages, eliminating problematic global variables for instance. Finally, last but not least, Java provides an advanced automated memory management scheme (called garbage collection), greatly simplifying object lifecycle management. Programmers need not worry about releasing memory when a given object goes out-of-scope; the garbage collector thread will silently collect unused objects (i.e. no longer referenced anywhere in the code) and free memory as needed.

Furthermore, in most embedded systems, security must be enforced and maintained at all cost. Java provides a secure execution environment where the virtual machine performs strict code verification prior to execution. Local system resources can therefore be shielded from potentially harmful code. Java's multi-threading model adds to this security aspect by imposing well-bounded execution contexts for each active thread.

NETWORK CENTRIC ARCHITECTURE

Another great facet of Java lies in its ability to interact with network applications very easily, almost seamlessly. Java was conceived around the idea that "the network is the computer" as Scott McNealy (CEO, Sun Microsystems) puts it. Distributed computing is therefore an inherent part of the Java language.

Although standard network protocols such as TCP/IP are fully supported in Java, its real strength comes from the Remote Method Invocation (RMI) classes. Vaguely similar to Remote Procedure Calls (RPC) in other languages, RMI redefines distributed computing for high-level languages. Connecting to remote objects and invoking remote methods has never been so easy.

Benefits to the embedded programmer are obvious. First of all, most standard network protocols are supported in a platform-independent way, making it very easy to establish socket connections or to run local servers in Java. Moreover, RMI allows embedded devices to exist and advertise public services seamlessly to other computers running on the same network.

As an example, L-3's high-end embedded telemetry system makes extensive use of RMI to create a set of self-contained telemetry modules readily available on the local network – and this is possible independently of how these modules are locally configured or

controlled. This approach provides redundancy and module isolation (in case of problems) and facilitates remote management from a powerful workstation for instance.

The following diagram shows a typical distributed system using L-3's design:

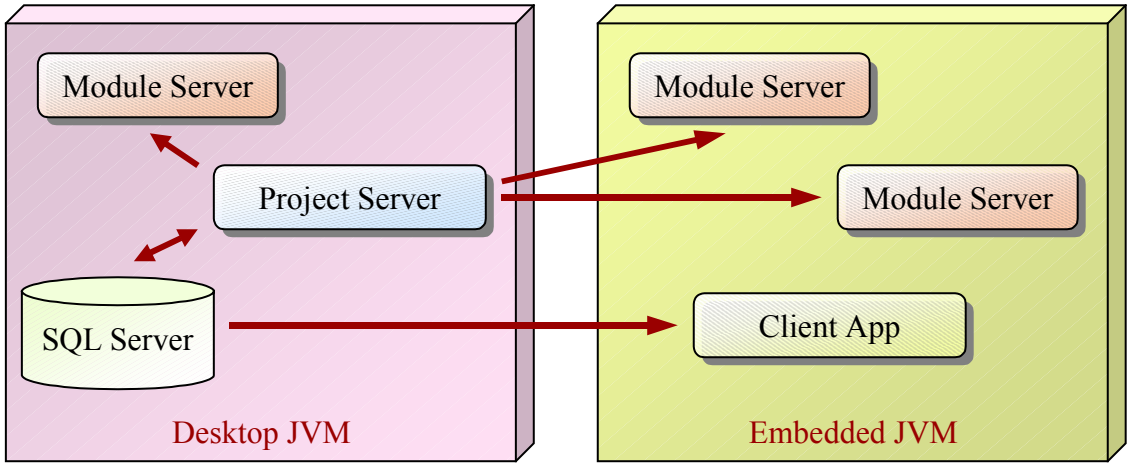

Figure 7 – Seamless distributed computing in Java

ISSUES FOR EMBEDDED COMPUTING

While Java provides many attractive features for the embedded developer, there are still several issues to overcome. There is no denying that Java has come a long way since its inception to fit the embedded programming mould. Still, certain tasks such as direct access to hardware devices remain cumbersome to say the least. The following sections will uncover various performance issues or language limitations and hopefully propose potential workarounds for the enthusiast embedded Java developer.

PERFORMANCE

Designed as an interpreted language, Java is plagued with execution speed issues. Although Java byte codes are perfect for software portability, they do not help performance, quite the contrary in fact. A major problem for certain embedded applications.

Historically, Sun and other vendors have been constantly working on performance improvements for current virtual machines. Over the years, Java execution speed has gradually improved. More particularly, the introduction of the HotSpot virtual machine brought significant changes to the picture. Nevertheless, even today, Java is still slower than native code in most cases. Graphics performance is one of the most noticeable degradations. Luckily, embedded programming rarely requires graphical capabilities.

To counter these issues, a few optimization techniques are being tested and applied to recent virtual machines. For instance, a popular way to improve runtime performance is to use a Just-In-Time (JIT) compiler as part of the virtual machine. This is essentially

used to convert Java byte codes to native machine code on-the-fly. However, JIT compilers are not well suited for embedded computing because they use a lot of memory and may cause unpredictable delays during execution. In addition, for various reasons, optimizations performed by a JIT compiler are always less effective than those performed for traditional languages (C and C++).

More advanced techniques had to be developed. Enters adaptive optimization. This new scheme allows a virtual machine to detect and analyze sections of code that are executed very often. This particular code is then compiled to native code and possibly optimized using method inlining for instance.

Specific embedded virtual machines may use similar techniques such as dynamic adaptive compilers (as in JWorks from WindRiver Systems) or ahead-of-time (AOT) compilers.

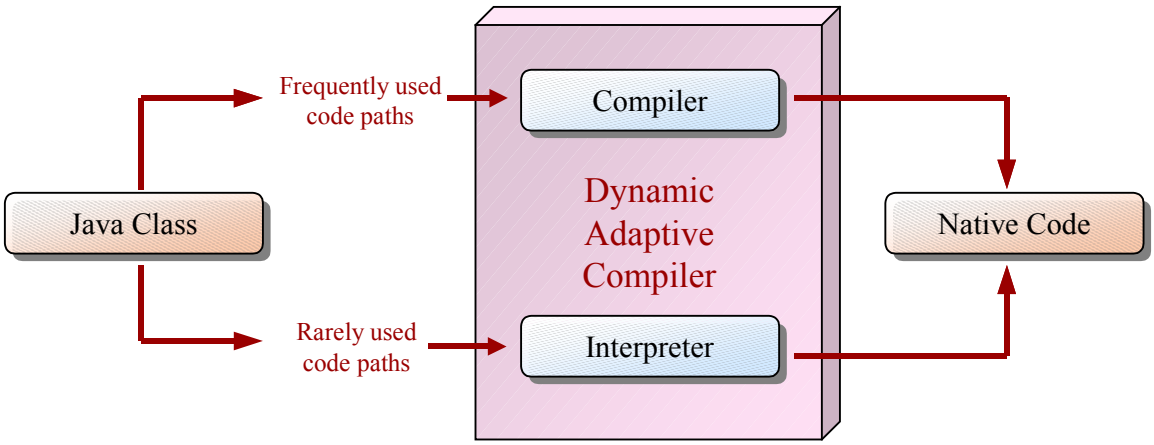

Figure 8 – Dynamic Adaptive Compiler example

Overall, despite various optimization techniques, experience has shown that execution speed remains a problem when using Java for performance-hungry applications.

Nevertheless, in most real-time operating systems, Java threads are fairly well isolated from real-time tasks. The virtual machine does map Java threads to native tasks but they can all be preempted by higher priority tasks. This works really well as long as Java is not used for critical operations.

As an example, L-3 decided to use Java for setup and control operations but kept real-time processing functions in C to address performance issues. Most telemetry hardware modules only require software interaction at setup time. For instance, a typical hardware decommutator module needs to be configured at setup time and operates on its own at runtime. So, in this case, Java runtime performance is not really an issue. But there are cases where interrupts need to be handled or real-time algorithms are needed. This is when a language like C remains superior. Consequently, it is this hybrid approach that seems to work best in most cases.

DIRECT ACCESS TO HARDWARE RESOURCES

One of the consequences of platform independence is low-level resource isolation. Basically, to ensure that Java software be 100% portable, the language prohibits any direct access to local resources. This includes operations like memory addressing, low-level I/O, video memory access, etc.

Needless to say, this is a serious limitation for any embedded developer. Fortunately, Java designers realized early on how critical this issue was and released a set of classes to access native resources through the use of native code.

The Java Native Interface (JNI) allows applications written in Java to interact with other pieces of software written in a different language such as C and C++. It is the ultimate solution when Java cannot be used. Unfortunately, JNI turns out to be quite convoluted and makes basic statements complex. Indeed, Java primitive data types and classes have to be somehow mapped into structures of the other language. This often results in artificial, cumbersome software. Embedded programming demands features such as memory access and interrupt handling. Until last year, JNI seemed to be the only way to accomplish this.

However, with Java 1.4, Sun introduced the concept of direct memory access through the use of direct buffers (java.nio.ByteBuffer). Upon allocation of a direct buffer, the JVM will make a best effort to perform native I/O operations on that particular memory area. This is a potential solution to direct memory access even if it is still pretty limiting at this point in time. If nothing else, it is a step in the right direction for future developments.

As bad as it sounds, there are still ways to make JNI interact well with Java applications. To manage various telemetry modules, L-3 developed a generic Hardware I/O library in Java and implemented its sibling in C using JNI as a bridge. As a result, programmers only see the Java interface and can seamlessly interact with physical memory, performing operations such as 32 bit read/write or block read/write, etc. Obviously, matters become worse when interrupt handling is required. The best option seems to rely on native code to register interrupt handlers in native tasks and service interrupts as they arise.

The following diagram shows how JNI can be used to mix Java and C/C++ code:

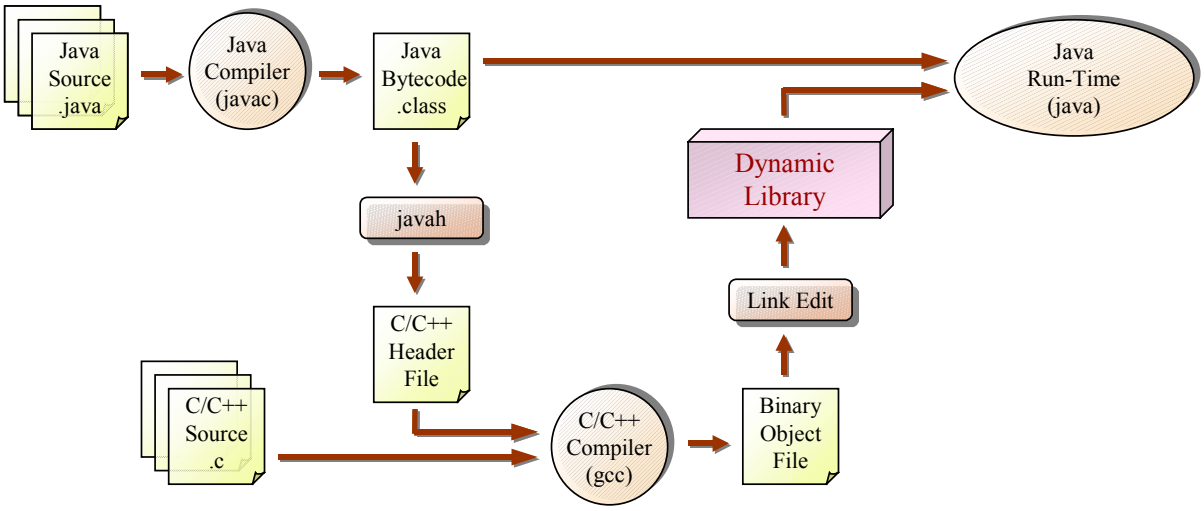

Figure 9 – JNI usage overview

MANAGED MEMORY, BLESSING OR CURSE?

Garbage collection was briefly mentioned in previous sections. It is by far one of the best features of the Java platform. Never before were programmers able to rely on the runtime environment to reclaim unused memory objects dynamically. Memory leaks are a major source of failure in languages like C and C++ but are a thing of the past in Java.

But garbage collection can also be a real nightmare for embedded programmers. This particular thread is run by the Java virtual machine itself and is usually not preemptible. Consequently, once the garbage collector starts running, it has to complete before any other Java thread can resume. Any premature termination would leave the memory in an undefined state. Furthermore, nobody can predict how long the garbage collector will run for. This is unacceptable in most embedded environments, where high predictability is required.

As with virtual machine performance issues, various garbage collection algorithms are being implemented and tested in different application contexts. Traditionally, memory is reclaimed all at once and the process cannot be preempted. Execution time is unknown. This is how initial garbage collectors were implemented, which is obviously inappropriate for embedded programs.

Recent embedded virtual machines offer solutions such as incremental garbage collection, which is a smoother clean up process, based on incremental, fixed steps. This may address the issue of consistency for execution times but is still not a preemptible process.

On the other hand, solutions like concurrent garbage collection provide preemptible algorithms with undetermined execution times.

To date, no garbage collection algorithm addresses all the problems identified above. The programmer must basically choose which algorithm to use based on the type of application to be written.

In perspective, having an automated memory management system is still beneficial to most projects. Once again, a hybrid approach making use of native code when critical performance is needed seems like the best solution. It is quite easy to implement and provides even more freedom in the Java code, since critical operations are left to native tasks and will not be affected.

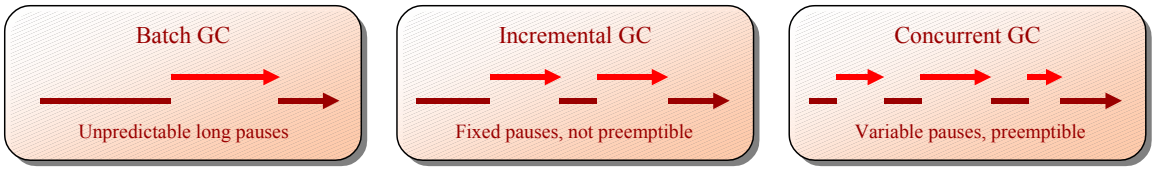

Figure 10 – Different types of garbage collectors

REAL-TIME JAVA OFFERINGS

Getting an embedded virtual machine for a given configuration is usually pretty straightforward. Most real-time operating system vendors provide one or more Java virtual machines for their platform. And they usually support multiple architectures. In addition, source code for reference implementations can also be obtained from Sun if necessary.

Most common embedded virtual machines run under VxWorks or embedded Linux. Companies like Wind River Systems, NewMonics and MontaVista provide several implementations of virtual machines for different processors.

Although J2ME 1.3 or 1.4 seems to be most up-to-date version available, it is surprising to notice that implementations such as JWorks from Wind River still rely on old specifications like Personal Java and do not have a full J2ME compliant virtual machine yet.

One concern could be that current development appears to shift more and more toward small devices like handhelds and cell phones. It seems that little effort is put into Java software for high-end embedded systems.

FUTURE DEVELOPMENTS

One of the most encouraging improvements proposed by the Java Community Process lies in the development of a real-time specification for Java. The specification document is in its final stages and beta implementations are quickly becoming available.

The specification focuses on the following topics:

- Scheduling
- Memory management (with new concepts like immortal memory)
- Synchronization
- Asynchronous event handling
- Asynchronous transfer of control
- Asynchronous thread termination
- Physical memory access
- etc…

Timers are being studied to provide accurate and precise time management and task scheduling. Threads are being improved to support stringent real-time requirements.

This is probably the most interesting Java improvement for embedded computing. Still at an infant stage, the future looks promising for this particular technology.

CONCLUSION

No tool is perfect and Java is no exception. It does offer real benefits to embedded system developers as long as they make intelligent use of it. Hybrid solutions are often needed to take advantage of Java's advanced features without sacrificing native performance. It is obviously convenient to be able to connect to an SQL database server from an embedded Java thread and to register low-level interrupt handlers at the same time. It is only a matter of making educated choices during the project design phase.
The Java language is in constant evolution and will undoubtedly become an even more attractive tool for embedded developers over time.