Abstract—Real-time operating systems (RTOSes) are required to run for years without human intervention, and never fail. Safety is a concern when they control physical equipment. One area of real-time operating system (RTOS) research is looking at the question: can developing an RTOS in a safe language result in a system that an errant process cannot crash? In this paper, we examine the advantages and problems of writing an RTOS in a safe language, namely Java. Then we design, implement and measure the performance of a minimal RTOS to schedule processes as threads on a Sun SPOT micro-controller.

Index Terms—real-time operating system, Java, Sun SPOT, embedded system, safe language, green thread, performance measurement, TINI, kernel overhead

I. INTRODUCTION

A. Real-time Software

Real-time software has much more stringent requirements than personal computer software [1]. It must execute within strict time deadlines, it must be correct, and it must be robust. For example, mobile robots are controlled by embedded computers that are expected to run for years without crashing or receiving software upgrades to fix bugs [2].

This requirement places huge demands on the operating system that supports it. Some embedded systems are written as single processes with no operating system to try to avoid this problem. The only advantage of this approach is that the programmer knows all the code.

Using an operating system (OS) enables the programmer to better focus on programming the real-time task because the OS abstracts away many of the low-level details. 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 is increased.

However, the programmer has to rely on the operating system to execute every process reliably and in time [3]. Also, the operating system must provide the low-level services the programmer requires to implement the task. Most Real-time Operating Systems (RTOSs) are written in C [4].

One area of RTOS research is looking at protecting processes by controlling access to resources with resource reservations [5]. The Minix 3 OS was designed to be highly dependable by restricting servers according to the principle of least privilege. Yet it cannot safeguard processes with stringent timing requirements. Experiments with a resource reservation server showed improved dependability of this OS in the temporal domain.

Another area of RTOS research is looking at protecting processes by writing the code in a safe language. The proponents of this approach claim that if we write both the OS and the processes in a safe language then we can guarantee that a process can not cause harm because the compiler has removed all the unsafe code.

A number of research projects have attempted to achieve this goal. The Burroughs B5000 did not have a memory management unit (MMU) so it relied on the Algol compiler to detect dangerous code [6]. X0/2 [7] is an RTOS developed at ETH in Zurich in Oberon to run on PowerPC embedded processors.

A recent project is the development of the Singularity operating system by Microsoft Research [6]. 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, although special hardware may be needed to protect against DMA problems. Each process has 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.

B. Embedded Development in Java

The Java Community Process is a formalized process that allows interested parties to be involved in the definition of future features of Java. Under this process, the Real-Time Specification for Java (RTSJ) was developed. It is a set of interfaces and behavioral
specifications to enable real-time programming in Java [8]. Their goal was to overcome the issues that limit the use of Java in real-time systems, such as its lack of support for priority based threads.

Sun Microsystems has promoted Java as a suitable language for programming small embedded systems for simple applications. The Sun SPOT system [9] consists of an embedded processor, an I/O card and an API. It does not include the extensions proposed in RTSJ.

In this paper we discuss the development of a minimal RTOS (JARTOS) to run on a sun SPOT [10]. Typical applications of small embedded systems include signal I/O, may have a network connection, but do not have disk drives. JARTOS is a minimal operating system that provides support for process scheduling, including application start up and shut down.

Brega [11] claims that Java is a safe language suitable for embedded systems. We have been asked why we write an OS in Java when it has threads, doesn’t the Java Virtual Machine (JVM) perform the same function as an OS? As the Sun SPOT JVM uses green threads [12], with a simple cooperative thread scheduler, JARTOS provides a way of managing the scheduling of these threads to achieve determinism.

One goal of this research is to investigate how to design a minimal RTOS for implementation in Java. “Minimal” is in the sense that its main goal is to provide a process scheduling and inter-process communication environment. Many embedded applications do not require more OS functionality than this. Also, if we cannot implement these in Java then it is unlikely that we can implement other operating system functionality in Java.

Our initial design ideas were all based on experience with similar minimal RTOSes written in assembler and C [4, 13, 14]. How would Java change these ideas? A second goal is to investigate ways to solve the low-level issues not normally supported in Java. A third goal is to evaluate the performance of an RTOS running on a Sun SPOT.

II. REAL-TIME OPERATING SYSTEM 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. To keep the number of interrupts to a minimum, polling of i/o is preferred to interrupts.

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 undefined period of time. For this reason, many real-time systems use cooperative scheduling. With a cooperative scheduling system, a process with a loop may hog the processor [15], so it must be able to timeout processes that are taking too long.

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. Safe usage of common data requires establishing and sticking to read and write protocols. Hardware failures also cause interrupts.

III. SAFE LANGUAGE ISSUES

The use of a safe language should guarantee that illegal instructions and invalid memory address errors do not occur due to data being written over code. The requirements of RTOSes [4] call for language features that are not found in most high-level languages, including C. 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, including mathematical errors.

The features that make a language safe [15] include: type safe variables, assertions to check conformance to design [17, 18], object references in place of pointer arithmetic, run-time data structure bounds checking, and real-time garbage collection. Sun SPOT Java has a real-time garbage collector.

There are a number of problems that we may have to solve when we are developing an OS in a safe, high-level language [15].

A. Low-Level Operations

Often there are low-level operations that cannot be coded in the high-level language [19], such as accessing specific memory locations, setting processor register bits, and return from interrupt instructions. These low-level constructs are machine dependent. They lack the redundancy required by the compiler to check them for consistency with the rest of the program, so the compiler is not able to protect the programmer against errors. Often some operations have to be programmed in an unsafe language. This code is called “trusted” code because it is locked away inside the OS so that the application programmers cannot modify it.

B. System Clock

A system clock is required to implement a deadline scheduler. Sun SPOT Java has a library function for attaching a thread to a hardware clock.

C. Interrupt Handling

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Do not use abbreviations in the title unless they are unavoidable.

Interrupt handling requires the ability to:
- store the address of the handler in a specific memory location,
- write a method that returns via the hardware to the interrupted process,
- change the return address to the calling address of another process, and,
- transfer data between the hardware and the application process.
Sun SPOT Java includes library functions for detecting and responding to interrupts.

D. Code Libraries

When a language does not support low-level features then either it has to be extended or it has to support calls to trusted library routines written in C. Sun SPOT Java includes a library with many functions for handling low-level operations, such as toggling digital outputs.

E. Networks

Sensor networks distribute processing to multiple processes over a network [20]. When processes are running on separate processors it is much more difficult to guarantee that deadlines are met. Sun SPOT Java includes a network class and library functions.

F. Threads

A Java programmer will observe strange behavior when the thread to use an object runs before the thread to initialize it. As Lee points out, in an article on concurrent programming, threads result in non-determinism [21]. A Java 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.

The Squawk virtual machine [23] on the Sun SPOT schedules its green threads with a cooperative scheduler. JARTOS controls the scheduling of these threads to achieve determinism. Another issue is that when multiple threads try to access shared data [22] they must be coordinated with locks.

IV. OVERVIEW OF JARTOS DESIGN

JARTOS is designed to use cooperative multiprocessing to schedule real-time processes. Each process is required to execute quickly and then give up the processor. Otherwise it will be timed out. Our design goals for a minimal RTOS included the following:

- Separate the application code from the OS code, so that the developer of the application only has to write application code and need make no changes to the OS.
- Reduce the indeterminacy in process execution time by using a cooperative scheduler to minimize the use of interrupts and control the scheduling of threads.
- Achieve real-time execution with a clock interrupt to set a scheduler flag to execute a timer process. Then the timer process checks synchronous processes (ones that run every n clock ticks) and sets scheduler flags to schedule any that are due to execute.
- Asynchronous process execution is scheduled on request from an other process including itself.
- Handle processes running over time with a timeout.
- Interrupt code is designed to be transparent to the processes. An interrupt handler sets a flag to schedule a process, reads or writes values, and then returns to the process it interrupted.
- Support a style of application programming where the real-time task is decomposed into several interacting processes.
- Use a data driven approach where OS functions are performed with processes.
- Measure the performance of processes for debugging and optimization.
- Test the OS by running it as a stand alone executable.

A. Scheduler

The OS kernel consists of a timer interrupt, a scheduler, a set of tables and supervisor call methods. The Main method starts the kernel by initializing the OS tables, enabling clock interrupts and calling the scheduler to schedule the first process.

The set of supervisor call methods enable processes to request work by the OS. Some supervisor calls set flags in OS tables. For example: the Run process call sets the run flag in the scheduler table, so that the specified process will be executed.

This functionality is placed in supervisor calls to protect the OS from poorly written application code. The OS and application are placed in separate classes to force the programmer to conform to this restriction.

The scheduler (Fig. 1) loops through the scheduler table (Table 1.) looking for the highest priority process whose flag is set to run. When it finds a process ready to run it resets the run flag, calls the performance probe and runs the process. When the process finishes, control is returned to the scheduler, which calls the performance probe and repeats the scheduling process.

B. Processes

Processes are separated into two groups: OS processes and application processes (Fig. 2). The OS processes

![Figure 1. Flow of control of JARTOS in response to clock tick.](image-url)

<table>
<thead>
<tr>
<th>Table I. OS TABLES.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table Name</strong></td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>Scheduler</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Event</td>
</tr>
<tr>
<td>Message</td>
</tr>
<tr>
<td>Performance</td>
</tr>
<tr>
<td>Circular Buffer</td>
</tr>
<tr>
<td>Common Data</td>
</tr>
</tbody>
</table>
TABLE II. PLANNED OS PROCESSES.

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>Maintains the timer table and sets flags for processes to run</td>
</tr>
<tr>
<td>Message Monitor</td>
<td>Checks for message arrival</td>
</tr>
<tr>
<td>Performance Analysis</td>
<td>Analyses data collected by performance probes</td>
</tr>
<tr>
<td>Timeout</td>
<td>Reports on process timeout</td>
</tr>
<tr>
<td>Garbage Collector</td>
<td>Cleans up heap</td>
</tr>
<tr>
<td>Idle</td>
<td>Runs when no other process require the CPU</td>
</tr>
<tr>
<td>Terminate</td>
<td>Disables timer interrupt and resets tables to stop scheduler</td>
</tr>
</tbody>
</table>

(Table 2.) handle the low-level operations required to keep the application processes executing. New functionality can be added to the OS by writing additional OS processes.

The lowest priority process is the Idle process, which sets its own run flag, so that when there is nothing to do the scheduler loops through it. Also it sets the run flag for the Event Monitor process to run to pole the digital inputs for events, if it exists. The Terminate process attempts to gracefully shut down the OS.

C. Synchronous Processes

When a clock interrupt occurs, it sets a scheduler flag to run the timer process. Synchronous processes have a count, a period and a phase. The period is the number of clock ticks to elapse between executions, and the phase is the position of the clock tick relative to the start of the period. These values control the timing of each synchronous process and are set by the application designer. The phase allows the designer to spread the load between clock periods.

When a process is enabled, its count is initialized to its period. For every clock tick, the timer process decrements the count for each enabled process. When a process’s count reaches zero, the timer process sets its run flag in the scheduler table and resets its count.

D. Application Processes

The purpose of an OS is to run applications. An application consists of several communicating processes. All applications are started by a Start Application process (Fig. 2) whose responsibility is to set up the processes required to perform the application and schedule at least one to run. Its run flag in the scheduler table is set when the OS initializes its tables. Thus, it will run as soon as the OS has done its startup house keeping. It is a low priority process as it usually runs only once.

The purpose of the Stop Application process is to gracefully shut an application down. It is often a low priority process, so it will only run when the application goes idle.

The Event Monitor process polls I/O to check for external events and sets scheduler flags to start application processes to respond to the events. The Event Monitor process is an application process because the events are specific to each application.

V. SUN SPOT

With the release of the Sun SPOT [9] in April 2007, Sun claims 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 IEEE802.15.4 wireless links.

Sun is tackling the issues of using Java to program embedded systems with the Squawk virtual machine (VM) [23]. It is a small JVM with a split architecture: a suite creator that runs on the host, and the SPOT embedded device. 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 improve performance, parts of the onboard VM and run time (e.g. the garbage collector) are translated from Java to C and then to machine code [24].

Applications are represented as objects that are instances of the Isolate class to isolate them from one another. Sun [9] claims that the Sun SPOT has no OS, but that OS functionality is built into Squawk with green threads [12]. They emulate a multi-threaded environment without relying on an underlying operating system. Green threads implement cooperative multiprocessing, so a thread has to release the processor before another thread can run. When waiting on something a thread is blocked on an event queue that is polled by the thread scheduler.

Interrupts are handled by assembler routines that set bits in an interrupt status word. The JVM 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.
VI. CODE DESIGN

JVM and hardware are considered to be the machine (unlike an assembler OS where only the hardware is the machine). JARTOS was designed to run as a single application on top of the JVM. We can also run it as an application in Mac OSX (host development environment) and on the TINI (Tiny Internet Interface) Java machine [25]. The application code is compiled and downloaded with the OS code.

The code consists of three Java classes: OS, Process and Application (Fig. 3). All OS tables are inner classes of the OS class. OS processes are mainly inner classes of the OS class, and inherit from the Process class. Application processes, including Event Monitor, Start Application and Stop Application processes are inner classes of the Application class, and inherit from the Process class too.

The use of inner classes means that the application designer will extend the Application class. As the OS and Process classes are separate files the designer can conform to the principles outlined in Section IV by not modifying these classes. This achieves the separation between application and OS shown in Fig. 2.

A. Main Method

The Main method in the OS class starts the execution of the OS. It constructs an instance of the OS class and an instance of the Application class. It constructs the OS processes, and sets their run flags. Then it enables the timer interrupt and calls the scheduler. The scheduler executes in a loop until the Terminate process runs. On its first loops, it runs OS processes and then the Start Application process.

Every process is defined by declaring an instance of the Process class, and overriding the process method with its process specific code. The Process class contains a standard template for process methods, and methods for working with processes. The Application class contains the processes for a specific task. The number of application processes is limited by a constant in the Process table.

VII. CODE IMPLEMENTATION

In this section, we give some code snippets to show how we coded specific operations in Java. We experimented with different code constructs, and the ones given here are from the current “best” implementation in the sense that their performance has been measured to be the best when running on Sun SPOT.

Also, “best” in the sense that the chosen language construct best suits the task. For example, in our initial design processes were scheduled with function calls. However, we found that threads resulted in a scheduling system that is easier to program and executes slightly faster.

All classes (Fig. 3) are declared as public, so that we can pass objects by reference. All the fields are declared as private with public set and get methods to write and read them. Supervisor calls and system methods are included in relevant classes. The OS tables are stored in arrays where OS functions can access them directly.

A. Passing Objects by Reference

We pass the objects of public classes by reference in order to avoid repetitive object creation. If we construct an instance of a class every time we need to call its methods, the runtime data will be lost. When we call the methods of an object, we should pass the object reference to this object. There are three steps:
1. declare a reference to an object and define a method to pass the object reference;
2. after constructing the object, call the reference passing method; and
3. call the method with the object reference.

For example, we need to call getClock and setEnablePerformProbes methods of the OS class in the run method of the Performance Analysis process so that we can do timing measurements for performance evaluation. First, in the Application class we declare a reference named refToOS to the OS object (Fig. 4a) and define a method called passRef() to pass the object reference.

Second, in the calling class we construct an instance of the OS class, and then we call the reference passing method (Fig. 4b). Third, in the calling class we call the methods, getClock() and setEnablePerformProbes(), by the object reference (Fig. 4c).

```java
OS refToOS;
public void passRef(OS myOS){
    refToOS=myOS;
}

a. Declare a reference to the OS object

OS myOS=new OS();
myOS.passRef(myOS);
b. Construct instance of OS class

if(!refToOS.getClock()%20==0){
    refToOS.setEnablePerformProbes(false);
}
c. Call methods by object reference
```

Figure 3. Class diagram of JARTOS.

Figure 4. Example code for passing object by reference.
B. Processes

Application processes use the above reference to set up other application processes to run. The code in Fig. 6 shows how one process can load and run a second process. It is based on the standard template for an application process. The run method of the application process class overrides the run method of the Process class with the application specific code.

As shown in the run method of the Start Application Process in Fig. 5 there are several steps to running a new application. First, a new application (App1) is created and passed to the OS. Then three flags are set in OS tables: one in the Process Table to indicate that the process is loaded, one in the Scheduler table to indicate that the process is enabled to run, and a second in the Scheduler table to tell the scheduler to run it.

C. Scheduling Processes

We schedule processes with threads, making use of the thread scheduler in the JVM. There are three core threads in JARTOS: main thread, scheduler thread, and timer interrupt thread (Fig. 6). At startup, the main thread creates the scheduler thread and the timer interrupt thread, and then waits until the OS is terminated. Each process is defined as a thread, so to run it the scheduler calls its run method with thread start and waits for it to complete with thread join (Fig. 7).

D. Handling Timer Interrupts

To implement the clock that ties the processes to real-time the Main method creates and starts a timer interrupt thread (Fig. 8). The run method of the timer interrupt (Fig. 9) sets up the AT91 hardware timer counter [26] and then waits on the hardware interrupt. Every time an interrupt occurs (clock tick) it calls the timerInterruptHandler method (Fig. 10) to handle the interrupt. It increments the clock, sets the Timer process to run and executes the timeout function (Fig. 11) to determine whether the current process has gone over time.

VIII. PERFORMANCE MEASUREMENT OF JARTOS

An issue that is of interest is: “What is the overhead of an RTOS written in JAVA?” In particular, the execution time of the kernel places constraints on the performance of applications [27], especially when they are decomposed into lots of small processes to work with cooperative scheduling. In this section, we present some measurement results for processes running under JARTOS.

A. Java Performance

The Sun SPOT processor is an ARM 920T [28] that has a performance of 200 MIPS with a 180 MHz clock. We chose “\(i = i+1;\)” as a representative instruction for evaluating performance. To conduct this measurement we executed this instruction thousands of times in a while loop and measured the total execution time. From this we subtracted the total execution time of an empty while loop and divided by the number of loop iterations. The

```java
class TimerInterrupt implements Runnable, TimerCounterBits{
    public void run(){
        int cnt = (int)(60000/0.5342); //millisecond
        timerConfigure(TC_CAP11TC_CPTC1GTC1CLK1_MCK132);
        timer.enableAndReset();
        while(true){
            timer.enableIRQ(TC_CPCS);
            timer.waitForIRQ();
            timer.status();
            timerInterruptHandler();
        }
    }
}
```

Figure 8. Code to start timer interrupt thread.

```java
class StartApplication extends Process{
    public StartApplication(String procName,
        int synWaitTime, int synWaitPhase,
        int timeout, int event1, int event2,
        int message1, int message2,
        long processExitAddress){
        super(procName, synWaitTime,
            synWaitPhase, timeout,
            event1, event2, message1, message2,
            processExitAddress);
    }
    public void run(){
        Application1 App1 = new Application1("Application1",1,2,1,0,0,0,0);
        refToOS.passRef(App1);
        refToProcessTable.loadProcess(App1);
        refToScheduler.enableProcess(App1);
        refToScheduler.runProcess(App1);
    }
}
```

Figure 5. Code of Start Application process.

```java
public class StartApplication extends Process{
    public StartApplication(String procName,
        int synWaitTime, int synWaitPhase,
        int timeout, int event1, int event2,
        int message1, int message2,
        long processExitAddress){
        super(procName, synWaitTime,
            synWaitPhase, timeout,
            event1, event2, message1, message2,
            processExitAddress);
    }
    public void run(){
        Application1 App1 = new Application1("Application1",1,2,1,0,0,0,0);
        refToOS.passRef(App1);
        refToProcessTable.loadProcess(App1);
        refToScheduler.enableProcess(App1);
        refToScheduler.runProcess(App1);
    }
}
```

Figure 5. Code of Start Application process.

```java
Figure 6. Thread based scheduling of processes.
```

```java
Figure 7. Excerpts from the code of the scheduler.
```

```java
Figure 8. Code to start timer interrupt thread.
```

```java
Figure 9. Timer interrupt code.
```
execution time for integer increment is 0.47 μsec and the execution time for an empty while loop is 1.01 μsec. We found that the execution of library functions is very good on the SPOT. For example, getCurrentTime() takes 5.8 μsec. Functions involving access to hardware are fast indicating excellent design of I/O hardware and software. From these measurements we expect that the Sun SPOT Java will be able to implement small real-time applications.

B. Process Execution Time

The first component of JARTOS that we wanted to measure was the timer interrupt handler (Fig. 12). Using a software call to the handler, we measured its execution time to be 142 μsec. This is a good result as it indicates that the basic function of clock handling will only use 0.5% of CPU time when running with a 30 msec clock tick. Also, it starts to define the limits of performance of JARTOS.

In its simplest instantiation, JARTOS runs three synchronous processes (Fig. 12): Timer, Idle and Test processes. The Test process contains the code whose performance we wish to measure. After the Test process is executed the required number of times, the Performance Analysis process is scheduled to print out the recorded data, and finally the Terminate process stops JARTOS.

An example Test process (Fig. 13) blinks a LED on the Sun SPOT Demo board three times on each execution. The clock was set to tick every 15 msec. We used this Test process to validate the flow of control of JARTOS by looking for excessive execution times and anomalies in execution time.

Scheduler loop time includes time to schedule a process, to read the clock and to execute the process (Fig. 14). Scheduling time is the time that the scheduler takes to select which process to run. Process time is the execution time of that process. From a performance trace [29], we calculated the average process time and average scheduling time for each process (Table 3). We found that a process always has the same scheduling time, which increases with the process number, because the scheduler iterates down the scheduler table until it finds a process to run. These results further define the performance limits of JARTOS, with the scheduling of simple processes taking about 10% of their execution time. However, a scheduling time of 510 μSec is slow compared to the process switching time of 12 μSec reported for RTOSs written in C [4]. However, they were measured on very different hardware: a Renesas M16C/62P 16 bit microcontroller with a 24MHz clock, so direct comparison is not possible.

For this simple example, the total execution time (Figs. 12), to handle one interrupt, schedule the three processes (Timer, Test and Idle) and execute each process once is 14.392 msec. The OS overhead, the time to execute the interrupt handler and schedule and execute the Timer process, is 5.162 msec. With a 15 msec clock tick, this
We validated this approach by reducing the time between clock ticks to produce an overload. The average time between executions of the test process was longer than the designed two clock ticks, showing that it had failed to execute on some ticks. The overload did not crash JARTOS, it caused the test process to run late, and not to run at all on some clock ticks.

When it was later than two clock ticks it did not run because the scheduler run flag had been over written to call the next execution. The failure to run freed up CPU time for the process to execute on subsequent calls. An alternate design, with a run count instead of a run flag, would have avoided the failure to run but the process would get successively later and the OS would always be overloaded.

IX. EXPERIENCES

The design of JARTOS was derived from our experience in developing and applying small real-time systems in non-safe languages. We reflected on this experience in the light of reported developments of embedded systems in safe languages, particularly XO/2 [11].

We found that implementing the design required two paradigm shifts in our thinking. One was from structured to object-oriented coding approaches, and the second was from using non-safe to safe language constructs. The question we regularly asked is: “What language construct can be used to implement the desired feature?”

JARTOS is written in Java. Java can implement all the high-level functions in the design. The Sun SPOT library includes many functions for low-level operations. Without that library it would not have been possible to implement some of the required low-level features.

A. Scheduling

One area where Java provided an alternative to the original design of JARTOS is how the processes are scheduled. The original design was to define a process as a function, store its reference in the scheduler table and call it directly. When we implemented this approach we found that it was slower than running the process as a thread. The thread approach makes use of the Sun SPOT’s green thread mechanism, thus achieving better integration of the OS with the JVM.

B. Inter-process Communication

We implemented inter-process communication with supervisor calls in Java. Messages are handled by a message class with get, send, receive and release functions. Buffering between hard real-time operations (such as reading inputs) and soft real-time operations (such as processing those inputs) is done by a producer/consumer queue that is an instance of a circular list class. Similarly, common data is supported by a class.

C. Low-Level Issues

Choosing to use cooperative scheduling rather than pre-emptive scheduling eliminates the need for the

<table>
<thead>
<tr>
<th>Process no.</th>
<th>Process name</th>
<th>Process time/μs (medium ± max)</th>
<th>Scheduling time/μs (medium ± max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Timer</td>
<td>4510±100</td>
<td>510±100</td>
</tr>
<tr>
<td>5</td>
<td>Test</td>
<td>6400±100</td>
<td>530±100</td>
</tr>
<tr>
<td>31</td>
<td>Idle</td>
<td>1700±100</td>
<td>600±10</td>
</tr>
</tbody>
</table>

overhead is 34.4% of CPU time. Increasing the clock tick to 100 msec reduces it to 5.162% of CPU time.

These results define the performance limits of JARTOS. They indicate that it is suitable for small tasks with a 30msec clock tick through to larger tasks with a 200msec clock tick. Whether JARTOS can support a particular embedded application [10] is determined by whether the code to perform the task can be executed within the clock tick required to achieve the required deadlines.

C. Process Flow of Chart

We can use the data from the performance probes to study the sequence of execution of the processes. The application designer can use a process flow-of-control diagram (Fig. 15) as a design tool to plan the scheduling of the synchronous processes [10]. Then, she can evaluate the correctness of the implementation by comparing the measurements to the design.

The Test process in the test application is scheduled as a synchronous process. A measurement of the sequence of execution of the processes is shown in Fig. 15. The clock was set to tick every 15 msec. As expected, it scheduled the Timer process (p1) to run once every 15 msec. The Test process (p5) was set to run every 2 clock ticks with a phase of 1.

The rest of the time is used by the Idle process (p31) which runs at least twice every clock tick in this example, confirming that when JARTOS has no processes to run it runs the Idle process. The processes execute in order of priority, showing that the Timer process has the highest priority.

A third use of the performance data is reliability testing. To test its reliability an application is set to run for a long period of time (one or more days). With the performance data for the whole run, the application programmer can compare the process flow-of-control at the end with that at the start. Also, the count of executions of each process can be used to check whether the system failed to execute a process for some reason.

We validated this approach by reducing the time between clock ticks to produce an overload. The average time between executions of the test process was longer than the designed two clock ticks, showing that it had failed to execute on some ticks. The overload did not crash JARTOS, it caused the test process to run late, and not to run at all on some clock ticks.

When it was later than two clock ticks it did not run because the scheduler run flag had been over written to call the next execution. The failure to run freed up CPU time for the process to execute on subsequent calls. An alternate design, with a run count instead of a run flag, would have avoided the failure to run but the process would get successively later and the OS would always be overloaded.

IX. EXPERIENCES

The design of JARTOS was derived from our experience in developing and applying small real-time systems in non-safe languages. We reflected on this experience in the light of reported developments of embedded systems in safe languages, particularly XO/2 [11].

We found that implementing the design required two paradigm shifts in our thinking. One was from structured to object-oriented coding approaches, and the second was from using non-safe to safe language constructs. The question we regularly asked is: “What language construct can be used to implement the desired feature?”

JARTOS is written in Java. Java can implement all the high-level functions in the design. The Sun SPOT library includes many functions for low-level operations. Without that library it would not have been possible to implement some of the required low-level features.

A. Scheduling

One area where Java provided an alternative to the original design of JARTOS is how the processes are scheduled. The original design was to define a process as a function, store its reference in the scheduler table and call it directly. When we implemented this approach we found that it was slower than running the process as a thread. The thread approach makes use of the Sun SPOT’s green thread mechanism, thus achieving better integration of the OS with the JVM.

B. Inter-process Communication

We implemented inter-process communication with supervisor calls in Java. Messages are handled by a message class with get, send, receive and release functions. Buffering between hard real-time operations (such as reading inputs) and soft real-time operations (such as processing those inputs) is done by a producer/consumer queue that is an instance of a circular list class. Similarly, common data is supported by a class.

C. Low-Level Issues

Choosing to use cooperative scheduling rather than pre-emptive scheduling eliminates the need for the
scheduler to save and restore state, to have access to processor registers, and to be able to return from an interrupt to a different process.

The Sun SPOT includes two I/O libraries: one for the peripherals on the main board and one for the peripherals on the demo board. These enable the reading and writing of analog and digital values in Java. Thus the problem of accessing I/O is solved by wrapping a Java class around a low-level “trusted” class.

D. Garbage Collection

The Sun SPOT JVM includes a garbage collector, so we have not implemented one as a process. While we have not measured its impact on execution directly, we have observed stretching of up to 3 msec on a digital output pulses from a CPU intensive process [10].

E. Timeouts and Thread Safety

One reason for choosing thread based scheduling is that threads should provide a neat and safe way of dealing with timeouts. When the timer interrupt handler detects that a process has run over time it would stop it, enabling the scheduler to run the timeout process to gracefully shut down or attempt to recover the application.

Unfortunately, in the latest version of Java, thread stop has been deprecated because it is considered unsafe in some circumstances. We believe that the changes have made the thread class unsafe. If a Java program cannot stop an errant thread then the only option when the OS is written in Java is manual intervention. In fact, when a Sun SPOT program gets into an infinite loop, the only way to get out is to push the hardware reset button. Thus, we conclude that the Java thread model is badly broken.

F. Safe Languages

The main argument against the use of safe languages, and the reason that a lot of code is written in unsafe languages, is that a desired function cannot be implemented in a safe language. Sadly the designers of safe languages tend to reduce the capability of the language to overcome safety problems. This forces programmers to use unsafe languages instead of safe languages, defeating the goal of safe language design. The Modula II approach of putting unsafe instructions in an explicit system class appears to be a workable compromise.

Due to the deprecation of thread stop, we cannot write code to stop a process that has run overtime. The best we can do is to attempt to gracefully shut down the application and wait for human intervention to fix the cause of the timeout and reset the system. At this point the application is shut down and the OS is paused, so we do not achieve our safe language goal, but the system is safe.

X. CONCLUSIONS

This paper reports on our experience in developing a minimal RTOS in Java to run on the Sun SPOT. We have explored the advantages and associated problems of writing an RTOS in a safe language. JARTOS takes advantage of Java language constructs to achieve many functions.

It schedules processes synchronously as threads in response to a clock tick, which is generated by a timer interrupt handler that is programmed as a thread attached to a hardware timer. Also, the way JARTOS schedules the threads results in deterministic execution of the Java threads, overcoming one of the significant criticisms of using Java for real-time systems.

We have been able to solve all the low-level issues except timeouts. The libraries provided with the Sun SPOT to handle input/output functions are essential for writing an embedded system in Java.

A problem with co-operative multitasking is that when a process fails to terminate, the system stops executing the application. One solution is to timeout the hung process. We thought that we could do this by stopping the thread in Java, but this feature has been removed from the language. So we designed a way for the timeout to kill the application gracefully. However, it requires manual intervention to kill the errant Java application and start again.

To measure the performance of JARTOS, we installed performance probes to produce a trace of process execution. JARTOS is synchronized to a clock tick, and achieves the scheduling of a simple test application with a clock tick of 15 msec and a process switching time of 510 μsec on the Sun SPOT. The OS overhead is 5.15 msec or 34.4%. If the clock tick is increased to 100 msec, the OS overhead is reduced to 5.16%, indicating that the system is suitable for programming embedded applications that have a 10 to 50Hz update requirement.

ACKNOWLEDGMENT

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. We wish to thank Sun Computers for providing us with a Sun SPOT kit, and the Apple University Consortium (AUC) for providing us with a Macintosh Host for development and a Sun SPOT kit for this research.

REFERENCES


Qinghua Lu is a fourth year PhD student at the School of Computer Science and Engineering, University of New South Wales and graduate researcher at the Software Systems Research Group, NICTA, Australia. Her research interests include business process management, business-driven IT management, autonomic computing, service-oriented architecture, cloud computing, and embedded real-time operating systems.

Phillip John McKerrow is an honorary principal fellow in computer science at the University of Wollongong, Australia. His research interests include perception of the environment with ultrasonic sensing and mobile robot navigation. McKerrow graduated from the University of New South Wales with a BE (Hons) in electrical engineering, and from the University of Wollongong with an ME in electrical engineering and a PhD in computer science.

Zhi Quan Zhou received the BSc degree in computer science from Peking University, China, and PhD degree in software engineering from The University of Hong Kong. He is currently a senior lecturer in software engineering at the University of Wollongong, Australia. His research interests include software testing, maintenance, symbolic analysis, and Web search engines.