Evaluation of Software Tools for the Development
of an Embedded Medical Device

by

Benjamin W. Sanders

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

February 2, 2000

Copyright 2000 Benjamin W. Sanders. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

February 2, 2000

Certified by

Stephen K Burns

Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Evaluation of Software Tools for the Development of an Embedded Medical Device

By

Benjamin W. Sanders

Submitted to the
Department of Electrical Engineering and Computer Science

February 2, 2000

In Partial Fulfillment of the requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Two software tools, PersonalJava and Rational Rose RealTime, are evaluated for the purpose of determining their applicability to the building of a real-time embedded medical device. A testing platform, consisting of a modern embedded microprocessor, input, and display is developed, running a real-time operating system. The tools are evaluated performing simplified versions of tasks they would in an actual product.

Thesis Supervisor: Stephen K. Burns
Title: Senior Research Scientist, Health Sciences and Technology
Table of Contents

1. Introduction ........................................................................................................ 4

2. Background ........................................................................................................ 7
   2.1 VxWorks .................................................................................................... 8
   2.2 UGL .......................................................................................................... 10
   2.3 Java .......................................................................................................... 12
   2.4 Rational Rose RealTime ........................................................................... 15

3. Selection of Hardware Platform ...................................................................... 17

4. Development of Platform ............................................................................... 18
   4.1 Display ..................................................................................................... 18
   4.2 Input ......................................................................................................... 20

5. Java .................................................................................................................. 23
   5.1 How measurements were performed ....................................................... 23
   5.2 Simple AWT tasks .................................................................................. 27
   5.3 Waveform display .................................................................................... 29
   5.4 Problems/Issues ....................................................................................... 30
   5.5 Discussion of hybrid ............................................................................... 32
   5.6 Garbage Collection .................................................................................. 33

6. Rose RealTime .................................................................................................. 34

7. Discussion ......................................................................................................... 36

8. Further work ..................................................................................................... 37
1. Introduction

The development of software for medical devices is a complex process. In addition to the normal complexities of software, they often have hard real-time constraints, and must continue to be easy to use. In the future, it is likely that this software will be even more complex, due to expanded feature sets. Not only will they do more, they will have expanded user interface features and expanded data transfer abilities. However, to remain competitive in the market, they must be developed at the same or faster speed as previous products. To this end, I have singled out two new tools – Java and Rational Rose RealTime – that may aid in the development of said software. The main goal of this project is to determine the viability of these tools for our application – mainly that of an external defibrillator. Secondary goals include identifying various pitfalls in using these tools for our purposes, producing a platform that others may use to further evaluate ideas for future products, and to produce documentation (this document) to illustrate the use of these tools and this platform.

Many places have cited productivity gains from the use of these products. For example, AT&T compared implementing a protocol stack for a wireless product with and without Rational Rose RealTime\(^1\). The group using Rose designed a stack that was the same size, but spent significantly less time and had fewer defects. Furthermore, they only spent 4 days doing maintenance on that code, whereas the non-Rose group spent 4 weeks. There are many reasons why I also feel that these tools would be of benefit. Java would allow for the development of software on a regular PC. The developer could get immediate
feedback on the product and do much of the work before the actual target hardware was even available for use. The developer can make use of the various visual Rapid Application Development (RAD) environments available for Java, which allows him to concentrate more on the look of the user interface and less on the implantation. Java's graphics and windowing capabilities could greatly ease the development of more complicated user interfaces that future products might require.

Rose can also offer great benefits. The operation of a defibrillator is largely state machine based. Because Rose RealTime models its actions through state machines, it use should be well suited. Not only could it increase the speed in which the product is produced, if designed well, many components could be reused in future products. Also, many bugs are introduced into applications because the documentation and the implementation do not match. Since for a large part in Rose, the documentation (i.e., the model) generates the implementation, the use of Rose could decrease the number of bugs.

There are many reasons to question whether or not these tools would work for our applications. For example, the present version of Java is not a real-time product. This is mainly because Java does garbage collection, and it is a non-deterministic process.

---

2 A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred (from http://www.realtime-info.be/encyc/techno/publi/faq/rtfaq.htm#realtime_definition).
3 A process by which allocated memory is automatically reclaimed by Java when the objects using that memory is no longer in use.
4 Both how often it occurs and how long it will take to return.
Rose may add too much overhead to the application. Just as compilers often do not produce as efficient executables as hand-tuned assembly code, Rose may not produce as efficient executables as ones written in C. However, just as compilers allow for efficiency and maintainability gains in writing programs over assembly, so may Rose have an advantage over C.

This being said, there are still many reasons to consider using these. For most of the UI functions, Java's non-real-time properties might be sufficient. For example, being a split second late changing the value of a parameter that only changes every minute or so (non-invasive blood pressure, for example) is probably acceptable. However, latency in the waveform display is probably not acceptable. If there were problems with the truly real-time critical aspects of the UI, they might be handled through some native method. If this could be done without much hassle or loss of productivity, it could be a viable option. With Rose, there are many reasons for using it (such as development efficiency and increased maintainability), but the question of its efficiency prompts us to test it.

The tools will be evaluated on a platform that is reasonable for our future products. Various benchmarks will be run to determine the feasibility of doing different tasks (such as displaying graphics for real-time applications), and sample applications will be written to determine issues related in the use of these tools for our applications.

This document will first introduce background information necessary for the understanding of the project. It will then describe the hardware platform used for evaluation, both in terms of its selection and its development. Then it will
describe the methods used to test Java and the results, and do the same for Rose Realtime. It will conclude with a discussion of the work, and present future possibilities of research.

2. Background

The tools and applications will run on an embedded computer (not a PC), using the VxWorks real-time operating system. The relationship of these is illustrated in figure 1.

![Diagram of the relationship of the various tools discussed](image)

**Figure 1** – The relationship of the various tools discussed. Higher boxes indicate they are implemented in terms of the boxes below them. Note the line indicates communication between the RoseRT model and the UI.

1 User Interface.
2.1 VxWorks

VxWorks is a real-time operating system (RTOS), made by Wind River Systems. An operating system can be defined as ‘A program or a collection of programs, utilities, and APIs that support the execution of other programs, offering resource management, scheduling, and communications services to those programs.’\(^1\) The POSIX Standard 1003.1 defines ‘real-time’ in operating system as ‘the ability of the operating system to provide a required level of service in a bounded response time’\(^2\).

VxWorks is the RTOS associated with Wind River's Tornado Development Environment.

---

\(^1\) Online at http://www.lynx.com/glossary/index.html
\(^2\) Online at http://www.realtime-info.be/encyc techno/publi/faq/rtfaq.htm#realtime_definition
Environment. This environment provides for editing/managing/compiling code, communicating with the target, and running/debugging/monitoring VxWorks applications. The association is demonstrated in diagram 2.

A typical development scenario:

- A board Support Package (BSP) is either written, in the case of custom hardware, or obtained from a vendor if available for "off the shelf" hardware. The BSP is the hardware dependent layer of VxWorks, and provides hardware specific information to the OS (such as memory and IO maps).
- A VxWorks image is made from Tornado’s facilities, adding or removing whatever parts are needed. A boot loader is made (or obtained) for the hardware and is burned into the target’s ROM.
- The operating system is loaded onto the target. This is typically accomplished using a network connection – the host PC will run an FTP server, and the boot loader will connect to that and download the OS image.
- The Target Server is run on the host PC. It establishes communication via a network connection with the Target Agent running on the target hardware.
- Various tools may be attached to the target server from within Tornado to allow for controlling or monitoring the target:
  - Tornado can dynamically download compiled code;
  - The interactive Shell provides a command line from which VxWorks commands and applications can be run;

1 A boot loader is a small program that loads the OS into memory from some source
2 File Transfer Protocol. An FTP server allows files to be downloaded from a host machine via a network connection.
• The debugger allows for debugging;

• The Browser allows for monitoring of various parameters, including free memory, tasks, etc.;

• An optional product WindView allows visualization of system events, such as task switching, memory access, and semaphore activity.

• Application code is written and compiled from within Tornado, and can be dynamically loaded onto the target for testing.

• Once development is done, a VxWorks image can be linked with the application code and burned onto the target's ROM for production.

2.2 UGL

The Universal Graphics Library (UGL) is a graphics stack written to support graphics applications running on embedded systems. It provides a framework to develop standardized, custom device drivers. Also, UGL provides facilities to manage graphics hardware, to process rendering requests and events from input devices. UGL also transparently manages the memory resources needed to perform the requested graphics operations.

UGL consists of four main parts: 2-D code, drivers (output and input), fonts, and OS bindings. The 2-D code provides rendering services to applications and sits above the UGL drivers. The UGL drivers manage the underlying graphics hardware, the output drivers manage the drawing operations,

---

1 There are many methods the target and host can communicate, including serial connections and NetROM, but Ethernet is common and was used here.

and the input drivers manage input events generated by the hardware and channels them to the application software for processing.

Writing drivers for UGL consists mainly of defining a set of functions that the 2-D code can use. These functions have hardware-specific knowledge of the target and the video system. The output driver sets up the necessary system resources and establishes various drawing primitives. The input driver sets up the input system. It also defines functions that are called upon an input event and whose job it is to process that event and place it into a queue for the application. Many drivers already exist for UGL, so creating them is sometimes unnecessary or a matter of modifying an existing driver.
It should be noted that UGL is not a windowing or widget library, and so by itself is not an ideal UI solution. However, several tools exist that for this purpose that sit on top of UGL. Solutions provided by Wind River include Personal Jworks (see below), Zinc (a C++ library for windowing and UI), HTMLWorks (an HTML rendering program), and eNavigator (an embedded web browser). The later three products are not discussed in this paper.

2.3 Java

Java is an object oriented programming language, developed by Sun Microsystems, similar in syntax to C. It is interesting because Java compilers compile the code into bytecode. Instead of running natively on a physical processor, the bytecode is the machine language for the Java Virtual Machine (JVM). The JVM is an execution environment that is available for many different processors and hardware platforms. This arrangement allows for the same Java program to run on many different platforms without porting or recompiling.

Java has many advantages over development in other languages. It has a rich set of libraries that provide for user interface, multi-threading, and file and network I/O, all in a platform neutral manner. It also automatically handles memory allocation and deallocation and avoids the use of memory pointers, both of which are a source of many bugs for programs developed in C.

Personal JWorks is Wind River’s implementation of PersonalJava. PersonalJava 3.0 is a subset of the Java Development Kit (JDK) 1.1 API that is

---

1 A combination of a graphic symbol and some program code to perform a specific function. E.G. a scroll-bar or button. Windowing systems usually provide widget libraries containing commonly used widgets drawn in a certain style and with consistent behavior. From http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=widget&action=Search
intended for embedded applications\(^2\) with network and graphics capability. Specifically, PersonalJava has a smaller memory footprint and is more scalable.

Like JDK 1.1, PersonalJava provides graphics through the Abstract Windowing Toolkit (AWT). The AWT is a set of classes for writing graphical user interface (GUI) programs. These classes include windows, graphics, input areas, text, pictures, choice boxes, buttons, and more. It also includes classes for handling input events, and provides a mechanism for handling the redrawing of objects and other windowing tasks\(^3\).

Important for this project are two Java concepts: Java Native Interfaces (JNI) and JavaBeans. JNI is a mechanism by which methods can be implemented using native code. This is important for several reasons. First, Java cannot access certain hardware resources any other way. For example, a Java program cannot write to a specific location in memory which is important for memory mapped I/O or memory mapped registers. To accomplish this, a Native method can be written in Java, and through using the Javah\(^4\) tool, a header file for a C program is produced. If a C program is then written to accomplish the task using that header and compiled and packaged, Java knows how to call that C function. Then the Java method can be used in the Java program just like any other method.

---

1 Sun's slogan for this is 'write once, run anywhere,' or WORA.
2 Sun has developed other Java products for other segments of the embedded market, including Java 2 Platform, Micro edition, K Virtual Machine, EmbeddedJava, Java Card, Java Phone, and Java TV.
3 Java applications now often use Swing components, which are a more complete set of graphics components that are part of the Java Foundation Classes (JFC). PersonalJava does not at this time support Swing components.
4 A program that produces a C header file from Java code.
JavaBeans is a portable, platform-independent component model written in the Java programming language\(^1\). It allows for visual manipulation of components and increases the ease of reuse. For example, a "waveform Bean" could be written to display a waveform on the screen. The UI designer does not have to deal with the implementation of the Bean (as that is hidden), and can use a visual rapid application development environment to place the Bean on the user interface where he wishes. Though the implementation is hidden, the behavior and appearance of the Bean could be changed from within the app builder. The Bean could be placed in the palate of available components, and could easily be used in future projects as well.

---

\(^1\) JavaBeans FAQ: General questions, online at http://java.sun.com/beans/faq/faq.general.html#Q1
2.4 Rational Rose RealTime

Rose RealTime is a tool for visually constructing real time systems. The structure of the system is captured using the Universal Modeling Language (UML). UML provides a common notation for describing software systems. The bulk of the modeling consists of capsules (concurrent logical objects) communicating via defined protocols through ports (see figure 4). Behavior for

![Figure 4 - A Rose RealTime structure diagram. TimeBaseR1 and FlipFlopR1 are capsules.](image)

...
There is much more to UML, including use diagrams, object diagrams, collaboration diagrams, sequence diagrams.

After modeling the real-time system, Rose RealTime can compile the model into a high-level programming language (C++), which then is compiled into native code, allowing the actual model to be the system. Rose RealTime also has tools that allow components to be run and observed on a processor. The execution may be paused and started, and the states may be observed. Probes may be placed on ports to observe the messages being passed through them, and sequence diagrams can be generated from the resulting list.

Figure 5 - A Rose RealTime state diagram. S1 and S2 are the states of this capsule.
3. Selection of Hardware Platform

The platform used for this project must be a "reasonable" platform base for future products\(^1\). It should be based around a state of the art embedded processor, one that is towards the beginning of its production cycle, so that it would be around for a maximum amount of time. This processor must also have an upgradability path, so that a faster or more featured processor could be used if requirements for the processor increase. The platform must have onboard flash and ram, an LCD controller, LAN, serial IO. A PCMCIA port, IrDA port or other peripheral support would be helpful as well for future investigation.

The processors considered for this task included the PowerPC, the StrongARM, the SuperH, and the x86. The selection was narrowed to these because Java requires 50+ MHz, 32 bit processor. A brief survey of the various offerings seemed to indicate that these were the main contenders. The processor chosen was the PowerPC 823. Its integrated LCD a plus, and the fact that WindRiver had pre-made drivers for the graphics was also a plus. Furthermore, there was internal interest in the processor, and there seemed to be a good selection of evaluation boards from which to choose.

The evaluation board selected was the MDP823PRO, made by EST. They are a local company (good for getting support), and their board had all the required functionality and more. Furthermore, the board had its own BSP for VxWorks, eliminating the need to write additional software. Other benefits were

---

\(^1\) This does not mean one that will be used, simply one that could be used.
that the board was a reference design\(^1\), and the board did not have a long lead
time which is very important on projects with a limited time budget.

4. Development of Platform

4.1 Display

An NEC TFT Color LCD Module (NL6448AC33-24) was chosen for the display, as it was readily available for use in this project and because of its large size would allow for greater flexibility of UI exploration for subsequent projects. This is a 26 cm TFT (thin film transistor) active matrix color liquid crystal display (LCD), a driving circuit and a backlight. It is capable of displaying in VGA resolution (640X480 pixels) and at 6-bit per RGB channel (i.e. 262,144 colors). The LCD controller on the PowerPC 823 is such that changing from one display to another is a simple task, and with proper software abstraction (like in the UGL graphics library), requires no change in application code.

A board was made to provide for the conversion between the pin out of the MDP823Pro board and the input to the LCD module. One issue is that while the LCD module supports 6 bits per channel, the PowerPC 823 only supports 4, and furthermore, the MDP823Pro only brings out 3 bits per channel on the LCD connector. The 823 does not have a 3 bit per channel mode, so in actuality, it is running in 4 bit mode. If the color is being incremented by only one bit at a time, the difference would not be noticed, as the LSB is tied low. But this was considered acceptable for the purposes of this project.

\(^1\) we could freely use any part of the design of the board in our own designs
The graphics components options were included in the Tornado VxWorks facility, the m821 display driver was included, and the parameters for the configuration were set. This VxWorks image was then loaded onto the target with a sample UGL program. However, the LCD did not work. It was found that the BSP did not set the Port D pins (which control the LCD) on the 823 to the 'output"setting, and this was changed in estpro823/syslib.c. Signals were then present on the LCD bus, but the image was garbled. It was eventually determined that the BSP did not correctly define the operating frequency of the chip, and that timing difference was the cause of the LCD malfunction. This was changed in estpro823/target.h, and then the LCD worked.

As an initial test, the included program "uglDemo.c" was run (modified to account for the lack of input at this point). The demo program shows the various capabilities of UGL by displaying screens of the different primates in random colors, sizes, and orientations. This loop is very fast. It was noticed that for high bandwidth operations (for example, displaying the ellipses at high speed), the screen would flicker. Upon investigation of the graphics related registers, it was noticed that there was an underrun condition. At high bandwidths, the video memory was not being filled fast enough. It was suggested by an engineer at EST that this could be caused by the LCD's DMA priority being too low. He suggested that changing the LAM and LAID fields in the SDCR register might solve this problem. It was also suggested that a PowerPC 823e would possibly

1 Direct memory access. This allows transfer between IO devices and memory without the need of the CPU. The DMA's priority level determines in what order different IO devices requesting DMA are serviced.
2 A register controlling the DMA priority of the LCD.
not have this problem, as its larger memory cache would ease the memory bottleneck causing this condition.

4.2 Input

To test the various aspects of the tools (especially Java), some form of input device was required. Several choices were considered: touch screen, mouse, keyboard, soft keys (physical buttons), and socket/shell input. Touch screen would have been advantageous for later exploring UI issues, as in addition to being able to evaluate touch screens, you could easily emulate a mouse or soft key input. However, a touch screen was not readily available for the project. Soft keys are physical buttons that correspond to options on the screen\(^1\). These are popular in many medical products, but are limiting for UI exploration, and the same functions can be done using a keyboard or mouse. Also, hardware resources were constrained.

A factor in deciding the mode of input to be used was the UGL library. It had many pre-made drivers for various kinds of input. For example, it had drivers for keyboard, mouse, and touch screens. However, all of these would require modification to work with the target hardware. Specifically, the keyboard and mouse drivers were for PC-like hardware, and made use of a UART that the target hardware did not have. Of the two, it proved to be far easier to interface a mouse to the target hardware and modify its UGL driver than to do the same for the keyboard.

\(^1\) I.e., a physical button that has a label on the screen above it corresponding to what that button will do at that time.
A final option was to insert events onto the UGL input message queue through some method over the network. One method (that was actually used while later debugging a mouse driver) was to have a program that could be run from the shell prompt that inserted a message into the queue, thereby avoiding the necessity of writing a driver. This proved to be too cumbersome. Another option was to write a driver that listened to a socket, and have a corresponding program on the host computer that could send a variety of messages. However, this proved to be very complicated to write.

It was eventually decided to use a serial mouse for input. There were two options for writing the mouse driver. One involved dealing directly with the mouse. The other involved using UGL's serial services, which take care of the lower level tasks and deliver the input as a stream. This would take much less time to implement, and would be more portable across hardware, so this approach was chosen.

Once the driver was written, the mouse would not work, and occasionally the target hardware would reboot. To debug this problem, a program was written for the PC that sent out known, valid mouse packets, and the driver was modified to print these packets as they arrived. Not only were they not the same, they would not always be printed. This indicated a problem with the serial port itself. A program was written to output characters over the serial port of the target, and the results were viewed on a protocol analyzer. It became apparent that the probable cause was that the settings of the serial port were wrong. Many settings of the protocol analyzer were tried, but none worked. Finally, the output
was analyzed using a digital storage oscilloscope, and it was determined that the
serial port was operating at 5000 baud. It is believed that this is because the
BSP was originally written for slower processor. The register controlling baud
rate (the CD field of BRGC1) was adjusted so that the oscilloscope measured
1200 baud.

The serial port then correctly read data, but the target would still reboot.
The number of packets sent before a reboot varied. Again using the PC program
that sent mouse packets (this time sending random valid packets), it was noticed
that it rebooted upon receiving f7 83 b7 (hexadecimal). It was realized that ‘83’
in binary was 10000011, and bit 11 was the ‘control c’ character, which is often
a reset character. Upon investigation, it was noticed that the serial port was set
up to reboot the system upon receiving a ‘control c’. The settings of the port
were subsequently changed to correct the problem.

Finally, the mouse did not crash the system, but the pointer would
disappear as soon as the mouse was moved. The driver was made to print what
dx and dy it was giving, and it was always some large negative number. Since
the pointer starts in the upper left-hand corner of the screen, that input would
make it move off the screen where it could not be seen. The reason for this was
that the code that was modified was written assuming seven bits per packet.
Currently, there were 8 bits in a packet, and the upper bit was being set to one.
The top bit was masked in the code, and the mouse worked.

The mouse, however, did not perform quite as well as hoped. If the
mouse was moved quickly and then stopped, the pointer would track it slower
than the original movement and continue moving after the movement had stopped. (This was on the order of a quarter of a second lag, but was still noticeable.) Two possibilities seem likely. One, the UGL input queue could not have been consumed fast enough, so that it is still being processed after the movement ended. The other possibility was that since the driver was written using the UGL serial services, it could have its own queue that was not being consumed fast enough. This would be the case if the serial service were slow about calling the driver's event processing function, such that the serial services queue was filling faster than the driver was processing it. Later tests using UGL only display showed that the mouse driver was not at fault. This means that the fault lie with Java – it was simply processing the input too slowly.

5. Java

To get an idea of the performance of the graphics in Java, it was determined that measurements should be done of how long it takes the AWT to perform different tasks.

5.1 How measurements were performed

There were many different ways the measurements could have been performed. Having the code print the time before and after a call might have been sufficient. However, it was feared that some of the measurements would be so short that the overhead of the time and printing operations would be quite noticeable. This could have been overcome by performing the action to be measured many times, timing that, and then dividing that time by the number of
times performed. However, the Java Virtual Machine schedules many things to happen, and it would be hard to tell if other actions (such as garbage collection) had happened in that time and skewed the results.

If it were all in C/C++, CodeTest could have been used. This tool pre-formats the code with tags, and then monitors the bus for these tags. For simply measuring function call times, the overhead is minimal. But the tool as of yet does not have this same functionality for Java code.

It was determined that a good choice here would be to use a method similar to that used by CodeTest, but simplified. A value would be written to a specific memory location before and after the event that is to be measured. A logic analyzer would monitor the bus, and when something was written to the specified memory location, the time and value would be recorded. This was deemed as an acceptable method, as the overhead required to write to a memory location would be very low, and this would provide an extremely fine grain measurement.

There were several issues with this method. Since the Java Virtual Machine takes care of all the memory management, it was not possible to directly write to a specific memory location. This was overcome by the use of the Java Native Interface (JNI). A Java class (declared Native) was written, which had a method that took the value to be recorded as an argument. A C program was written that assigned the argument to a static variable. The memory location of the static variable could be found on the target by using the VxWorks command “Ikup”.

24
Another issue is that of the memory cache. If values were simply being written to a memory location, they would often be written only to the processors memory cache (not to main memory), and the value would never been seen on the bus. The PPC823 was set up to have a Write Back caching scheme, meaning that the value was only written to main memory when the storage location needed to be removed from the cache, due to the processors least recently used (LRU) replacement scheme. The processor could be set up to have a Write Through caching scheme, where the value was written to both the cache and main memory, and the cache could even be turned off. However, since the processor would not normally be used this way, it was determined that a different solution should be found. Most of the IO on the processor is accomplished through memory mapped IO, and the memory management unit (MMU) has facilities to allow for particular memory regions to be non-cacheable. However, it was not clear how to do this, and it appeared that it would take more time to learn the MMU than was acceptable.

Another solution would have been to have the native method force the processor to flush the memory location in question from the cache. The PPC823 has instructions to do this, but not to directly flush an address (translation is required). Again, it was deemed unacceptable to spend time learning this section of the processor, as it was not directly applicable to the larger project. (It was later discovered that the OS provides functionality to perform the cache flush, but it was not discovered until the measurements were all ready under way.)
It was eventually found that declaring the variable Volatile caused it to be written out to memory. However, if the same value was written to the same memory location twice in a row, the second write would not appear on the bus. This was a problem initially because the measurements were being delimited by the same value before and after. Once discovering this issue, the measurements were delimited by different values (for example, a 2" might mean the point before a particular method call, and a 3" might mean the point after it).

Initially, the logic analyzer was set to simply look for the address, and it never appeared. Once the cache issue was resolved, the address would appear too many times. It was realized that the memory control lines had been neglected. The logic analyzer was then set up to observe the Wr, ~ta, and ~burst lines of the 8236 bus, and to only trigger when they were the correct value.

When a run was completed, the results were exported as a tab delimited file to the floppy disk. Those results were then imported into Microsoft Excel for analysis.
5.2 Simple AWT tasks

In order to get an idea of the display performance, a Java program was written to benchmark several of the AWT primitives. This program consisted of a Frame whose paint method was overridden to draw a variety of Graphics\(^1\). These methods included

ClearRect, drawLine, drawRect, draw3dRect, drawOval, drawString

A call to the TimeStamp.Stamp()\(^2\) method was placed before and after each of the drawing calls, with a unique integer passed as an argument. This integer allows for the identification of which call to Stamp is which when looking at the results on the logic analyzer.

The main() method was overridden to make a series of calls to repaint(), sleeping between calls\(^1\). The sleeping was a precaution, as it was not known how much resources the painting would require (doing too much painting might have starved other threads of lower priority, which was not desirable).

The times required for different Graphics tasks are useful, as they give an idea of how long painting the screen will take. But they do not give as much of an idea of how much overhead Java adds to the process. For this, a program was written in C using UGL that draws the same thing as the Java program. As before, Stamp() (or rather, the C function to which the JNI call was linked) was

\(^{1}\) A Frame is a type of Component that is a window. Graphics are a class whose methods perform various drawing operations on the screen (onto a graphics context).

\(^{2}\) This is the JNI method that was written for purposes of benchmarking. It takes an integer and writes it to a specific memory location.
called before and after the graphic function calls. This way, the analogous Java and UGL functions could be compared.

Each painting routine was run 10 times, and the times for each function were averaged.

<table>
<thead>
<tr>
<th>AWT task</th>
<th>Time (ms)</th>
<th>Time (ms)</th>
<th>UGL task</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClearRect</td>
<td>41.59414</td>
<td>21.37787</td>
<td>uglRectangle</td>
<td></td>
</tr>
<tr>
<td>SetColor</td>
<td>0.792183</td>
<td>8.489269</td>
<td>uglLine, 2 uglSetColor</td>
<td>To short to measure</td>
</tr>
<tr>
<td>DrawLine</td>
<td>7.670095</td>
<td>2.768269</td>
<td>uglSetColor</td>
<td></td>
</tr>
<tr>
<td>SetColor</td>
<td>0.980344</td>
<td>To short to measure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DrawLine</td>
<td>6.748718</td>
<td>To short to measure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DrawRect</td>
<td>7.445056</td>
<td>20.9995</td>
<td>uglRectangle</td>
<td></td>
</tr>
<tr>
<td>draw3dRect</td>
<td>32.63458</td>
<td>No UGL equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DrawOval</td>
<td>8.27791</td>
<td>3.321547</td>
<td>uglEllipse</td>
<td></td>
</tr>
<tr>
<td>FillOval</td>
<td>11.32952</td>
<td>3.346623</td>
<td>UglEllipse</td>
<td></td>
</tr>
<tr>
<td>DrawString</td>
<td>40.96913</td>
<td>27.29249</td>
<td>UglText</td>
<td>Measured separately</td>
</tr>
</tbody>
</table>

Table 1 - Comparison of times to execute certain AWT and UGL tasks

A problem arose in the UGL benchmarking program in that not all the values were getting written to memory. This happened when either two calls to the Stamp() function were either consecutive or were separated by another function that took very little time to do (such as uglLine or uglSetBackgroundColor).

In an attempt to get all the Stamp() functions to write out to memory, a taskDelay() call was added to the UGL program between blocks of UGL/Stamp code. This way, the program would wait for a specified amount of time before it moved onto the next section of code. This had the effect that almost all Stamp()

---

1 The repaint() method causes a graphics context to redraw all the graphics on it.
calls were writing out to memory. However, the times for the UGL tasks were now much higher (an order of magnitude in some cases) than they were previously. Since the taskDelay() function lowers the priority of the given thread for the specified amount of time, the OS scheduler was swapping some other task in during this time. Thus, each time it swapped back to the UGL program's task, it could have had to do some set up to resume the UGL tasks, and this set-up cost becomes figured into the results. This is not insignificant, as if there is a lot of task switching in a program, the UGL task times could be higher.

A real time waveform display is one of the tightest constraints on the UI. Since this waveform will mainly be made of lines, the uglLine function was tested further. Ten lines were drawn consecutively, and the time recorded. For a series of trials, the average time required to draw a line was 1.38 ms. However, the times required from trial to trial tended to be either around 1.6 ms or .4 ms. The reason for this is not known at this time.

**5.3 Waveform display**

In order to explore the issues concerning using Java for the display functionality in a medical instrument, a sample waveform display program was written. This program displayed data from a stored waveform.

The program consisted of a class Frame whose main() method called the step method a custom class named WaveCanvas every 30 ms, with the current waveform data as the argument. The WaveCanvas class is a subclass of Canvas, and it is what accomplishes the painting. It was also made into a Bean,

\[1\text{ taskDelay()} \text{ is a VxWorks method which suspends the thread that calls it for a specified amount of time.}\]
as it would probably be done in an actual project. (The main method of the Frame substituted for the ISR handler for now.) There were also three buttons on the frame.

5.4 Problems/Issues

The first problem encountered was that the waveform display had gaps in it where the waveform was not being drawn. The reason for this is that the program was written to draw one segment on each call to the WaveCanvas step function. This function would call redraw(), which instructs the AWT to call update(). However, Java does not guarantee that redraw() will immediately call update. In actuality, if Java is running behind schedule, it will group multiple requests together and only call update once. The program was changed to reflect this behavior (update henceforth kept track of how much of the waveform it had drawn so far). The interesting thing here is that this behavior was not noticed when the program was run on the development computer (a Pentium II 350), as it is fast enough for the JVM to never have to group multiple calls together.

This problem was solved by having a pointer be kept of the how much had been drawn and how much should have been drawn, and having the update method draw the points in between. Also, depending on how fast the hardware would be sampling the waveform, it might package several samples together, which would lesson the number of times per second update would need to be called.
The largest problem with this Java-only waveform display is that there was occasionally a noticeable (greater than 500 ms) pause in the display of the waveform. Upon further investigation, it became apparent that this was due to the Java VM performing garbage collection. The effects of the garbage collection are discussed in more detail later in this document.

An additional problem was that Java responded sluggishly to the mouse input. As previously described, the pointer would lag behind the actual mouse movement, and when a button was pressed, there was a distinct lag (~ 500 ms) before the button would change. This would indicate that a faster processor would be necessary for Java to be used.

To have a basis of comparison, a version of the waveform display was written in straight UGL. This waveform could be run very quickly without any degradation. However, adding pushbuttons and other things done easily in Java would have been cumbersome. It was determined that a good compromise would be to have a Java program that used straight UGL written in C for the real time components. This was accomplished using JNI.

A program that would draw a waveform in a specified box was written in C using UGL. A Native Java class that was a subclass of AWT.Component was made, and this program simply called the UGL program with the arguments being the size of the Component. Then this Component was made into a Bean. In this way, the benefits of the native code could be used within the visual realm of VisualCafé.

---

1 A leading RAD for Java.
First problem with hybrid was that the waveform never appeared. It was determined that this was because it was failing to get the graphics context. The code for the waveform display was getting called before the graphic context was being initialized. Since the object spawns a thread that actually does the work, this thread was made to test to see if the result of uglGcCopy() was null, and if it was to taskDelay() and keep trying until it was no longer null.

Once the graphics context problem was resolved, there was still no waveform - just a dot in the upper left-hand corner of the screen. It was determined that when the Canvas called the native object, it had not yet been resized. Therefore, the native task was called with arguments reflecting a size of zero. This was resolved by having the native thread wait and then get the size properties from the Canvas once it had been resized.

5.5 Discussion of hybrid

The hybrid waveform display worked quite well. Since all of the waveform handling is outside of Java, the garbage collection was not a problem. One issue that did arise was that of redrawing. Moving an object (such as the mouse) over the UGL waveform would destroy that part of the picture. This was because the waveform was not being redrawn (the UGL waveform simply clears and paints what is new in the waveform). To get around this, the Java component needs to override the repaint() method such that it calls a native function to repaint the whole waveform.
5.6 Garbage Collection

When running the first GUI, it was noticed that the waveform display would occasionally pause for a noticeable amount of time. Upon investigation, it was determined that this was due to the Java VM performing garbage collection. Garbage collection is the process by which the Java VM reclaims memory that is no longer being used so that it may be used again. This is a large feature of Java, as having to specifically allocate and deallocate memory is the cause of defects in many C programs. However, in current Java VM implementations, the time required to perform garbage collection is non-deterministic, thus making it difficult to use for some applications having hard real-time requirements. Also, in this particular implementation, the process of garbage collection blocks the AWT from updating the screen.

To determine how much garbage collection affected the display, the program was run on the target using the -verbosegc flag. This flag instructs the Java VM to output a message when garbage collection occurs, describing how much memory was reclaimed and how long garbage collection took.

---

1 This issue is being addressed by several groups drafting standards for a real-time Java.
2 An AWT event will stop the garbage collection process, so no input is lost.
<table>
<thead>
<tr>
<th>Object Heavy</th>
<th>Differences</th>
<th>Times</th>
<th>Object Light</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:54</td>
<td>0:50</td>
<td>0:53</td>
<td>1:44</td>
<td>1:34</td>
</tr>
<tr>
<td>1:44</td>
<td>0:42</td>
<td>2:12</td>
<td>0:38</td>
<td></td>
</tr>
<tr>
<td>2:26</td>
<td>0:24</td>
<td>3:40</td>
<td>1:28</td>
<td></td>
</tr>
<tr>
<td>4:16</td>
<td>1:26</td>
<td>4:34</td>
<td>0:54</td>
<td></td>
</tr>
<tr>
<td>5:20</td>
<td>1:04</td>
<td>6:22</td>
<td>1:48</td>
<td></td>
</tr>
<tr>
<td>6:12</td>
<td>0:52</td>
<td>7:25</td>
<td>1:03</td>
<td></td>
</tr>
<tr>
<td>6:41</td>
<td>0:29</td>
<td>9:30</td>
<td>2:05</td>
<td></td>
</tr>
<tr>
<td>8:23</td>
<td>1:42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:38</td>
<td>1:15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average =</td>
<td>0:58</td>
<td></td>
<td>average =</td>
<td>1:19</td>
</tr>
</tbody>
</table>

Table 2 - Time between garbage collections (Times are in minutes and seconds).

The initial GUI test program created and destroyed many objects. (Java lends itself to this type of programming, and delays due to garbage collection are not visible on the much faster workstation on which the test program was developed.) It was decided to re-write the program, taking pains to create and destroy as few objects as possible.

It took on average of about a second to do the garbage collection, and it did it about once a minute. While the time required to do the collection would decrease with a faster processor, it appears that it would always take a significant fraction of a second, and the waveform pausing for that amount of time is unacceptable. Therefore, a straight Java waveform display would appear to be unacceptable.

6. Rose RealTime

One of the main concerns with Rose is efficiency, specifically of message passing. The events in Rose are triggered by messages, and there was concern
that the overhead introduced by Rose might make this much slower than a similar system implemented using just operating system constructs. To test this, a simple Rose model was made (see diagram). It consisted of two camules, communicating via a port. The timebase capsule sent out a signal at regular intervals out the port. The flipflop capsule had two states, and it changed between these states upon receiving the signal from timebase.

The timebase capsule was made optional, meaning it was incarnated via code at run-time (as opposed to being created at compile-time). This way, it could either be made to run on the same physical thread as flipflop, or a different physical thread. This was of interest because there are certain cases where running capsules in separate physical threads is desirable. For example, if one capsule should have a higher priority than all others, it might be placed on a separate physical thread with a higher priority. It was of interest to see if this would introduce additional overhead.

The time was tested by writing an integer to a volatile memory location as before. A call was placed before the timebase sent the message, and on the transition between the two states of flipflop. To get an idea of how much overhead Rose creates, this system was re-created without using Rose. One thread was created that sent a VxWorks message at the same interval as timebase, and another thread was created that waits for that message, and upon receiving it, switches states.

It was found that, on average it took approximately 15 ms for the VxWorks only version of the system to propagate the message. In contrast, it took
approximately 25 ms for the Rose version of the system to propagate the message. (There was no real difference between the same-thread and separate-thread versions of the Rose model.) The high times for both models are due to the low tick rate set up in the operating system (i.e., how often the PowerPC's decrement counter triggers and interrupt).

7. Discussion

Between its slow performance and the artifacts of its garbage collection, Java would not appear to be a viable option at this time. However, it may be an option in the future. With improvements in the implementation of the Virtual Machine, the performance will increase (reference Sun's HotSpot technology), and faster processors will further improve performance. Further more, there are real-time Java standards being drafted now, and these standards will address issues such as garbage collection.

Besides performance related issues that will hopefully be resolved in the future, Java worked quite well. A Java program created by VisualCafé on a Windows NT machine was downloaded to the PowerPC target running VxWorks, and the program ran the same on both machines. There were a few minor issues. For example, because of the speed difference between the PC and the target, the waveform display problem described before was not evident on the PC. Also, JNI functions cannot be tested (though a stub component can be used), so if the program made extensive use of JNI, some of the gains of using Java would be lost. But in spite of these minor issues, once the performance
issues are dealt with, Java will be a viable option, and will meet the original goal of designing the UI without needing the target hardware.

Rose Realtime appears to be acceptable. The only question that remains is how much it would help our process. If little of our future products can be modeled easily within its state machine framework, it will be of little use. Future system architects will answer this question.

8. Further work

There are several paths down which Java could be pursued further. One is to further test the performance of the present product. The amount of time required for Java to start (from power on to the program running) and the percentage of CPU time devoted to Java are two questions yet to be answered about its present incarnation.1

Another avenue of research would be the various Java to C compilers. These take Java programs and compile them into C code, which is then compiled for the target. They could offer substantial speed benefits over the present Virtual Machine. However, they may still have similar problems with garbage collection. Furthermore, debugging becomes more of an issue, and portability is not as smooth.

Perhaps the most interesting path of further exploration lies in the proposed real-time extensions to Java. Various groups are in the process of designing the specifications for a Java incarnation designed specifically for real-time applications. The design work is not finished, and an implementation is

---

1 It was originally planned to address both of these issues, but it was not done due to various constraints.
probably a few years off. However, at that time, this real-time Java should be examined and considered for this application.

The most interesting path to explore with Rose RealTime would be to actually measure the productivity gains made by using it. That, however, would require extensive work – actually implementing an entire system twice – and require two separate groups to implement it.