Java technology trends offer renewed promise for portable embedded applications

By Dave Wood

Because of the promise of increased productivity and reduced error incidence, achieving program portability has always been an important goal in software engineering. The goal of portability has been met with mixed success because of varying approaches to portability from one programming language to another. Further, the special characteristics of embedded, real-time, and safety-critical applications present additional challenges to portability. Dave examines the characteristics of C++, Ada, and Java and their implications for portability in embedded development.

Creating highly portable code is an important mechanism for increasing software development productivity while improving software maintenance and longevity. Today's primary languages for military embedded systems – C++ and Ada – along with emerging embedded Java technology, each provide their own portability characteristics.

Not all portability is created equal

C and C++ programs often are presumed to be portable. Consider the following code:

```
#include <stdio.h>

main()
{
     printf ("This code is
portable!\n");
}
```

This piece of code will compile and run on practically every commercial processor/operating system combination in existence. The fact that it will do so is not a reflection of anything inherently portable about the language itself, but rather it is a reflection of the fact that C or C++ compilers are available on essentially every platform. As such, C qualifies as a portable language, at least so long as the programs adhere to commonly supported core syntax. Thus C/C++ are portable in the sense of *platform pervasiveness*.

On the other hand, Ada was designed specifically to promote code portability with an emphasis on tightly controlled international standardization and rigorous conformance test suites. Although Ada includes many implementation-dependent features, its portability features provide substantial built-in standard libraries and a standard multi-threaded execution model. As such, Ada also qualifies as a portable language. Yet, although Ada compilers are available on a wide array of platforms, they are not nearly as widely available as C/C++ compilers. Therefore, we consider Ada portable in the sense of *platform independence*.

C/C++ is portable thanks to ubiquity rather than design. Ada is portable thanks to design rather than ubiquity. These kinds of distinctions form a rich basis for "my language is better than your language" debates among advocates.

Defining portability

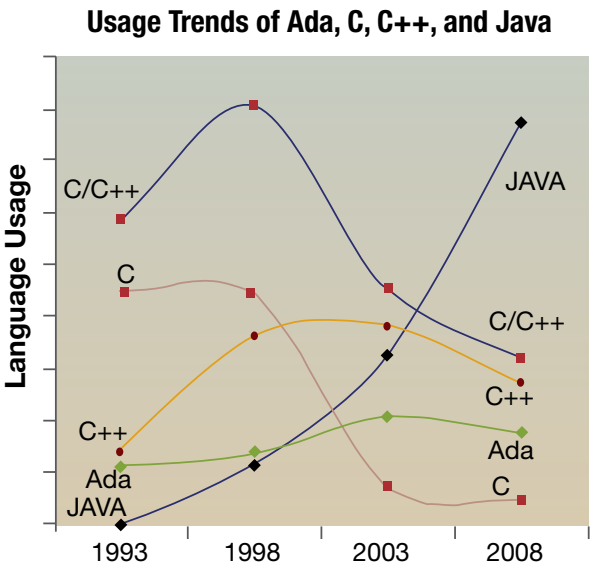
Clearly, portability means different things to different people. By evaluating the merits of various portability paradigms against the actual needs of a particular audience, we can make value judgments about the suitability of a given language's portability characteristics. Table 1 lists cross-platform portability factors, and their relative importance is dependent upon individual needs.

Choosing between C++ and Ada offers a trade-off between a more portable design (Ada) and a wider potential pool of available platforms (C++). A third option, Java, offers the promise of both platform pervasiveness and platform independence at the same time.

In the nonembedded software community, Java has already emerged as the favorite. Studies indicate that Java popularity has

Component	Possible Cross-platform Portability Requirements
Compiler	Compilers for language exist on required platforms
	Compilers exist and conform to same standard
Libraries	Common library types are commercially available
	Common libraries inclusive in language standard
Source Code	Compatible across platforms and OS versions
	Compatible across variants of language and libraries
	GUI code same across platforms without change
	No exposed OS or hardware dependencies
Runtime	Same runtime API across platforms
	Consistent runtime behavior across platforms

Table 1



Data Source: *Programming Language Trends, An Empirical Study of Programming Language Trends,* Yaofei Chen, PhD dissertation, New Jersey Institute of Technology, August 2003.

Figure 1

overtaken that of C/C++ in enterprise and desktop computing (see Figure 1), and is available on most computing platforms. At the same time, Java's design features mirror and even exceed many of the high-portability characteristics of Ada. This is a reflection of the "Write Once – Run Anywhere" philosophy that drove the design of the Java language, the Java library set, and the Java virtual machine.

As seen in Table 2, C++, Ada, and Java each offers its own strengths. For military embedded applications, the question is which of these options presents the best match to priorities. To help answer that question, three key areas to consider are source code portability, library portability, and behavioral portability.

Source code portability

At the source code level, it is certainly possible to write portable C/C++ code, given careful attention to coding practices. However, C/C++ presents many opportunities for making nonportable code. An example of some of the thought processes recommended for C/C++ programmers to improve portability can be found in *The C++ Portability Guide* (see sidebar). This publication identifies a number of areas where C and C++ programmers need to tread carefully in order to achieve higher portability.

In contrast to C++, Ada is a rigorously specified language with very well-defined and longstanding international standardization. Even so, Annex M of the language standard identifies more than 100 implementation-dependent aspects of the language that should be carefully encapsulated to facilitate portability.

An example of an area where a programmer needs to take extra care with C/C++ is with the type system. C/C++ typing is comparatively loose and implementation dependent. Details of atomic types such as type size (for example, the number of bits in shorts, ints, and longs) as well as "endianness" are implementation-specific. Other nonportable behavior includes the value of uninitialized variables and the results of accessing memory that has been freed. For Java, details and behaviors of this sort are either explicitly defined and abstracted away from the programmer or simply not applicable.

Library portability

Building complex systems is not practical without the availability of off-the-shelf libraries. Many such libraries exist for all three of the languages discussed herein, but not all are equally portable. Libraries can be considered most portable if they are defined as part of the standard, if the content of the standard libraries is comprehensive, and if the standard is widely adopted across implementations.

In this regard, Java clearly excels to levels far beyond any competing offering. The core Standard Edition libraries alone consist of 166 packages and 3,279 classes. Although the C/C++ ecosystem provides an extensive collection of open source and commercial libraries, the libraries that are inclusive in the language standard are comparatively Spartan. Library support can vary from implementation to implementation and from platform to platform, with the consequence of having an impact on portability.

The standard Ada library set is consistent across implementations and very well-defined. In comparison, the scope of the standardized set of Java libraries is staggering and growing because of the participation of a large, cooperative, and vibrant development community. Like C/C++, Java enjoys a very large ecosystem beyond the standard such as APIs for 3-D graphics, OpenGL, compression, and numerical analysis. Some libraries have achieved quasi-standard status and ubiquitous availability. An example of this is the SWT graphics library from the Eclipse Foundation.

Portability Factor	Ada	C++	Java
Platforms	moderate	pervasive	growing
Standardization	high	moderate	high
Built-in Libraries	moderate	minimal	extensive
Third Party Libraries	moderate	extensive	extensive
Commercial Use	minimal	extensive	extensive
Runtime Behavior	defined	undefined	defined
Safe Programming	yes	no	yes

Table 2

Behavioral portability

Even if portability is achieved at the source code level and at the library level, the runtime behavior of a program may be very different from one platform to another. In the case of C/C++, runtime behavior is undefined. C/C++ does not specify mechanisms for execution threads, priorities, or even interprocess communications. Thus, these types of issues are relegated to the operating system, and the interfaces are application-specific.

In contrast, both Java and Ada specify a clear runtime semantic. Java goes further than Ada in specifying not only semantics related to thread execution and communication, but also in defining an abstracted operating system complete with its own virtual machine code and memory management known as *byte code* and *garbage collection*, respectively.

For a particular application, all of these runtime characteristics may be fully implemented within the Java Virtual Machine, or may be partially mapped to underlying operating system functionality where available. In any case, the application code is unaware of the implementation layer, and so is completely portable.

It is this sense of portability that makes Java so attractive for embedded development: It is intrinsically much easier to move from one real-time operating system to another, making mission-critical developers much less dependent on, and less locked into, a particular RTOS supplier.

Even with Java's high level of portability, there remain behavioral characteristics that are not consistent across platforms. Most notably, the Java standard does not specify thread timing behaviors, nor predictability of garbage collection activities. These deficiencies generally are not a concern for desktop and enterprise applications but are a major hindrance to the spread of Java to real-time and embedded applications where timing and predictable runtime behavior are critical.

The C++ Portability Guide

(www.mozilla.org/hacking/portable-cpp.html)

1. Be *very* careful when writing C++ templates.
2. Don't use static constructors.
3. Don't use exceptions.
4. Don't use runtime type Information.
5. Don't use C++ standard library features, including iostream
6. Don't use namespace facility.
7. `main()` must be in a C++ file.
8. Use the common denominator between members of a C/C++ compiler family.
9. Don't put C++ comments in C code.
10. Don't put carriage returns in XP code.
11. Put a new line at end-of-file.
12. Don't put extra top-level semicolons in code.
13. C++ filename extension is `.cpp`.
14. Don't mix varargs and inlines.
15. Don't use initializer lists with objects.
16. Always have a default constructor.
17. Be careful with inner (nested) classes.
18. Be careful of variable declarations that require construction or initialization.
19. Make header files compatible with C and C++.
20. Be careful of the scoping of variables declared inside `for()` statements.
21. Declare local initialized aggregates as static.
22. Expect complex inlines to be nonportable.
23. Don't use return statements that have an inline function in the return expression.
24. Be careful with the include depth of files and file size.
25. Use virtual declaration on all subclass virtual member functions.
26. Always declare a copy constructor and assignment operator.
27. Be careful of overloaded methods with like signatures.
28. Type scalar constants to avoid unexpected ambiguities.
29. Always use PRBool for boolean variables in XP code.
30. Use macros for C++ style casts.
31. Don't use mutable.
32. Use nsCOMPtr in XPCOM code.
33. Don't use reserved words as identifiers.

Technologies such as real-time garbage collection and predictable thread semantics, which are embodied in Aonix's PERC Virtual Machine, provide the behavior required for mission-critical applications while remaining fully portable thanks to syntactic compatibility with Java standards. In the realm of hard real-time behaviors, new technologies have been developed that provide precise memory management and hard real-time guarantees suitable for highly constrained and safety-critical applications. These updates to the Java standard are being developed under the umbrella of the Java Community Process [JSR 302 – Safety Critical Java Technology]. This effort is being directed by the Open Group with the cooperation of key companies including Sun, Boeing, Aonix, Rockwell Collins, and Siemens.

Similarly, abstract interfaces for "close to the silicon" activities such as interrupt handling, memory mapping, and device control are being developed based on specifications that are friendly to cross-platform portability.

Java's portability reduces risk, increases productivity

As embedded applications become increasingly complex, placing the burden of source code, library, and behavioral portability on developers is increasingly risky and nonproductive. It also results in software that is more error-prone and more costly to maintain. Although C++, Ada, and Java each have strengths in terms of portability, the capabilities offered by Java technologies are inherently more productive and scalable, and therefore represent a better long-term bet for embedded systems development.

Dave Wood *has been involved with real-time and embedded software for more than 25 years. His activities have included analysis, design, coding, testing, research, and marketing of technology in commercial avionics, telecom, defense, multimedia, software tools, and consumer electronics applications. Dave has worked for Lear Siegler, General Electric, SofTech, ComLogic, and the Software Engineering Institute at Carnegie Mellon University, and presently serves as marketing director for Aonix. He holds a BA from Kalamazoo College.*

Aonix
5930 Cornerstone Court W, #250
San Diego, CA 92121
858-824-0254 • Fax: 858-824-0212
dave.wood@aonix.com
www.aonix.com

Graphics pose portability problem

One of the biggest obstacles to portability for desktop and enterprise applications is at the GUI level. Traditionally, GUIs have been application-specific both in characteristics and in terms of the API. A measure of portability can be achieved by building an abstraction layer that is consistent across platforms. Underneath, the abstracted API binds to the underlying graphics engine, such as the Win32 API for Windows, or a flavor of X11 on UNIX or Linux platforms. Traditionally, the GUI has been considered to be tightly tied to the operating system because that operating systems' users are sensitive to a particular look and feel and expect a consistent functionality across applications on their platform.

Neither C/C++ nor Ada provide a standard GUI API, leaving questions of GUI-specific bindings outside the scope of the language. In contrast, Java incorporates GUI APIs (Swing and AWT) directly in the standard. Applications written to the Swing standard ought to have high portability between conforming platforms through the use of a common API. As mentioned previously, the SWT API, defined as part of the open-source Eclipse libraries, offers another portable graphics interface that is widely used by Java programmers. Whereas the goal of Swing is to offer a consistent cross-platform look and feel, the goal of SWT is to offer cross-platform portability with a look and feel matching that expected on the target platform. Which is better is a matter of opinion, but both offer a means to source code portability.

Increasingly, complex embedded applications require more sophisticated graphics capabilities, yet these applications continue to be demanding in terms of performance and footprint characteristics. Swing solutions, typically mapped to a native GUI such as X11, are usually too big, slow, and unpredictable for use in mission-critical embedded applications. Specialized embedded graphics engines such as PEG+ from Swell Software are a more appropriate fit. SWT bindings to PEG+ exist, offering the high performance needed for multi-platform embedded applications while also accentuating portability.

Safety-critical Java enhances reliability while reducing test and certification costs

By Ole N. Oest, DDC-I

Java provides an excellent environment for developing embedded software. Until now, however, Java has been too big, complex, and unpredictable for safety-critical applications. To address this shortcoming, the Safety-Critical Java Expert Group (JSR 302) (http://jcp.org/en/jsr/detail?id=302) is working on a subset of real-time Java that will make it easier to develop reliable, deterministic code suitable for safety-critical applications requiring FAA certification.

The Java Community first addressed the real-time limitations of Java when it convened the Real-Time for Java Expert Group (RTJEG) in 1999, which developed the Real-Time Specification for Java (RTSJ). This specification, an extension of The Java Language Specification and The Java Virtual Machine Specification, enhances real-time responsiveness by introducing mechanisms for preemptive scheduling and priority inversion avoidance, and providing tools that allow tasks to avoid garbage collection delays.

The Safety-Critical Java Expert Group will further refine the RTSJ, making it suitable for safety-critical applications with the most demanding testing requirements. In particular, the Safety-Critical Java Expert Group will trim the RTSJ spec, ensuring that conforming safety-critical applications can be run without requiring a garbage collector or heap at all, and ensuring that the rigors of FAA certification to DO-178B level A can be met.

As a member of the Safety-Critical Java Expert Group, Phoenix-based tool developer DDC-I, Inc. (www.ddci.com/pr) will be drawing on its pioneering work in developing the Ada 95 specification, which is the "gold standard" for safety-critical programming languages. DDC-I will also bring to bear its extensive FAA DO-178B experience, which is the "gold standard" for certifying safety-critical systems.

Ole N. Oest is the chief operating officer of DDC-I, Inc., a manufacturer of safety-critical software development tools for more than 20 years. Ole can be contacted at ooest@ddci.com.

