2007

Developing real-time systems in Java on Macintosh

Phillip J. McKerrow
University of Wollongong, phillip@uow.edu.au

Publication Details
This conference paper was originally published as McKerrow, PJ, Lu, Q, Zhou, ZQ and Chen, L, Developing real-time systems in Java on Macintosh, in Proceedings of the AUC Conference, Gold Coast, 23-26 September 2007, 11.1-9.
Developing real-time systems in Java on Macintosh

Abstract
In this paper, we examine the advantage of writing real-time applications and operating systems in typesafe languages, such as Java. In this context, we look at the issues of using Java in real-time systems and the development tools available on Macintosh computers. Finally, we explore the potential of the Sun SPOT system: a credit-card sized Java computer with input/output, and a wireless network, that runs the Squawk JVM. It is planned to be released in Australia in the third quarter of 2007.

Disciplines
Physical Sciences and Mathematics

Publication Details
This conference paper was originally published as McKerrow, PJ, Lu, Q, Zhou, ZQ and Chen, L, Developing real-time systems in Java on Macintosh, in Proceedings of the AUC Conference, Gold Coast, 23-26 September 2007, 11.1-9.

This conference paper is available at Research Online: http://ro.uow.edu.au/infopapers/699
Developing real-time systems in Java on Macintosh

Phillip M’Kerrow, Qinghua Lu, Zhi Quan Zhou and Li Chen

School of Computer Science and Software Engineering, University of Wollongong
Wollongong, NSW, 2522. phillip@uow.edu.au

Abstract. In this paper, we examine the advantage of writing real-time applications and operating systems in type-safe languages, such as Java. In this context, we look at the issues of using Java in real-time systems and the development tools available on Macintosh computers. Finally, we explore the potential of the Sun SPOT system: a credit-card sized Java computer with input/output, and a wireless network, that runs the Squawk JVM. It is planned to be released in Australia in the third quarter of 2007.

Keywords. Real-time systems, Java, safe programming language, TINI microcontroller, Sun SPOT, aerial robot

1 Introduction

The eminent release of the iPhone, with predictions that it will become the hand-held device of choice, may result in many more people developing Java software on Macintosh computers. Sun claims that Java is used in 1.2 billion mobile phones, 1.4 billion smart cards, and many other consumer devices (SPOT). An increase in the number of developers will result in improvements in programming tools with spin offs for the development of real-time software. With the release of the Sun SPOT system, the next few years promise to be an exciting time for people who develop real-time systems on in Java on Mac OSX.

From its inception, Sun has promoted Java as a language to program embedded systems to connect them to the internet. One of its major uses is in web applications both in server applications (e.g. WebObjects) and as applets running on web pages. Up till now it has had very little penetration into real-time systems. With the release of the Sun SPOT (Small Programmable Object Technology) in April 2007, Sun claim to have achieved their original goal. A Sun SPOT is a small, Java machine (Fig. 1.) with input/output (i/o) that can be used stand alone or in sensor networks. It communicates with other Sun SPOTs using IEEE802.25.4 wireless links.

Fig. 1. Sun SPOT: battery, processor, i/o and sun roof.

Most real-time applications are developed on a host computer (Macintosh) and downloaded into a target computer (embedded processor). Most target systems don’t have any development tools or human interface. The software required on the host is an integrated development environment (IDE) for code editing, compilation and linking; a converter program to convert the output into a form suitable for downloading and a download program. The embedded
computer must have a download program to load the object code into its memory and to start the program executing. Newer systems support download and booting over a network.

Usually, they are small applications to control actuators and read sensors in real-time. Many consist of a single processor card, an i/o card, a power source and a connection to a host. Traditionally the connection to the host is an RS232C serial link, however, more modern systems connect to a network. Their user interface may be as simple as a few buttons and LEDs. They usually don’t have a keyboard and monitor, so a network connection to a remote desktop is desirable.

We have developed a web site to document out systems and to make our experience available to others (M’Kerrow, 2006). In the following sections, we look at the requirements of embedded software and the suitability of Java. Then we look at the TINI (Tiny Internet Interface) system and how to develop for it on the Macintosh. We finish with the exciting new Sun SPOT (Small Programmable Object Technology) system and its application to autonomous flight.

2 Development of real-time systems

Real-time software has much more stringent requirements than personal computer software (Laplante, 1993). It must execute within strict time deadlines, it must be correct, and it must be robust. Your car has an embedded computer controlling its engine. You expect it to calculate the correct fuel/air mixture every time you press your foot on the accelerator. Also, you expect that computer to run for years without crashing or having to perform software upgrades to fix bugs.

A hard real-time program must guarantee to finish its execution before a time deadline. Embedded systems either have no operating system, i.e. they are a single process, or they have a real-time operating system (RTOS) that uses a deadline scheduler to schedule tasks according to time priorities. A soft real-time program only has to meet its deadline on average.

The design of real-time systems involves the careful separation of tasks based on timing. For example, our ultrasonic sensors scan the environment every 100msec (M’Kerrow and Antoune, 2007). At the end of the scan, the input program reads the echo and places it on a queue. This input program is a hard real-time program because its execution is tied to the timing of the sensor. An analysis program reads the echoes from the queue and performs calculations to perceive objects in the environment. The producer/consumer queue provides a way of communication between the hard real-time capture process and the soft real-time analysis process.

2.1 Testing real-time systems

Debugging and rigorous testing of real-time systems remains a difficult problem. A network connection facilitates the development of remote tools where data collected on the real-time system can be analysed and displayed on the host. Also, the real-time system can receive commands from control programs and interactive user input on the host.

Performance monitoring and debugging take time to execute and consequently they impact the real-time performance of the processes running in the real-time system. A network connection enables a hybrid approach where small, fast probes collect data and put it onto a queue. A soft real-time process takes the data from the queue and outputs it to the host over the network. All the calculation and analysis software runs on the host, moving most of the execution load to the host.

3 Safe Languages

The requirements of real-time programs call for language features that are not found in many programming languages. These are discussed in detail in an accompanying paper (M’Kerrow, et. al., 2007). Low-level features are often removed from languages because they are not safe. That is, when incorrectly used, they can crash other programs or the operating system (OS).
C is the language most commonly used in real-time programming. However, it is not safe (McKerrow, et. al., 2007), and poor programming may result in an RTOS crash. It has week typing, which results in programs with numeric errors. Incorrect use of pointer arithmetic will allow a program to write anywhere and corrupt both code and data. C has no run-time environment so the programmer has to write all the memory management code. Also, he has to write the code to check for overflow and underflow of common data structures such as arrays.

The goal of a safe language is for the compiler to handle potentially unsafe operations rather than the programmer. Also, a safe language includes run-time support to catch and handle run-time errors. The features that make a language safe (McKerrow, et. al., 2007) include:

- A safe language is type safe, so there are no errors due to numbers changing in value when assigned from a variable of one type to a variable of another type.
- There is no pointer arithmetic. Programming with references stops a program scribbling outside a program’s memory area.
- A safe language includes overflow and underflow checking in its run time support, so that buffer writes can’t corrupt code. A common method of attacking the security of an RTOS is to attempt to achieve a buffer overflow or underflow.
- A safe language has real-time garbage collection, i.e. automatic memory management to avoid memory leaks which may cause an RTOS to run out of memory and crash.

4 Real-time operating systems in safe languages

We are developing a real-time operating system (RTOS) in Java. The requirement that a real-time application will run for years, and never fail, with no human intervention places huge demands on the operating system that supports it. Some embedded systems try to avoid this problem by not having an operating system. The only advantage of that approach is that the programmer knows all the code. The disadvantage is that the application programmer has to write all the code.

One advantage of using an operating system is that the programmer is better able to focus on programming the real-time task because many of the low-level details are abstracted away by the operating system. Another advantage is that the task can be decomposed into several interacting processes. As each process is small relative to the task, the complexity of the code is reduced and its correctness increased.

However, the programmer has to be able to rely on the operating system to execute every process reliably and in time. Also, the operating system must provide the low-level services the programmer requires to implement the task. In addition, the increase in programmer productivity and system reliability should far outweigh the increase in execution time due to using an operating system.

One strand of real-time operating system research is looking at the question: can developing an operating system in a safe language result in a system that an errant process can’t crash? This question decomposes into two sub-questions. First, if we write a process in a safe language can we guarantee that the process does not cause harm to other processes or to the operating system, because the compiler has removed all the unsafe statements?

Second, can we develop an operating system that can’t be crashed if we use a safe language? This goal raises a further question: are algorithms commonly used in RTOSs safe? Does writing these algorithms in a safe language make them safe or are there alternate algorithms that are safe because they are written in a safe language?

A number of research projects have looked for answers to these questions. The Burroughs B5000 did not have a memory management unit (MMU) so it relied on the Algol compiler to detect dangerous code (Tannenbaum et. al., 2006). XO-2 (Brega, 2002) is an RTOS developed at ETH in Zurich in Oberon to run on PowerPC embedded processors. Oberon is an object-oriented language developed by Nicholas Wirth to follow on from Modula-2 (Oberon). It is boot loaded from the host Macintosh and communicates to users via web pages running on the host network. XO-2 has been used extensively in the control of robots (Blubotics). The developers claim very fast switching times between processes because the MMU is only needed for address translation and not for catching program errors.
A more recent project is the development of the Singularity operating system by Microsoft Research (Tanenbaum et al., 2006). It is programmed in Sing#, a safe language based on C#. All processes run in a single virtual-address space, which is very efficient because it eliminates kernel traps to perform context switches. The exclusion between processes is complete (without using an MMU for protection) with each process having its own code, data structures, runtime, libraries and garbage collector. Processes communicate by sending strongly typed messages to the operating system over point-to-point bi-directional channels.

We face a number of problems when developing an operating system in a safe, high-level language.

- There are low-level operations that can’t be coded in the high-level language. This usually forces the person programming the operating system to program some operations in an unsafe language. This code is often called “trusted” code because it is locked away inside the operating system so that the application’s programmers can’t access it. To become trusted, it must be rigorously tested. Also, the smaller the amount of trusted code the less chance there should be of it causing problems.

- A system clock is required to implement a deadline scheduler. Typically, this clock will generate a hardware interrupt every n milliseconds. If the language does not include a clock function, this real-time clock has to be written in assembler.

- The clock is one example of an interrupt. When an interrupt occurs, the processor stops the thread of execution of the current process at the end of the current instruction, saves some system state and vectors to an interrupt handling function. Can a function be written in the chosen high-level language that is not called by software, and is there a facility to store its address in the memory location from where the hardware fetches the vector?

- When the interrupt handler completes servicing the interrupt it normally returns to the hardware, which restores the state and continues the thread of execution of the current process. This requires the ability to write a method that does not return to calling software but via the hardware to the interrupted process. An interrupt handler function finishes with a return from interrupt instruction not a return from subroutine instruction.

- In order to implement some operations in response to interrupts (for example a time out), interrupt handlers may have to change the return address of the process that it interrupts so that the operating system can take the processor away from that process.

- All libraries used by the operating system and the applications must also be written in the safe language and compiled with the operating system or the application. As C is an unsafe language, standard C libraries can not be used unless they are guaranteed to be trusted.

- Most modern embedded systems are connected to networks. Many distribute processing to multiple processes over the network, for example sensor networks. When two processes communicate by passing a message, the receiving process often waits for the sending process. When they are running on a single processor, the wait time is determined by the load on that processor and deadlines can be guaranteed to be met. When they are running on a separate processor it is much more difficult (an in many designs impossible) to guarantee that deadlines are met. We are still thinking about how to solve this one.

5.1 Real-time design issues

Real-time systems have to guarantee that processes meet time deadlines. Anything that causes indeterminism in the execution time makes it harder to achieve that guarantee. Interrupts are one cause of indeterminism because they cause the processor to stop what it is doing and service the hardware. They take the processor away from the running process. For this reason interrupts should always return to the hardware, so that they are transparent to the interrupted process.

To keep the number of interrupts to a minimum, polling of input/output is preferred to interrupts. However, the hardware designer may have reduced the amount of hardware by assuming that the software would respond to interrupts and the data request/available signal disappears too quickly to be detected by polling. Design of real-time systems involves a trade off between what is done in hardware and what is done in software. A poor decision by the hardware designer can result in the software taking much longer to execute than it would with a better design. A better hardware design for real-time systems is a handshake design where the data available signal is not reset until the data is read and the data request signal stays valid until the data is written.

McKerrow 11-4
Priority pre-emptive scheduling is another cause of indeterminism. An interrupt can result in the scheduler transferring control of the processor from the current process to other processes for an undetermined period of time. For this reason, many real-time systems use cooperative scheduling where the current process only gives up the processor when it is finished. However, to guarantee that deadlines are met the application must be designed as a number of small, fast, interacting processes. For example, instead of a single process having a loop whose execution time is determined by data values, the loop is divided into two processes that start one another. Each time a process returns to the scheduler, other processes get a chance to run, where with a single process it may hog the processor.

Executing multiple processes to perform a single task requires those processes to share data. Methods of doing this are common-data storage, message passing, and producer-consumer queues. One instance of a common data object has to be accessed by all processes. Access to attributes in the common data object must be done via methods that enforce a protection protocol. Only one process should be able to write to an attribute of the common data object. By not allowing pre-emption, this can be enforced for processes with out critical sections, because a write can’t be pre-empted by a higher-priority read.

However, even if we had critical sections, interrupt handlers will ignore them so care has to be exercised when they access common data to ensure that, at worst, access to a variable in common data can only result in a delay and not data corruption. Another reason is to keep the number of interrupts to a minimum. A producer and a consumer that share a queue write to and read from different places, so the methods for this class can be written to avoid data corruption.

When message passing is used, the process wanting to read the message may have to wait for another process to send it. So the process has to set its state to wait and return to the scheduler. When the message is sent, the sending process has to enable the receiving process to be restarted by the scheduler. Programmers have to avoid creating deadlocks where processes are waiting on each other.

Talking of hogging the processor, a cooperative scheduling system must be able to timeout processes that are taking too long. To implement a timeout, the real-time clock interrupt has to set a timeout flag to tell the scheduler to run a timeout process, and change the address that it will return to in the hogging process. The new return address is to an instruction in the hogging process that exits to the scheduler. In this way the hogging process will exit normally, the scheduler will continue to schedule tasks, and a timeout process will be run to report on the timeout error. Such hogging indicates either a software design fault or a hardware failure. Either requires human intervention to investigate and fix the problem.

Hardware failures also cause interrupts. As we commented before a mathematical error should cause an exception. The use of a safe language should guarantee that illegal instruction and invalid memory address errors do not occur due to data being written over code. Missed interrupts are usually the result of the software taking too long to service the interrupt and require a hardware or software redesign. Segmentation errors are only of concern when an embedded system uses virtual memory (unusual), and should be handled by the exception facility in the language.

6 Java

Java was designed to be a safe language and meets the criteria in Section 3.3. Having looked at the design issues of RTOSs in the last section, here we will look first at additional issues with Java and then examine how these issues are handled in two embedded systems programmed in Java.

Java is designed to compile a program every time it is run. Much work has gone into just-in-time compilers to compile the byte codes on the target machine so that performance is not reduced. This approach makes sense in mobile phones and in applets on the web were the code is often downloaded and runs only once. However, embedded systems are generally compiled once and run many times. This difference in underlying philosophy means that Java compilers normally are not optimised for producing code for real-time systems.

Much of the magic of Java is due to threads. Programmers can produce small applets simply by overriding 4 routines, because the run-time event loop does most of the work for them. However, all Java programmers have tried to find the size of a window in an instruction sequentially after the instruction to open the window, only to get a size of zero returned. The reason, it appears, is that Java started a separate thread to open the window and continued executing the constructor thread. We are still looking for documentation on when the Java run-time starts additional threads and why.
As Lee (2006) points out in an article on concurrent programming threads result in non-determinism. Worse, a programmer appears to have no way of knowing when this non-determinism is going to occur. So it may not be possible to guarantee that a hard real-time process will meet a deadline, because we don’t know when and how the language schedules threads. Another issue is that threads must be coordinated with locks when trying to access shared data (Ousterhout, 1995). Forget a lock and the result is corrupted data. Writing data access synchronisation is difficult, because circular dependencies must be avoided. While some developers wish that Java didn’t have threads, others are trying to improve the Java threading model (Wellings, 2004) through the development of the Real-Time Specification for Java (RTSJ). This approach forces a specific concurrency model on the design of the real-time system. Another addition that is required is a real-time clock class to Java.

7 TINI

To conduct our research into developing a real-time operating system in a safe language, we started with a TINI (Tiny Internet Interface). It is a microcontroller that runs a Java virtual machine (Fig. 2.). The TINI platform is a combination of the broad-based i/o, a full TCP/IP stack, and an extensible Java runtime environment that simplifies development of the network connected equipment. We develop the programs in XCode on a Macintosh and then download them into the TINI using the Slush command shell that runs on it. Documentation on how to set up a system to develop applications for TINI is available on our web site (M’Kerrow, 2006).

![Maxim TINI Java micro-controller from Dallas Semiconductor](image)

**Fig. 2.** Maxim TINI Java micro-controller from Dallas Semiconductor

![Software architecture of TINI micro-controller](image)

**Fig. 3.** Software architecture of TINI micro-controller

McKerrow 11-6
When you try to code your operating system or application design to run on a Java microcontroller, you are confronted with another issue. This issue can be summarised by the question: “Is the microcontroller already running an operating system?” If it is not, then how are the threads scheduled, how is the JVM executed to compile the code, and how is the run-time support achieved?

In the case of the TINI the Java program is downloaded using commands in Slush, not by a boot loader. The documentation claims that Slush is only a command shell and that it is a Java application running on TINI. As shown in Fig. 3., the JVM is running under an operating system called the TINI OS. So, whether the programmer likes it or not he has to use the TINI OS.

TINI OS is at the lowest level of TINI runtime environment. It consists of Process and Thread Schedulers, I/O Subsystem and Memory Subsystem. A micro-controller timer is used to update a real-time clock every millisecond. The thread scheduler runs every 2 msec. A round robin scheduler divides time between processes in 8 msec slices. Round robin scheduling makes it is very difficult to guarantee that any process running on TINI can meet a real-time deadline.

The JVM sits on top of TINI OS. In between there is a native interface layer, so TINI OS is probably not written in Java. We can invoke assembly code functions to solve low-level problems from Java applications using this native layer. The i/o library uses the native interface layer to call functions written in assembler to read inputs and write outputs.

Applications programs written in Java application sit on top of JVM. The JVM supports the Java API and libraries. In our research we are testing RTOS by running it as a single application on top of the JVM. We can also run it as an application in Mac OSX.

8 Sun SPOT

With the release of the Sun SPOT (Small Programmable Object Technology) in April 2007, Sun claim to have achieved their goal of Java being the language of choice for small real-time computers embedded into sensors, robots, instruments, machines and consumer devices. A Sun SPOT is a small, Java machine with i/o that can be used stand alone or in sensor networks. It communicates with other Sun SPOTs using IEEE802.25.4 wireless links.

![Sun SPOT Diagram](image)

**Fig. 4. Sun SPOT.** Left: suite creator that runs on host, Right: architecture of embedded device

Sun is tackling the issues of using Java to program embedded systems with the Squawk virtual machine (VM) (Simon, 2006). It is a small JVM with a split architecture (Fig. 4.). On the host machine the Java byte code is transformed into
a more compact execution format and packaged in a suite file for downloading. The VM on the SPOT interprets the suite file. To overcome the problem that Java is interpreted not compiled, parts of the onboard VM and run time (e.g. the garbage collector) are translated from Java to C and thence to machine code, improving performance and removing the need for just-in-time compilation.

Applications are represented as objects that are instances of the Isolate class to isolate them from one another. Sun claims that the SPOT has no operating system, but that operating system functionality is built into Squawk. It implements green threads which emulate a multi-threaded environment without relying on an underlying operating system. Green threads implement cooperative multitasking. When waiting on something a thread is blocked on an event queue that is polled by the scheduler.

Interrupts are handled by assembler routines that set bits in an interrupt status word. The scheduler checks the interrupt status word and resumes the thread for the device driver for that interrupt. Thus, many of the features required for real-time programming appear to be available in Squawk. Only when we implement a hard real-time application, such as the control of a UAV (Section 7) will we be able to confirm this.

![Radio controlled helicopters](image)

**Fig. 5.** Radio controlled helicopters. Left: Draganflyer 4-rotor helicopter with IMU and bluetooth serial link to host, Right: Lama dual rotor-helicopter with IMU.

9 Unmanned aerial vehicles

The Draganflyer (Fig 5.) is a four–rotor, radio-controlled helicopter (M’Kerrow, 2007). We have converted it to an unmanned aerial vehicle (UAV) by mounting an inertial navigation sensor (IMU) on it to measure its motion. IMU data is transferred to the host Macintosh over a Bluetooth serial link. The LabVIEW program on the host captures the data every 20 milliseconds and calculates velocity, position and angle values for feedback. The control loops on the host calculate new command values and transmit them to the Draganflyer via the radio controller.

One problem with doing all the software off board is that the range of the UAV is limited to the range of the two radio links. Also, control is impacted by radio noise. Another problem is the time delays caused by having to transfer the data to-and-from the host computer via serial radio links. The Sun SPOT is light enough and small enough to be mounted on board the helicopter to do all the fast real-time processing.

In a new project, we are converting a Lama dual-rotor helicopter (Fig. 5.) to a UAV by mounting an IMU and a Sun SPOT on board. We are investigating how much of the task planning, sensing, mapping, trajectory planning, navigation, control and driving software can be done on board the UAV. Our goal is to divide the application between the host and the embedded system on the basis of time requirements.
Acknowledgements

The authors wish to thank the Apple University Consortium for supporting this research through their “Innovation Grant” scheme.

References

Brega, R., A combination of system software techniques aimed at raising the run-time safety of complex mechatronic applications, PhD thesis, Zurich, Swiss Federal Institute of Technology (ETH), 2002.
Oberon, Oberon programming language, ETH Zurich, <http://www.oberon.ethz.ch>
Simon, D., Cifuentes, C., Cleal, D., Daniels, J. and White, D., Java(TM) on the Bare Metal of Wireless Sensor Devices -- The Squawk Java Virtual Machine, VEE’06, Ottawa, July 2006.