Real-time operating system in Java

Qinghua Lu
University of Wollongong

Real-time Operating System in Java

A thesis submitted in fulfilment of the requirements for the award of the degree

Master of Computer Science -Research

from

UNIVERSITY OF WOLLONGONG

by

Qinghua Lu

School of Computer Science & Software Engineering

August, 2007

Dedicated to

My Parents,

Lu Changyou and Luo Xiue!

The following papers were written as part of this research.

1. M[c]Kerrow, P.J., Lu, Q., Zhou, Z.Q. and Chen, L. (2007), Developing real-time systems in Java on Macintosh, ***Submitted to AUC'07***, Apple University Consortium, Gold Coast, September, 23-26, 2007.

2. M[c]Kerrow, P.J., Lu, Q., Zhou, Z.Q. and Chen, L. (2007), Software development of embedded systems on Macintosh, ***Submitted to AUC'07***, Apple University Consortium, Gold Coast, September, 23-26, 2007.

Declaration

I, Qinghua Lu, declare that this thesis, submitted in fulfilment of the requirements for the award of Master of Computer Science -Research, in the School of Computer Science & Software Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Qinghua Lu
31 August 2007

Table of Contents

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
EDF	Earliest-deadline-first
IDE	Integrated Development Environment
I/O	Input/Output
JVM	Java Virtual Machine
Mac OS	The Macintosh Operating System
Mac OSX	the latest version of the Mac OS
MMU	Memory Management Unit
msec	millisecond
msg.	message
no.	number
OS	Operating System
proc.	process
PC	Program Counter
rti	return from interrupt
RM	Rate-Monotonic
RTOS	Real-time Operating System
RTOSes	Real-time Operating Systems
Sun SPOT	Sun Small Programmable Object Technology
TCB	Task Control Block
TCP/IP	Transmission Control Protocol/Internet Protocol
TINI	Tiny InterNet Interface
TNI	TINI Native Interface
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language

Abstract

Real-time operating systems (RTOSes) are required to run for years, and never fail, without human intervention. Safety is the primary concern for RTOSes because they usually control physical equipment. One strand 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 can't crash? Choosing a good programming language can significantly improve the safety of the RTOS. In this thesis, we examine the advantages and associated problems of writing RTOSes in a safe language, namely Java.

We design an RTOS named JARTOS that schedules processes on a micro-controller called TINI. The code of the JARTOS system is mainly written in Java, since Java provides both static and dynamic safety. The Java compiler handles potentially unsafe operations rather than the programmer. Also, Java includes run-time support to catch and handle run-time errors.

JARTOS is designed to be a time-sharing system, where cooperative multiprocessing is used to schedule real-time processes. JARTOS switches processes on a timer interrupt. Each process is required to execute quickly and then give up the processor. Otherwise it will be timed out. To implement a timeout, JARTOS supports a timer interrupt that regularly updates a clock and checks for timeouts. To keep the number of interrupts to a minimum, input/output is done using polling where possible. Also, interrupts code is designed to be transparent to the processes. An interrupt handler sets flags and values, and then returns to the process it interrupted.

In the context of achieving real-time performance, we look at the issues of implementing our system design in Java. We introduce how we used Java constructs to implement the design of JARTOS, and how we solved the low-level issues.

RTOSes have to guarantee that real-time processes execute within specified time deadlines. Loss of synchronization can occur when deadlines are not met. Timing problems are often very difficult to find. In JARTOS, we designed a set of performance measurements to investigate timing problems. These performance measurements are carefully designed to provide the right information at minimal cost in performance. Performance of TINI and JARTOS are measured and discussed.

Acknowledgments

First, I would like to express my deepest appreciation to my supervisor, A./Prof. Phillip McKerrow, for his guidance, patience, support and enthusiasm during the process of this thesis.

I would like to thank my co-supervisor, Dr. Zhi Quan Zhou, for guidance, patience and help during my study.

Thanks also to my colleagues Sherine Antoun and Li Chen for their help and friendship.

From the bottom of my heart, I would like to express my special thanks to my parents, Lu Changyou and Luo Xiue, for their endless love, encouragement and support during my study and all of my life.

Finally, special thanks to my fiancé, Qiao Shuaichang, for his endless love, helpful advice. Without his support, the work in this thesis would not have been possible.

Chapter 1

Introduction

1.1 Background and Motivation

Real-time software has much more stringent requirements than personal computer software [Laplante, 2004]. It must execute within strict time deadlines, it must be correct, and it must be robust. Every modern car has an embedded computer controlling its engine. It is expected to calculate the correct fuel/air mixture every time the accelerator is pressed. Also, the computer is expected to run for years without crashing or having to perform software upgrades to fix bugs.

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, i.e. they are a single process. 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 (RTOS) research is looking at the question: can developing an operating system in a safe language result in a system that an errant process cannot 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 cannot be crashed if we use a safe language? This goal raises a further question: are the algorithms commonly used in RTOSes 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 does not have a memory management unit (MMU) so it relies on the Algol compiler to detect dangerous code [Tanenbaum, *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, 2007]. 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 have developed a simple RTOS named JARTOS in Java. Brega [Brega, 2002] claims that Java is a safe language suitable for embedded systems. One goal of this research is to investigate the advantages and disadvantages of developing an RTOS in Java. JARTOS

executes tasks on a microcontroller named TINI, which is to be mounted on our flying robot. The TINI platform [TINI, 2007] provides an extensible Java runtime environment.

We want to take an approach to developing an RTOS that uses advances in computer science. So, we try to write JARTOS using the design and compile technology of Java. The code of the JARTOS system is mainly written in Java, since Java provides both static and dynamic safety. The Java compiler handles potentially unsafe operations rather than the programmer. Also, Java includes run-time support to catch and handle run-time errors.

1.2 Objectives

Safety is the primary concern for RTOSes because they usually control physical equipment. Choosing a good programming language can significantly improve the safety of the RTOS. In this thesis, we will examine the advantages and associated problems of writing real-time operating systems (RTOSes) in a safe language, namely Java.

In this thesis, we will introduce the design of JARTOS that schedules tasks on TINI. JARTOS is small enough to run on TINI, which is to be mounted on our flying robot to do all the fast real-time processing. JARTOS is designed to be a time-sharing system, where cooperative multiprocessing is used to schedule real-time processes. We do not use priority preemptive scheduling because it is a cause of RTOS 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.

JARTOS switches processes on a timer interrupt. Each process is required to execute quickly and then give up the processor. Otherwise it will be timed out. To implement a timeout, JARTOS supports a timer interrupt that regularly updates a clock and checks for timeouts. To keep the number of interrupts to a minimum, input/output will be done using polling where possible. Also, interrupts code is designed to be transparent to the processes. An interrupt handler sets flags and values, and then returns to the process it interrupted.

In the context of achieving real-time performance, we will look at the issues of implementing our system design in Java. We will introduce how we used Java constructs to implement the design of JARTOS, and how we solved the low-level issues. We will try to design a set of tests that will thoroughly test the flow of control, performance and reliability of the OS (Operating System). They will be extended and run every time any part of the OS is changed. Then we can say JARTOS has passed a given set of tests, and hence has been updated without the changes reducing the correctness, performance or reliability of the existing code.

RTOSes have to guarantee that real-time processes execute within specified deadlines. Loss of synchronization can occur when deadlines are not met due to timing problems. Timing problems are often very difficult to find. In this thesis we try to design a set of performance measurements to investigate the timing problems. These performance measurements will be carefully designed to provide the right information at minimal cost in performance. Performance of TINI and JARTOS are measured and discussed.

1.3 Outline of the Thesis

- Chapter 2 introduces the definition and general concepts of RTOS.
- Chapter 3 focuses on real-time programming languages. The features of a safe language are presented. Different real-time programming languages are discussed in this chapter. Examples of OS developed in a safe language are also introduced in this chapter.
- Chapter 4 describes the design of JARTOS.
- Chapter 5 presents the code design of JARTOS. TINI runtime environment is introduced in this chapter.
- Chapter 6 introduces the issues of implementing our system design in Java.
- Performance of TINI and JARTOS are measured and discussed in Chapter 7.
- Conclusions are made in Chapter 8, where a summary of the work is given and future work is outlined.

Chapter 2

Real-time Operating Systems

A real-time operating system is an operating system designed for real-time applications, such as industrial robots, mobile phone and spacecraft. It must execute within strict time deadlines, it must be correct, and it must be robust [Purser and Jennings, 1975]. This chapter introduces the definition and general concepts of RTOS. Different concepts such as tasks, scheduling, timer, event handler, inter-process communication, memory management and networking will be presented.

2.1 What is a RTOS?

A real-time operating system is an operating system required to ensure that real time processes execute correctly within specified response-time constraints [Laplante, 2004]. RTOSes must guarantee that processes meet time deadlines. Anything that causes indeterminism in the execution time makes it harder to achieve that guarantee.

There are two types of programs in RTOS: hard real-time program and soft real-time program [Liu, 2000]. A hard real-time program must guarantee to finish its execution before a time deadline. A soft real-time program only has to meet its deadline on average.

2.2 Basic Concepts of the RTOS

2.2.1 Tasks

A task is an independent activity performed by RTOS [Barrett and Park, 2005]. As shown in Figure 2.1, a task can be in one of five states: Dormant, Ready, Running, Waiting, or ISR (Interrupt Service Routine) [Labrosse, 1999].

- Dormant: The task resides in memory but has not been available to the kernel.

- Ready: The task is waiting to execute but its priority is lower than the currently running task.
- Running: The task has control of the CPU.
- Waiting: The task has been delayed from execution since it requires the occurrence of an event
- ISR: The task is in the ISR state when an interrupt has occurred and the CPU is servicing the interrupt.

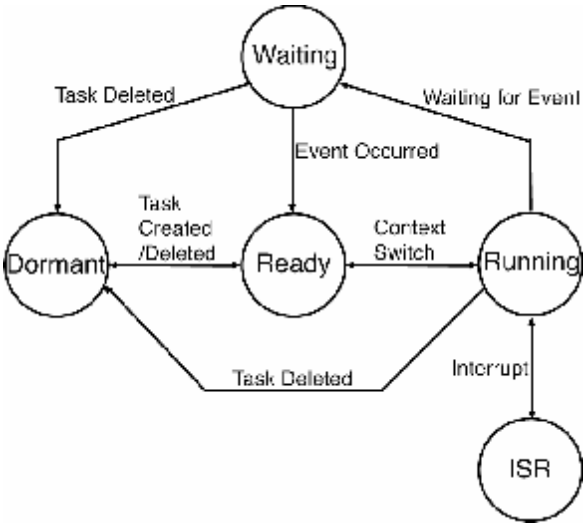

Figure 2.1 Task states

Real-time tasks can be classified by their timing requirements as hard-real-time task, soft-real-time task and non-real-time task [Brega, 2002]. A hard-real-time task must finish its execution correctly before a deadline. Missing the deadline will cause the task to fail. A soft-real-time task is desired to finish its execution before a deadline. The deadline is only a soft deadline that is not critical to the function of task. A soft real-time task should meet the deadline on average, where a hard real-time task must meet it every time. A non-real-time task is a task with no real-time requirements.

Real-time tasks can be further classified, according to the predictability of their arrival, as periodic, aperiodic and sporadic [Krishna and Shin, 1997]. A periodic task is a task that is executed repetitively at regular intervals of time. It can be prescheduled since it is known

to the developer. An aperiodic task is a task whose execution time cannot be predicted because its occurrence depends on an event. A sporadic task is a periodic task with a bounded interval time.

2.2.2 Design Architecture

There are two kinds of basic design architecture in RTOS: event-triggered architecture and time-triggered architecture [Nissanke, 1997]. An event-triggered RTOS switches tasks as a response to an external event. A time-triggered RTOS switches tasks in accordance with a clock. It is often complex to model an event-triggered system because many interrupts and priorities may be present. The programmers need to pre-allocate the system's time resources for a time-triggered system. [Brega, 2002]. Although these two design architectures have different concepts, they often exist at the same time in many RTOSes.

2.2.3 Scheduling

Scheduling is an essential function of an RTOS. The goal of scheduling is to guarantee that the performance of the system meets the time requirements [Krishna and Shin, 1997].

Priority-based scheduling can be classified as preemptive scheduling or non-preemptive scheduling. Preemptive scheduling guarantees that "each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current task's context in its Task Control Block (TCB) and switches to the higher-priority task. [Laplante, 2004]" A preemptive kernel guarantees that an interrupt is used to suspend the currently running task and invoke the kernel to decide which task will run next.

Non-preemptive scheduling is also called "cooperative multiprocessing," because tasks must cooperate with each other to share the CPU in this environment. In non-preemptive kernels, the task must run quickly without any interruption and explicitly give up control of the CPU. Therefore, inter-process communication is very important in a non-

preemptive kernel RTOS [Barrett and Park, 2005]. Exclusive access is allowed to shared resources in non-preemptive scheduling, thus the synchronization overhead is eliminated.

The major difference between preemptive scheduling and non-preemptive scheduling is what controls the CPU. In a non-preemptive kernel, the task gives up control of the CPU back to the RTOS. In a preemptive kernel, the kernel decides which task will run next and whether the current task will be preempted [Barrett and Park, 2005]. The most important drawback of a non-preemptive kernel is responsiveness [Horton, 2000]. Compared to non-preemptive scheduling, preemptive scheduling has better system responsiveness. Hence, a preemptive kernel is used in responsiveness-critical systems.

However, preemptive scheduling may be impossible or very expensive due to practical problems in RTOS scheduling [Horton, 2000]. It also can switch context at an inappropriate time. Priority-preemptive scheduling is a main cause of indeterminism in the execution time of real-time systems. Also, the design of a preemptive kernel is much more complicated than that of a non-preemptive kernel. Non-preemptive scheduling does not need to guard shared resources and data. Further, in a non-preemptive kernel, the interrupt latency is much lower than in a preemptive kernel.

RTOS scheduling can be further classified as static scheduling or dynamic scheduling. In static scheduling, all priorities are assigned to tasks as constants at design time. The priority of a task remains fixed for the lifetime of the task [Brega, 2002]. A rate-monotonic (RM) algorithm is a typical static scheduling algorithm in which a task is assigned a priority according to its execution time, so that a shorter period task is assigned a higher priority than a longer period task [Li, Potkonjak and Wolf, 1997]. In dynamic scheduling, priorities are assigned to tasks at run time. The priorities may be changed over time based on execution parameters of tasks [Brega, 2002]. Earliest-deadline-first (EDF) is a well-known dynamic scheduling algorithm in RTOS. With EDF scheduling, the task with the earliest deadline will always be assigned the highest priority.

The major disadvantage of dynamic scheduling is the higher run-time cost with respect to

static scheduling [Brega, 2002]. Undoubtedly, dynamic scheduling is more complicated than static scheduling because of extra computation for priority. However, compare with static scheduling, dynamic scheduling is more flexible and responsive in implementation.

2.2.4 Polling and Interrupts

Polling is a routine that continuously checks each device to see if the status of the device has been changed. In a polling-based program, the CPU keeps reading the status register of each device. If a device has completed the required task, the status of the device will be changed. Polling is simple to design. However, CPU has to waste its time to continuously check the devices over and over again. When the hardware is designed to support polled operation, the CPU does not have to or poll as often, reducing the time wasted on regular polling.

An interrupt is a signal indicating that the processor should stop the current process and service the interrupt. The function of interrupt handling is [Purser and Jennings, 1975]:

1. To identify the interrupt
2. To call the appropriate process.
3. To schedule.

When an interrupt occurs, the processor saves the state of the current process, and then services the interrupt task. After completing the interrupt service routine, the processor restores the state of the current process and resumes the original process. Interrupts have two types of latency: the time from where the interrupt signalled until the interrupt handler starts execution, and the time to save system state. However, interrupts are a cause of indeterminism in process execution time because they cause the processor to stop what it is doing and service the interrupt. They take the processor away from the running process for an indeterminate period of time.

2.2.5 Timer

A timer is a tool for checking the elapsed time and the process switching. “Polling a timer is a wasteful use of processor cycles. The code must contain a subroutine that frequently

checks the timer. In most applications, the timer should be interrupt-driven [Ford and Topp, 1988].” A timer is generated by a hardware clock periodically. Its purpose is to update the clock, set the timer process to run and time out any processes that are taking too long. The timer interrupt allows processes to be suspended for integral number of ticks and sets timeouts when processes are waiting for an event to occur. It is also responsible for returning program counter (PC) to where it was prior to hardware service interrupt. Normally the timer does not have any interaction with any other process.

2.2.6 Threads and Events

A thread is a single sequence of program execution. Threads are used to split a program into multiple simultaneously running tasks. An event is a change in the state of the system, such as a mouse click, timer timeout. It requires the execution of a process to handle it. An event handler is a program that is executed in response to events. The execution of the event handler is triggered by the reception of a hardware event or a software event.

Threads and events can be both used in concurrent programming. Threads can execute the task efficiently. Thread-based programs run faster on a computer system that has multiple CPUs. But as Lee points out, in an article on concurrent programming, threads result in non-determinism [Lee, 2006]. 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 do not 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, 1996]. If a lock is forgotten, it may cause corrupted data. Writing data access synchronization can become difficult, because circular dependencies must be avoided.

Events have better performance in managing concurrency than threads [Ousterhout, 1996]. Events are successful enough to solve almost all problems instead of threads [Gustafsson, 2005]. An event-based program runs even faster than thread-based program on a single CPU. There is no locking overheads and context switching in event-based

programs. Event-based programs reduce the complexity of programming and the overall usage of memory. Programming with events also makes debugging programs easier.

2.2.7 Inter-process Communication

Executing multiple processes to perform a single task requires those processes to share data [Richard, 1999]. Inter-process communication is a set of techniques supported by an RTOS that allows the flow of data between processes. Theses processes can be running on the same processor or on different processors connected by a network. Methods of doing this are common-data storage, message passing, and producer-consumer queues. The choice of inter-process communication method depends on the type of data being communicated and environment of communication.

2.2.8 Memory Management

Memory management has a significant impact on the security and reliability of the RTOS. Good memory management is essential in order to maintain the efficiency of the RTOS [Ethernut, 2007]. Improper memory allocation can destroy the system's determinism, for example, buffer overflow or underflow. Garbage collection is a technology for automatic dynamic memory management. It is used to identify and release the memory that is no longer being used by processes. Garbage collection can avoid memory leaks that may cause an operating system to run out of memory and crash.

2.2.9 Testing and Performance Measurement

Debugging and rigorous testing of real-time embedded systems remains a difficult problem. A network connection facilitates the development of better tools than a serial link [M^{c}Kerrow *et al.,* 2007]. Using a network, data collected on the embedded system can be analyzed on the host. Also, the embedded system can be controlled from the host. Performance monitoring and debugging take time to execute and consequently they impact on the timing performance of the processes running on the embedded processor. 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.

2.2.10 Networking

A Network processor is a programmable chip, which is optimised to support the implementation of network protocols at the high speed [Marwedel, 2003]. Most modern embedded RTOSes are connected to networks. Many systems distribute processing to multiple processes over the network, for example sensor networks. Network processors provide the environment to assist with network establishment.

However, networks can be a source of indeterminism that can cause a process to miss a deadline. For example, when process A sends a message to process B running on the same microprocessor, the CPU, memory and OS are common to both processes. So process A can continue confident that process B will get the message within a given time. While process B may be blocked waiting for the message, it receives it as soon as the OS schedules process B.

By contrast, where process B is running on a separate microprocessor, process A sends the message and continues. A network fault or a higher priority process running on the second processor may result in process B waiting for an undefined period of time. Assuming that the network is functioning correctly may be valid within a robot, but such assumptions become increasingly less valid between robots as their physical separation increases. Adding protocol to the message passing to confirm to process A that process B gets the messages has a significant cost in performance, and increases code complexity.

Chapter 3

RTOSes in a Safe Language

Safety is the primary concern for RTOSes. One strand of RTOS research is looking at the question: can developing an operating system in a safe language result in a system that an errant process cannot crash? Choosing a good programming language can significantly improve the safety of the RTOS. This chapter focuses on safe programming languages in RTOSes. Different real-time programming languages are discussed in this chapter. Some relevant research projects are also discussed.

3.1 The Language Requirements of RTOSes

The requirements of RTOSes call for language features that are not found in many programming languages. 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. If the language does not support low-level features then the language either has to be extended or it has to support calls to assembler routines. The latter is very unsafe. Rather than leaving these features out, Modula-2 places them in a system module, so that the programmer would explicitly recognize that the instructions are unsafe and use them with care.

Low-level features include:

- accessing specific memory locations, such as the address of a hardware input buffer;
- treating the contents of memory as different types, such as loading in bytes from a serial input and then using them as an array of pixels in an image;

- setting bits in a register, such as changing the processor from user to system state;
- changing the return-from-interrupt address, such as an interrupt handler returning to a priority-preemptive scheduler which dispatches a different process to the one interrupted;
- saving and restoring system state including register contents before and after handling an interrupt;
- programming an interrupt handler, so that it is vectored to by the hardware and not called from software, and
- an interrupt handler being able to transfer data to a user process, such as the interrupt handler reading a value and then storing it in a variable known to the process.

These low-level constructs are machine dependent. The problem with them is that they lack the redundancy required by the compiler to check them for consistency with the rest of the program, and the compiler is not able to protect the programmer against errors. Also, the IDE must be able to add the appropriate header and create links to routines in the run-time support software in the embedded system, including its operating system.

Choosing a good programming language can significantly improve the quality of the embedded software. Reliability is the most important feature for real-time systems. Real-time programming languages should include run-time support to minimize run-time errors and to reduce the probability of programming errors.

3.2 What is a Safe Language?

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 [M^{c}Kerrow, ***et. al.,*** 2007] include:

- A safe language minimizes the damage due to programmer error by getting the compiler to handle dangerous functionality. By catching more errors at compile time rather than at run time, it can also increase programmer productivity.
- A safe language is type safe. There is no mixing of types, 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. Cast operations have to be explicit and justified.
- A safe language has assertions to check conformance to design. Asserts can be used to catch incorrect usage of functions. An assertion performs a calculation on input values to confirm that they are in the desired range and type.
- There is no pointer arithmetic in a safe language. The compiler codes all address calculations, such as array indexing. Programming with references rather than with pointer arithmetic stops a program scribbling outside a program's memory area. As a consequence, it eliminates the need for memory management units as protection devices.
- A safe language includes overflow and underflow checking in its run time support, so that buffer writes cannot corrupt code. A common method of attacking the security of an operating system 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 operating system to run out of memory and crash.
- A safe language handles mathematical errors, such as divide by zero, which cause low-level hardware faults, with exceptions.

3.3 Low-level Languages

Machine code defines the capabilities of a processor and is directly executed by it. Instructions are represented with binary numbers that have a one-to-one mapping to a hardware function. As people find lots of numbers difficult to remember, they program in assembly language, which uses mnemonics to represent the machine codes [Burn and

Wellings, 2001]. However, assembly language code is difficult to read and it is easy to produce errors. Therefore, although it is common, the use of assembly language is not encouraged in real-time systems [Gritzalis and Iliadis, 1998].

Programming in assembler is tedious and time consuming even though the programmer has total control over the machine. Programmers soon realized that they were programming the same sequences of machine code over and over again. By replacing these sequences with high-level language constructs they were able to program faster and their programs had less errors as well.

3.4 High-level Languages

High-level languages achieve three purposes:

- They make the code easier for people to read.
- They protect the program from dangerous constructs that human programmers produce both by accident and deliberately. This protection is achieved by the compiler taking over the function.
- They increase programmer productivity. Programming is more enjoyable when you can focus on solving the problem and not be bogged down by run-time errors.

3.4.1 The C Programming Language

The C programming language achieves the above three goals to a limited extent. Firstly, the code is easier to read than assembler. Secondly, the compiler takes over the control of the register set so that the programmer can no longer select which register to use or explicitly change the content of a register. This protects a program against the programmer using one register for two different purposes. Also, in theory, it stops the programmer writing self-modifying code.

C is the language most commonly used in embedded programming. However, it has a number of serious problems that may result in a system crash, some of which are listed below [McKerrow, *et al.*, 2007].

- C code is very difficult to read and understand. It was designed prior to the research into human computer interface and its syntax is very poorly designed. Also, as it was developed before cut-and-paste editors, it has a cryptic syntax, making it easy to type but hard to read. Additionally, it has no concept of graphics.
- C has defined a couple of functions differently to how they are defined in mathematics which causes confusion.
- It has weak typing, which results in programs with numeric errors. By allowing statements that assign a float to an integer a program will truncate the value and give the wrong result.
- Pointer arithmetic allows the code to write anywhere and if the arithmetic is wrong the code will write over other code or data [Holzmann, 2006]. A hardware solution (the memory management unit) was invented to protect against this software problem.
- C does not support some low-level operations that are required to program an operating system. To "overcome" this problem a massive hole was created: in-line assembler. C allows both the system and application programmers to include assembler code in their program, which is extremely dangerous.
- As it has no support for exceptions [Rizk and Halsall, 1987], all errors have to be handled by the return of integers, which results in complex error handling code. These integer values are treated as true and false, because C does not have a Boolean type.
- It has no run-time environment so the programmer has to write all the memory management code. Also, the programmer has to write the code to check for overflow and underflow of common data structures such as arrays.

3.4.2 The Oberon-2 Programming Language

The Oberon-2 programming language was designed to be a highly reliable programming language, featuring strong typing, object orientation, modularization, bounds checking

and garbage collection [Nikitin, 1997]. It is a successor of the Pascal programming language and the Modula-2 programming language, but simpler and safer. Many errors can be detected at compile time rather than run time. Memory leaks are prevented by Oberon's runtime support of memory management. "It compiles with our request for compile-time enforceable static safety and run-time support for dynamic safety, while being well suited for component software development [Brega, 2002]." According to Brega, the Java programming language is at the same level of safety as Oberon-2.

3.4.3 The Java Programming Language

The Java programming language is designed to be safe and robust. A reliable and secure platform is provided for developing an RTOS in Java. The Java programming language has the following features that make it safe and reliable.

- Enforcing strict typing

  Java is a strongly-typed programming language. It enforces strict typing with type conversion functions. The type of every variable and every expression are known at compile time. Casts are trusted in Java because Java's strong typing ensures that every cast is checked at both compile time and runtime. "The Java language is designed to enforce type safety. This means that programs are prevented from accessing memory in inappropriate ways" [McGraw and Felton, 1999].

- Removing pointer arithmetic

  There is no pointer arithmetic in the Java language, which prevents the misuse of pointers. Although pointer arithmetic is a very powerful mechanism in programming, it is also a major source of RTOS crashes. All memory address calculations are handled in the reliable runtime environment. Java programmers must use object references instead of pointers to get access to any memory location. They cannot access memory directly by using pointers.

- Run-time data structure bounds checking

Buffer overflow or underflow is a programming error that may result in a security attack to RTOS. In Java, buffer overflow or underflow errors never happen because data cannot be stored into unallocated memory. Java provides overflow and underflow checking in its run-time support.

- Run-time support of memory management – garbage collection

Java has real-time garbage collection. Memory leaks, which may cause an operating system to run out of memory and crash, are prevented by Java's runtime support of memory management. Using a garbage collector not only eliminates code bugs, but also removes potential security dangers. In Java, the garbage collector also relieves programmers of the burden of performing manual memory management [Venners, 1996].

- Assertions for verifying that data conforms to design

Java has asserts to check conformance to design. An assert performs a calculation on input values to confirm that they are in the desired range and type [Gosling, *et al.*, 1996]. An expected Boolean condition is declared in an assert statement. If the assertion is enabled when the program is running, then the condition is checked at runtime.

- Object Oriented (OO)

Java is an OO language with structured programming of methods. Objects cannot be manipulated directly by programmers, but only through the public interfaces. "Object-orientation and a modern memory model both turn out to have a positive impact on Java security" [McGraw and Felton, 1999].

- Exceptions handling

Exceptions are for handling of errors deep down in a procedure call sequence. Programmers can write a function to define which exception it can raise in Java. Both expected and unexpected errors can be handled by using the exception handling mechanism.

3.4.4 Issues with Using Java

Java was designed to be a safe language and meets the criteria in Section 3.2. Here we will look first at additional issues with Java and then examine how these issues are handled in the RTOSes 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, real-time 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, as 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 points out, in an article on concurrent programming, threads result in non-determinism [Lee, 2006]. 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 do not 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, 1996]. Forgetting a lock may result in corrupted data. Writing data access synchronization is difficult, because circular dependencies must be avoided.

While some developers wish that Java did not 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.

3.5 Low-level Issues of Developing an OS in a Safe High-level Language

There are a number of problems when we are developing an operating system in a safe, high-level language [McKerrow, *et al.*, 2007].

- There are low-level operations that cannot 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 cannot 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 the system state and vectors to an interrupt handling function. This requires a facility to store the 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 should finish 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 cannot 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 separate processors it is much more difficult (and in many designs impossible) to guarantee that deadlines are met.

3.6 Examples of OSes Developed in a Safe Language

3.6.1 XO/2

XO/2 [Brega, *et al.*, 2000] is an object-oriented, hard-real-time system developed at ETH in Zurich to run on PowerPC embedded processors. It is designed for safety, extensibility and abstraction using the Oberon-2 programming language. XO/2 is boot loaded from the host Macintosh and communicates to users via web pages running on the host network.

Brega points out that most embedded systems are written using languages that provide neither static nor dynamic safety. This author summarizes a list of languages classified according to the degree of safety as follows [Brega, 2002].

- Static and Dynamic Safety: Oberon, Oberon-2, garbage-collected versions of Ada, Java, Sather, Component Pascal.
- Dynamic Safety Only: Smalltalk, Lisp.
- Partial Static/Dynamic Safety: Pascal, Modula-2, Ada (using explicit deallocation).
- Unsafe: C, C+.

Oberon is an object-oriented language developed by Nicholas Wirth to follow on from Modula-2 [Oberon, 2007]. Oberon-2 is chosen as the programming language for the XO/2 system since it is statically and dynamically safe [Tomatis, *et al.*, 2003]. Many errors can be detected at compile time rather than run time. Memory leaks are prevented by Oberon's runtime support of memory management.

The design architecture of XO/2 is time-triggered. The CPU time and system resources are pre-allocated, which can lead to a waste of the system time and resources. In the XO/2 system, the developers devised a scheme for approximating worst-case execution time. To approximate a more realistic value of a task's execution time, they use static analysis of the source code combined with task's runtime information that is collected by the performance monitor. Tomatis [Tomatis, ***et. al.***, 2001] claims that this scheme can work well even for the tasks with a lot of dynamic cache usage.

The XO/2 heap manager assigns a type to each allocated object, and a garbage collector is responsible for its reclamation [Tomatis, ***et. al.***, 2001]. This garbage collector provides good performance without any memory requirements at execution time, which is very important when it works in a low-memory condition. The developers claim very fast switching times between processes because the Memory Management Unit (MMU) is only needed for address translation and not for catching program errors.

One of the design principles is the separation of concerns [Tomatis, *et. al.*, 2001]. The XO/2 system is structured in modules. The presence of safe dynamic loading and unloading of compiled units, along with *short edit-compile-run cycles*, is an important precondition for this principle. New modules can be safely tested without threatening the stability of the system.

A static, EDF (Earliest Deadline First) algorithm, with admission testing, is adopted in the priority assignment of XO/2 [Tomatis, *et al.*, 2001]. Tomatis claims that the improved modelling capabilities trade off for the increased processing time. A task is statically assigned a priority according to its deadline. This task will remain its priority, until its normal execution is completed, or when a task with an earlier deadline has been activated by the occurrence of an event. In the XO/2 system, non-real-time tasks are brought to the foreground only when no real-time task is waiting. The non-real-time tasks with the same priority are scheduled in round-robin algorithms, which assign the same time slice to each process.

The XO/2 system has been used for many research projects and commercial products. Brega argues the XO/2 system has successfully implemented the software techniques addressing safety on a system-wide level [Brega, 2002]. Brega points out that the Java programming language has the same level of safety as Oberon-2. This is one of the motivations for us to develop an RTOS in Java.

3.6.2 JX Operating System

The JX operating system is a single address space operating system mainly written in Java [Golm *et al.*, 2002]. Golm and colleagues believe that the features of Java raise the level of abstraction and help to develop more robust systems in less time. Protection in JX is no longer based on MMU, but on the type-safety of the Java byte code instruction set. Therefore, there is no memory space switch caused by inter-process communication and system calls. To expand the address space, MMU support can be added. Typical Java security problems, such as native methods, execution of code of different trustworthiness

in the same thread, and a huge trusted class library, are avoided by JX. The code written in C and assembler is kept to minimal, which makes the system simple and robust.

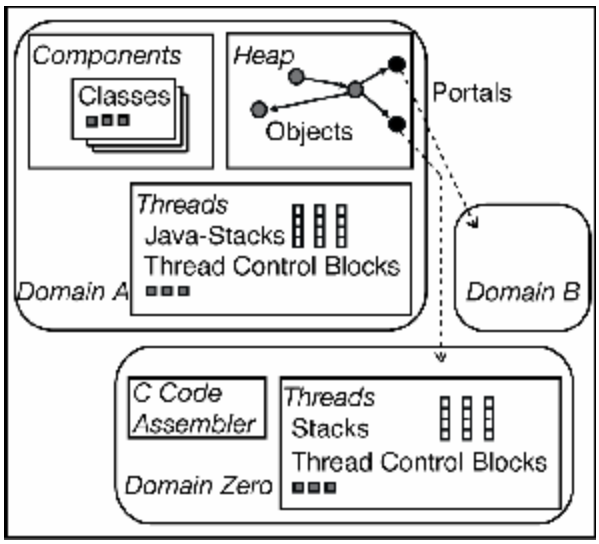

Figure 3.1 JX architecture

The system architecture (Figure 3.1) is comprised of a number of components, which are loaded into domains, executing the JX kernel that is responsible for system initialisation, saving and restoring CPU state and low-level domain management [Golm *et al.*, 2001]. All domains, except Domain Zero, are written in Java. All the native code of JX is stored in Domain Zero. Operating system code is completely isolated from application code, communicating via portals. A domain can only access other domains when it possesses a portal to a service of the other domain. The operations that can be performed with the portal are listed in the portal interface. Each domain is assigned its own heap with it own garbage collector. These garbage collectors can use different algorithms, running independently. Each domain has its own threads, which do not migrate during inter-domain communication. The stacks and thread control blocks are assigned memory from the domain's memory area.

3.6.3 Singularity Operating System

Singularity is an operating system developed by Microsoft Research [Tanenbaum *et al.*, 2006]. It uses advances in programming languages to develop an operating system that an errant process cannot crash [Hunt *et al.*, 2005].

Singularity is programmed in Sing#, a new type-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.

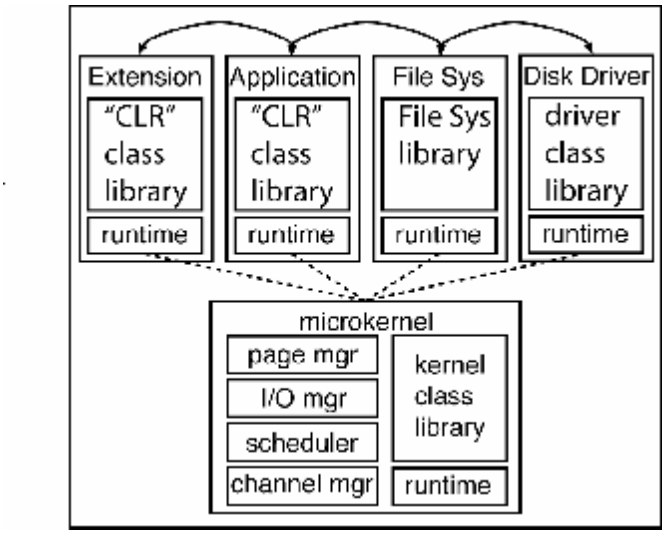

Figure 3.2 Singularity Operating System

Figure 3.2 shows the architecture of the Singularity Operating System. The microkernel provides the core functionality of Singularity, including process creation and termination, channel management, scheduling, I/O management and memory management. Most functionality and extensibility of the system exist in OS processes, not in the microkernel. Singularity is built on an extension model based on Software-Isolated Processes (SIPs) [Hunt *et al.*, 2005]. SIPs are OS processes that provide strong interfaces, failure isolation and information hiding. Singularity uses SIPs for both extensibility and protection.

Chapter 4

Design of JARTOS

JARTOS is designed to be a time-sharing system, where cooperative multiprocessing is used to schedule real-time processes. JARTOS switches processes on a timer interrupt. In this chapter, real-time design issues are discussed. The components of JARTOS are introduced and discussed in detail.

4.1 Real-time Design Issues

RTOSes have to guarantee that processes meet time deadlines. Anything that causes indeterminism in the execution time makes it harder to achieve that guarantee. In the design of the JARTOS system, the real-time design issues considered are discussed in the following sections.

4.1.1 Interrupts

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 service the hardware and then return to the interrupted process, 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.

4.1.2 Scheduling

Priority preemptive 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.

4.1.3 Inter-process Communication

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 without critical sections, because a write cannot 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. This is another reason 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.

4.1.4 Timeout

With respect to 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. Both require human intervention to investigate and fix the problem.

4.1.5 A Safe High-level Language -Java

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 instructions 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 (which is unusual), and should be handled by the exception facility in the language.

As discussed in Section 3.4.3, Java was designed to be a safe language and meets the criteria in Section 3.2. Therefore, the majority code of JARTOS is written in Java language. Assembly code is only allowed in the OS code. There is no calling of C libraries.

4.2 Components of JARTOS

Figure 4.1 depicts the overall architecture of JARTOS. The Operating system (OS) is completely isolated from user applications. Splitting the responsibility in this way results in the application programmer being able to focus on the design and programming of the set of processes required to solve the application problem. The OS part provides the main components of JARTOS, including OS Methods (Table 4.1), OS Processes (Table 4.2), OS Tables (Table 4.3) and Supervisor Calls (Table 4.4).

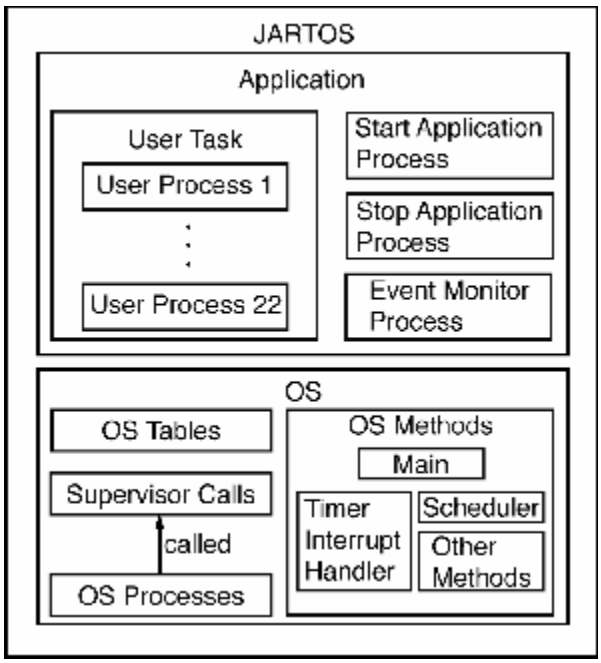

Figure 4.1 Overall architecture of JARTOS

In JARTOS, a few method (Table 4.1) work together to provide the run-time kernel of the OS. Much of the work of the OS and all the work is done by applications. So the task of the OS kernel is to schedule processes. The Main method is called to start the OS kernel by enabling timer interrupts and then calling the scheduler to schedule the first process. Performance probes are placed in the scheduler to measure process performance.

Table 4.1 OS methods

Name	Description
Main	Initializes OS tables and processes, enables timer interrupt, enables

	processes and calls scheduler to start OS
Scheduler	Decides which process is to run and dispatches it
Enable timer interrupt	Enables/disables timer interrupt
Timer interrupt handler	Sets flag to run timer process and handles time out
Performance probe	Collects performance data and places it onto a circular buffer
Process	A method of a process object that performs computation

The OS processes (Table 4.2) do the work of the operating system apart from scheduling. The timer process, which scheduled in response to the timer interrupt, sets flags to tell the scheduler when to run time-triggered processes. The other processes in Table 4.2 handle common OS functionality. Note, the event monitor process is an application process not an OS process because the events are specified to each application.

Table 4.2 OS processes

Name	Description
Timer process	Maintains the timer table and sets flags for processes to run
Message Monitor process	Checks for the arrival of messages
Performance Analysis process	Analyses data collected by performance probes
Timeout Report process	Reports on the timeout of a process
Garbage Collector process	Runs when time available to clean up heap
Idle process	Runs when no other process requires the CPU, enables the event monitor to run (and can simulate the timer interrupt)
Terminate process	Disables timer interrupt and resets tables to stop scheduler

The scheduling of processes and other OS functionality requires a number of tables. These are given in Table 4.3.

Table 4.3 OS tables

Name	Description
OS table	For OS variables
Process table	Dynamic part of process control block (process state)
Scheduler table	For scheduler variables
Memory table	For memory variables
Event table	For event variables

Message table	For message variables
Performance/testing data table	For performance and testing data
Circular Buffer table	For producer/consumer separation of real-time concerns, and for performance analysis
Common Data table	For common data

To request work by the OS, processes execute supervisor calls (Table 4.4). The calls allow a process to start and stop other processes, to communicate with other processes, and to respond to events. By restricting this functionality to supervisor calls, we stop poorly written application code corrupting the OS table.

Table 4.4 OS supervisor calls

Name	Description
Run Process	Sets the execute flag in the scheduler table for a process that has been loaded and enabled, so scheduler will run process
Stop Process	Resets execute flag in the scheduler table
Get Message	Gets a message object – array of ints, floats or string
Send Message	Writes message into message buffer and sets available flag
Receive Message	If there will get message and reset available flag, else will return and set process up to waits for the message, giving up processor
Release Message	Returns message resource to OS
Get circular list	Creates a circular list object
Add to Circular List	Adds values to a circular list
Remove from Circular List	Removes values from a circular list
Write common data value	Writes values to common data area
Read common data value	Reads values from common data area
Wait Event	Waits for an event
Wait	Goes into wait state for n clock ticks – a form of self scheduling
Load Process	Loads a process - set up OS tables using values in process control block
Remove Process	If process is running stops it and clears out OS tables
Enable Process	Add a process to Scheduler table
Disable Process	Removes a process from Scheduler table
Change priority	Changes the priority of user processes by moving them in process table
Simulate event	Switch from hardware event to software event simulator for testing
Get OS Tables	Copy the current value of all OS tables for use in debugging, testing and

	performance measurement
Library of event handlers	A library of event handlers.

Finally, the purpose of the OS is to run applications An application consists of several communicating processes. All applications are started by a Start Application process where responsibility is to set up the processes required to perform the application and schedule at least one to run. The purpose of the Stop Application process is to gracefully shut an application down. The event monitor process polls I/O to check for extend events and sets scheduler flags to start application processes to respond to the events.

Table 4.5 Processes in a typical user application in priority order

Name	Description
Event Monitor process	Polls for I/O event
Application specific process	Application specific code
……	……
Application specific process	Application specific code
Start Application process	Sets up processes to get the application to be run by the OS
Stop Application process	Stops all processes in the current application

The operating system is an instance of an OS class. The JVM, JVM runtime (underlying thread mechanism), and hardware are considered to be the machine (unlike an assembler OS where only the hardware is the machine). JARTOS runs as a process on top of the JVM runtime (Figure 4.2).

Figure 4.2 Runtime environment of JARTOS

Getting the system running is the responsibility of the Main method, which starts the operating system, enables the Start Application process, and calls the scheduler loop. The

scheduler runs the Start Application process, which starts the processes to perform the user application. The Main method (Algorithm 4.1) first declares an object of OS, calling the OS constructor. The constructor is responsible for declaring instances of all tables, initialising all table values to zero, and initialising the OS table. To correctly initialise the timer, the Main method sets the value of current time to the previous time. The Main method loads and enables the Timer process, Garbage Collector process, Timeout Report process, Performance Analysis process and Idle process. These processes are needed for OS house keeping. It also loads and enables the Start Application process, which starts the user application. The Main method sets the Start Application process and Idle process to run. If the Start Application process does nothing then the OS will run servicing timer events by calling the timer process on each clock tick. Then the Main method enables the timer interrupts and calls the scheduler method.

Algorithm 4.1 Main method

1 Declares an object of type OS – calls the OS constructor, which
   Declares instances of all tables
   Initialises all table values to zero
   Initialises the OS table
2 Reads current time and sets to previous time to correctly initialise the timer
3 Loads and enables the following processes, which are considered to be part of the OS: Timer, Garbage Collector, Timeout Report, Performance Analysis and Idle.
4 Loads and enables the Start Application Process: to load, enable and run application processes.
5 Sets the Start Application Process and the Idle process to run
6 Enables timer interrupts
7 Calls the scheduler method, which only returns to Main when terminate process is called. Then Main should exit gracefully.

Figure 4.3 shows the timing control flow of JARTOS. The JARTOS system schedules processes on a timer interrupt, which we have simulated in the Idle process for much of our testing. There is an interrupt handler handling the timer interrupt (either hardware or simulated). It sets the execute flag of Timer process in Scheduler table, and execute the timeout function if any process in the Scheduler table has gone over time.

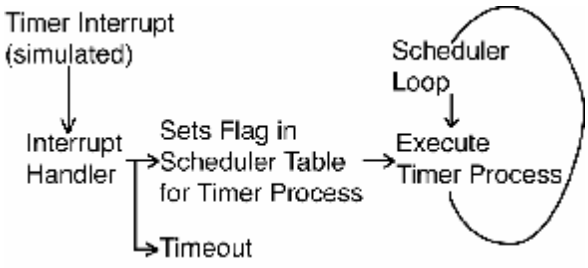

Figure 4.3 Timing flow control in JARTOS

4.3 Scheduler

The scheduler runs at the completion of each process and should run at least every clock tick. It is responsible for giving the CPU to the processes that want it, in priority order. The scheduler loop will only exit when a call to the terminate process resets all execute flags in scheduler table. The scheduler checks the flag of each process in the scheduler table. If the process is ready to run, the scheduler will reset its execute flag to false in the scheduler table. The scheduler sets timeout counter and current process number in OS table. Then the scheduler will start the process and pass state to it. The process executes and returns to the scheduler. If the performance testing flag is set, the scheduler will call performance probe method before and after calling the process.

Algorithm 4.2 Scheduler

i = 0 // set process number to zero – loop invariant *i*<= number of processes
WHILE there is a process to run //last idle process is always ready to run
i = *i* + 1 //check next process
IF process *i* is ready to run //flag in scheduler table is true

```
            Reset process execute flag in scheduler table    //something else has to set it
            Set timeout counter in OS table
            Set current process number in OS table
            IF performance testing flag set in OS table THEN call probe method
            CALL process    //start process and pass state to it
            //process executes and returns to here
        ENDIF
        IF performance testing flag set in OS table THEN call probe method
            i = 0        //return to highest priority process
            Set current process number in OS table //i = 0 is scheduler
        ENDIF
    ENDWHILE
```

NOTES

1. When a process is timed out it is set to return to the scheduler as if it was a normal exit so that the scheduler does not have to clean the timed out process.
2. By doing things (such as timing) with processes, the scheduler is simplified
3. When a process goes into wait, it must call a method to set up the timer table values, then it must set its state values, then it returns to the scheduler.
4. All processes must execute quickly and return to the scheduler. This requires a style of code writing where work is broken up into little bits, for example, a loop may execute one iteration and then return to the scheduler
5. The scheduler is held in an infinite loop by the last process in the scheduler table (idle process) always being enabled to run. A call to the terminate process will reset all enable flags in the scheduler table and the scheduler will return to the Main method, whose task it is to exit gracefully.

4.4 User Process Design

A process object (Table 4.6) contains attributes and methods. Attributes include process control block, constants and variables. Process control block is a static part with initial values, which cannot be updated by OS. Methods includes process constructor, process

method, and private get and set methods. The process method, which must conform to design rules in Section 4.4.1, is called by the Scheduler method.

Table 4.6 A process object

Name		Description
Attributes	Process control block	Static part with initial values (cannot be updated by OS)
	Constants	Process constants
	Variables	Only part of user process that can be changes, local copy of state
Methods	Constructor	Constructs a process
	Process	Called by scheduler – must conform to design rules in Section 4.4.1
	Private get and set methods	Gets and sets process attributes

4.4.1 Process Method Structure

Process execution is intended to be short. Long calculations involving loops should be rolled out so that one iteration is done each time the process is called. This design uses cooperative multiprocessing where a process must release the CPU by returning to the scheduler. If it does not, and timeouts are enabled, it will be timed out and stopped, because a time out is considered to be an error. If timeouts are not enabled it will hang the system.

A process is expected to enable timeouts when it starts and disable them when it ends. Having the process set and reset a timeout flag in the OS table means that there is no need to disable timer interrupts at any time. It solves a difficult critical section problem. Also, it means that trusted (i.e. tested) processes do not have to be protected by timeouts.

Having to rely on the programmer of the process to include the timeout enable/disable is a weakness. To overcome it, we would have to override the method call and return functions with ones modified to perform these operations.

A process can wait on an event, a message or a time period to finish. In each case it will set the wait state and return to the scheduler. When the process executes again it is its responsibility to check why it was called. Hence the ELSE IF structure in algorithm 4.1. The process need only test for those states that it is expecting. For a simple process, there may be no tests, and for many processes there will only be one or two.

A DEBUG flag in the OS table can be used to turn debugging code on and off (a compiler directive is a preferable alternative).

Algorithm 4.3 Default process structure

```
Process name (state)
Enable timeouts for duration of process //set flag in OS table
// so timer interrupt knows it is interrupting a process and not the scheduler
IF waiting on event AND event occurred THEN
    process event
    //may disable event if asynchronous
    IF DEBUG THEN //debug flag in OS Table
        execute debug code
    ENDIF
ELSE  IF want to wait on an event THEN//enable wait for event
    EventNumber = WaitEvent (parameters) //when return to scheduler it will wait
ENDIF
IF waiting on a message AND message received THEN
    ReceiveMessage(message number, message)
    Process message
    //may disable wait for message
ELSE IF want to wait for a message THEN
    MessageNumber = WaitMessage(parameters)
ENDIF
IF waiting on time AND time is up THEN
    Time code
```

```
//may disable wait on time if asynchronous operation
ELSE IF want to wait on time THEN
    Call wait on time
ENDIF
common execution code
exit Disable Timeouts //reset flag in OS table
//timer interrupt returns to here on process time out
RETURN – return to scheduler
```

4.4.2 The Life of a Process

A process is an instance of a process class. The executable method is called by the scheduler. Thus a process is a piece of code that is compiled. In the initial version it will be part of a single Java application that includes the scheduler, etc. Thus, the code downloaded to the embedded system includes everything for an application to run. This is appropriate for small real-time systems that do not have disk drives to load processes from. Extension to include loads of applications is left for a later project.

During its life time a process is (refer to Figure 4.4 and Figure 4.5):

1. Compiled and loaded into memory with the OS, including its process control block
2. Loaded by the load supervisor call – sets up OS tables from process control block
3. Enabled by the Enable supervisor call – adds it to the scheduler table so that it can execute
4. Set by the Run Process supervisor call – sets the run flag in the scheduler table
5. Executed by the scheduler when it is the top priority process and resets its run flag
6. Repeat 4 and 5 until process is Disabled – can also be Stoped or Timed out (error).

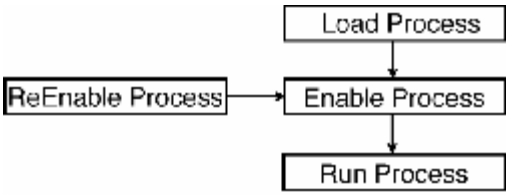

Figure 4.4 Asynchronous flow of control of scheduling a process

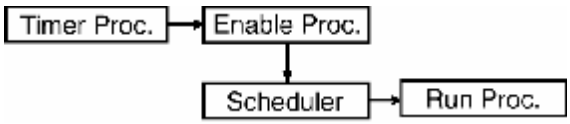

Figure 4.5 Flow of control of scheduling a synchronous process

4.4.3 Process Timing

Synchronous – run every *n* clock ticks

Time – number of clock ticks between executions

Phase – which clock tick relative to first

Asynchronous – called by events – time = 0 = run once

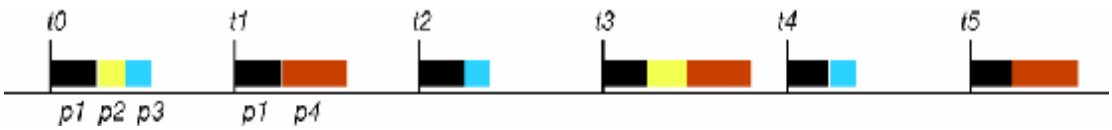

Figure 4.6 Process timing

p1: time = 1, phase = 1

p2: time = 3, phase = 1

p3: time = 2, phase = 1

p4: time = 2, phase = 2

idle: rest of time – OS is checking for events etc.

4.4.4 Events

Events are changes in system state that require the execution of a task to handle them. Events are usually hardware changes such as timer timeout, analog to digital conversion complete, digital input set, and character arrived. In this design, the hardware is expected

to make events easy to detect and handle by the software. However, we realize that the hardware design is not always under the control of the software designer and there are times when interrupts are unavoidable. It is typical in the embedded system world to use cheap hardware at the cost of making the software more difficult to write.

Preferably, events should be levels not edges, and should hang around for a while and not be fleeting. Also, inputs should be buffered so that the software can read them within a given time and does not have to either respond immediately or consume CPU time waiting for them. When a process blocks waiting on an event, we will store the time for testing purposes (a timeout may be added later).

A software event is a software simulation of an event that one process can create to signal to another to run, etc. It is commonly used to in simulations of hardware events during testing. Only one entry should be in the event table for each event and an event should only start one process. If other processes are required to run then they should be started by the process that waited on the event.

Four types of events are to be handled:

1. Level - the level of an input can be detected by polling inputs and reading its value,
2. Edge - a change in an input can be detected either by an interrupt which vectors to an interrupt handler (e.g. timer) or by polling the input and comparing successive values,
3. Handshake – output a level (or pulse) in response to an input event, and
4. Software –a method call that simulates an event.

While polled and interrupt events are enabled and detected in different ways, the response to all events involves the following steps:

1. A process enables the event (and interrupt handler) and sets a process (can be itself) to wait for the event.

2. The event monitor process, or the interrupt handler, sets flags in the event table and process table to tell the OS that the event has occurred and sets the scheduler flag to run the application process that is waiting on the event.
3. The application process executes in response to the event. It clears the event occurred flags. It may also disable the event (and interrupt handler), depending on whether synchronous or asynchronous operation is required. NOTE: when a process is disabled any events that it enabled must be disabled.

Polled Events

An event monitor process is a process in the user application that monitors events by polling hardware inputs. It is called regularly by the scheduler. When it detects an event it sets the execute flag in the scheduler table for the process waiting on the event. Then this process is run by the scheduler. The algorithm for the event monitor is given in Section 4.7.6.

Interrupt Events

A hardware interrupt causes the processor to vector to an interrupt handler. The interrupt handler sets the execute flag of the process that is waiting on the event. It may also read (write) an input (output) value and place it in (get it from) a circular list, a message or common data. The goal is that the hard real-time part is done in the interrupt handler and the soft real-time part is done by the process.

Algorithm 4.4 Interrupt handler

```
IF event enabled in event table THEN
    Read data into common data, circular list or message
    IF process is enabled in scheduler table THEN
        Set flags in event and process tables
        Set execute flag in the scheduler table
    ENDIF
    RETURN from Interrupt
ENDIF
```

Enabling Events

To enable or disable an event, an application process (such as start applications) calls EventNumber = WaitEvent(parameters) and passes in the parameters for the event. The method finds the next available event in the event table and returns the event number to the process. The process should then exit to the scheduler.

Algorithm 4.5 Enabling event

Find the next available entry in the event table.
From the parameters passed in set up the event table entry for this event
IF the process that is to wait on the event is not in the scheduler table THEN
    Enable the process in the scheduler table with execute flag not set
    Set the waiting on event flag and event number in the process table
END
IF the event is detected by an interrupt THEN
    Enable the interrupt – pass in the event number to the handler
Return the event number

Disabling Events is done in reverse order to enabling events.

4.4.5 Inter-process Communication

A crucial feature of a real-time system is the flow of data between processes. Typically, a fast process will read input data and make it available to other processes for calculations etc. Normally, only one process can write a data value while several processes can read it. The write of data does not have to be in a critical section because we are using co-operative scheduling not preemptive scheduling. All data will consist of a value and a time stamp (when the data value was updated). Three mechanisms will be used for inter-process communication, common data, circular buffers and messages (Section 2.2.7).

Common data is a set of variables defined at compile time in a common data object that can be read and written to using public get and set functions. Each data object includes value, type, time updated, and number of updating process. The additional data is useful for testing and debugging. As shown in Figure 4.7, a process that reads sensors may save their values in common data for other processes to read.

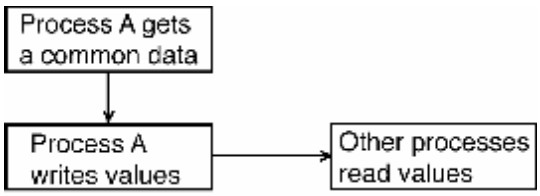

Figure 4.7 Flow of control of common data

Circular buffers are used to separate slower calculation processes from very fast input or output processes. Often used as a way of handling input data that comes in bursts where processing can be done at leisure. Also, useful for output data that has to be synchronized to real-time but can be calculated ahead of time. As shown in Figure 4.8, one process adds to the buffer and a second removes from it. In this design, when the buffer is full, new data overwrites old, so that the most recent n data values are available. A flag is set to indicate that data has been lost.

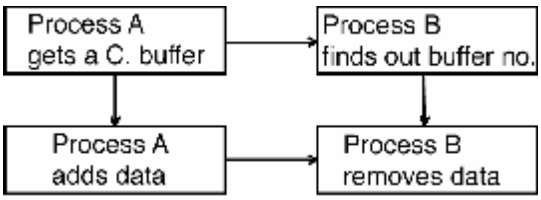

Figure 4.8 Flow of control of circular buffer

Messages require synchronization between processes. They are used to pass data values between processes and the synchronization guarantees that a process does not proceed until it has the latest data values. The messages are objects that are created at run time. We will store the time the process started waiting for use in testing (a timeout may be added later). Inter process communication over the network will be handled with

messages so that data can be shared between multiple processors. These processors could be other Java machines, the host development machine, or web clients displaying data on a web page.

Steps for passing a message (Figure 4.9):

1. Get Message – gets a message object – array of ints, floats or string
2. Send Message – writes message into message buffer and sets available flag
3. Receive Message – if there will get message and reset available flag, else will return and set process up to wait for the message, giving up the processor
4. Release Message – returns message resource to O.S.

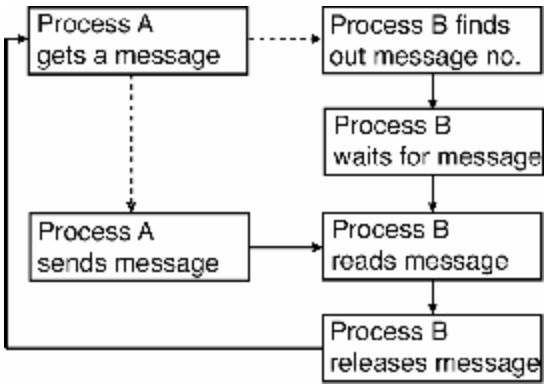

Figure.4.9 Flow of control of passing a message

4.5 Tables Design

4.5.1 Process Control Block

Process control block (Table 4.7) is located in a process object. It contains constants from which the process table is loaded, so it cannot be updated by OS.

Table 4.7 Process control block

Process name	
Synchronous wait time	In ticks, where 0 = run once
Synchronous wait phase	< wait time
Timeout	In ticks – < wait time – 2 is the minimum
Event1	Number of event, 0 = no event

Event2	
Message1	
Message2	
Process exit address	Used to update rti on time out

4.5.2 Configuration Constants

OS methods must check the configuration constants (Table 4.8) for overflow/underflow when adding to or deleting from the tables.

Table 4.8 Configuration constants

Maximum number of processes	e.g. n=32
Size of memory blocks	e.g. S=1K
Number of memory blocks	m * n
Number of memory blocks per process	M
Number of messages	p * n
Number of messages per process	P = 2
Number of events per process	2
Number of timers per process	1
Number of memory blocks for common data	1
Number of circular buffers	4

4.5.3 OS Table

The OS table (Table 4.9) contains the values of all the OS variables. The scheduler sets the timeout counter and current process number in the OS table. If the Enable Performance flag is set, the scheduler will call the performance probe before and after calling the process.

Table 4.9 OS table

Clock	Enable Timeouts flag	Enable DEBUG flag	Timeout counter	Timeout report flag	Number of process timed out
Enable perform	Current Process	Start of Memory	Common data	C lists address	Previous Time

4.5.4 Process Table

The Process table (Table 4.10) contains dynamic values instead of process control block, including process state. Note, the number of entries changes when processes are added and removed so that n is always less than the configuration constant. The OS is process 0.

Table 4.10 Process table

x

Process no	Process Name	Waiting on event	Event number	Event Occurred	Waiting on message	Message number	Message arrived	Waiting on timer	Time is up	Reference to process

y

Process no	1	2	3	4	n-7	n-6	n-5	n-4	n-3	n-2	n-1	n
Process name	Timer	Event Monitor	Start Applicatio	..User 1...	..User last	Message Monitor	Performan ce	Timeout Report	Garbage Collector	Stop Applicatio	Idle	Terminate

4.5.5 Scheduler Table

Processes in the scheduler table (Table 4.11) are in priority order. Setting an execute flag will tell the scheduler to dispatch the process when the CPU is available. When tick count reaches 0, the timer process sets the execute flag in the scheduler table and resets the tick count to *wait time*.

Table 4.11 Scheduler table

Process number	Execute flag	Loaded flag	Wait time	Wait phase	Tick count	Waiting on event

4.5.6 Event Table

The Event table (Table 4.12) contains all the values of events. Setting the *Event occurred* will tell the waiting process to run. There are four types of events: Level, Edge, Handshake and Software.

Table 4.12 Event table

Event number	Event type: Level, edge, handshake, software	Interrupt or polled	Event enabled	Event occurred	Process waiting on event	Reference to event method	Reference to Interrupt handler

4.5.7 Memory Table

The Memory table (Table 4.13) contains all the values of memory. If memory has not been allocated, it could be allocated by process, message, circular list or common data.

Table 4.13 Memory table

Block number	Allocated	Process number	Message number	Circular list number	Common data

4.5.8 Message Table

The Message table (Table 4.14) contains all the values of messages. Setting *Message sent* will tell the To Process that this message has been sent and can be read.

Table 4.14 Message table

Message Number	Allocated	From	To	Waiting for message	Sent	Reference	Type	Transaction number	Sent time	Over Flow flag

4.5.9 Circular Buffer Table

Circular Buffer table (Table 4.15) contains circular buffer values for producer/consumer separation of real-time concerns, and performance analysis. One buffer is permanently allocated to the performance probe

Table 4.15 Circular buffer table

Buffer number	Allocated	Buffer reference	Add process	Remove process	Overflow flag	Add index	Remove index

4.5.10 Common Data Table

Common Data table (Table 4.16) contains a set of variables defined at compile time in a common data object. The additional data is useful for testing and debugging. A process that reads sensors may save their values in common data for other processes to read.

Table 4.16 Common data table

Value	Type	Time Written	Number of Writing Process	Additional Data

4.6 OS Methods

The OS methods contain scheduler, timer interrupt handler, performance probe and enable interrupt. As discussed in section 4.3, the scheduler is responsible for giving the CPU to the processes that want it, in priority order. The scheduler loop will only exit when a call to the terminate process resets all execute flags in scheduler table (Section 4.5.5). We will look at the timer interrupt handler, performance probe and enable interrupt as followed.

4.6.1 Timer Interrupt Handler

The purpose of the timer interrupt handler is to update the clock, set the timer process to run and timeout any process that is taking too long. It is designed to be invisible to the rest of the system except on timeout. Normally there is no interaction with any other

process. It returns the program counter (PC) to where it was prior to hardware servicing interrupt (only exception is timeout)

The timer interrupt handler sets the values in the OS table and handles the timeout operation. The execution time of the timer interrupt handler is kept to be minimum, otherwise it will affect the execution of the processes.

Algorithm 4.6 Timer interrupt handler

```
Save state //implementation dependent e.g. registers used in this code
Increment clock value in OS table //done here rather than in timer process for accuracy
Set timer execute flag in scheduler table //work is done by timer process
IF timeout enabled in OS table THEN // only timeout processes not scheduler
    Decrement timeout counter in OS table
    IF timeout counter is 0 THEN
        Set timeout report flag and process number in OS table
        Enable timeout report process to run in scheduler table
        Clear timed out process's execute flag in scheduler table
        Reset process table to match process control block
        Modify rti on stack to return to the exit point of timed out process
    ENDIF
ENDIF
Restore state
RETURN from interrupt
```

4.6.2 Performance Probe

This probe saves process number and time stamp in a circular list, from where it will be read by the performance analysis process. When full the circular list overwrites itself, so collected data may be lost, it will contain data on the execution of the last few processes.

4.6.3 Enable Interrupt

This process enables/disables interrupts. For example, the timer interrupt so that it calls the timer interrupt handler on each tick. First it sets up the tick time etc.

4.7 OS Processes

As discussed in Section 4.2, getting the system running is the responsibility of the Main method, which starts the operating system, and the Start Application process, which starts the user application. The Main method (Algorithm 4.1) loads and enables the Start Application process and other OS processes, which are needed for OS house keeping. Then it gets the Start Application process and Idle process to run. We will introduce what the OS processes do in this subsection.

4.7.1 Start Application Process

The task of the Start Application process is to set up the application. It will load the processes into the process table, it will enable those processes that are to be executed initially in the scheduler table, and it will set one or more processes to run by setting the execute flag in the scheduler table. It may also set processes to wait on events. If performance analysis is enabled, it sets the performance analysis process to run.

Once started, the application will be performed by the choreography of the synchronous and asynchronous processes that it starts. It loads, enables and sets the application processes to run. When it finishes it returns to the scheduler and the application starts.

This design allows for the possibility of dynamically changing the application processes to suit new operating conditions or to change the application. Based on the current values of data, a process may load, enable and set to run another process. Thus, a different process can be run in response to changes in operating conditions. At a higher level, a process can load, execute and run a different Start Application to set up the system for a new application.

4.7.2 Stop Application Process

This process stops the application by reversing the actions of Start Application and returns the system to the state of just the OS running. First it disables any events that the Start Application process has enabled. It calls Stop, Disable and Remove for each process in the application. It is a low priority process so that it is called when the OS is going to idle. Then it must either start a new process by setting a Start Application process to run or enable Terminate process to run so that next time the system is idle it will terminate the OS.

4.7.3 Terminate Process

This process shuts the OS down by reversing the actions of Main. It disables the timer interrupt and then resets all execute flags in the scheduler table so that the scheduler stops. If log or error (e.g. timeout report) process flags are set, these processes should be run on exit to ensure all debugging information is available on termination.

4.7.4 Idle Process

This process enables the event monitor process to run so that during idle the system is checking for events. This improves response time to events on average. The Idle process must re-enable itself so that the scheduler keeps calling it, else the scheduler would exit and the operating system would stop. Also, the idle process must check that the event monitor process has been enabled to run before setting its run flag in the scheduler table. When testing the OS, to simulate the timer interrupt, simulation code is added to the idle process. This code will simulate a timer interrupt by updating current time and setting the timer process to run.

4.7.5 Timer Process

This process maintains the timer table and sets processes to run.

Algorithm 4.7 Timer process

```
IF current time – previous time <> 1 tick THEN
    log error
```

ENDIF
Previous time = current time
FOR each process in scheduler table DO
Decrement tick count
IF tick count <= 0 THEN
Sets execute flag in scheduler table
Resets tick count to wait time
ENDIF
Resets its execute flag to false //flag set by timer interrupt
END

4.7.6 Event Monitor Process

This process polls for I/O events by calling event methods

Algorithm 4.8 Event Monitor process

FOR each process in event table DO
Execute event method
IF event has occurred THEN
Set process execute flag in scheduler table
ENDIF
ENDFOR

4.7.7 Message Monitor Process

This process checks for the arrival of messages and sets execute flags for the process that is waiting on the message.

Algorithm 4.9 Message Monitor process

FOR each process in message table DO
IF wait flag is set and message has been sent THEN
Set process execute flag in scheduler table
ENDIF

ENDFOR

4.7.8 Garbage Collector Process

This process runs when time available to clean up heap, so it is called by the idle process. The Garbage Collector process has not been designed in detail yet. JARTOS relies on the garbage collection provided by the JVM.

4.7.9 Performance Analysis Process

This process reads the performance data from the circular list and calculates execution times etc. If it is not running, the circular list will contain the last n readings.

The Performance Analysis Process will

1. Allocate Circular Buffer;
2. At a given time call performance probes by setting the performance testing flag;
3. Sets itself to run every *n* msec at priority lower then processes being monitored;
4. Reset performance testing flags at a later time (i.e. after a time interval);
5. Read data from circular buffer and produce analysis trace on each run;
6. Display results to and interact with user.

4.7.10 Timeout Report Process

If there is a timed out process, the timeout report process will print out the detailed timeout information.

4.8 O.S. Supervisor calls

Supervisor calls are methods provided by the OS class, which processes can call to get work done, such as getting and putting values in tables. The first sets are for inter-process communication (Table 4.6.8). The second sets are for event handling (Table 4.6.6).

4.8.1 Get Message

Gets a message object for passing messages between two processes.

Steps for getting a message object:

1. Searches message table to see if message is already allocated (message 1 or message 2). If it is, return the message number and transaction number
2. ELSE search message table for first unused message, simple algorithm – start at last used and use next - roll around at end of list (faster than for loop searching whole table) – if no unused message log error
3. set allocated, from process (this one), to process, message type, transaction number
4. Reset wait flag
5. Get a memory block for the message and make it the correct type
6. Save message number in process table for both processes
7. Return message number

4.8.2 Send Message

Send message to other process and continue.

Steps for sending a message:

1. Get message number and transaction number
2. Copy data into message
3. Increment transaction number
4. Set *message available* flag
5. Set *message sent* flag and time
6. Return

4.8.3 Receive Message

Process asks for message. If it has been sent process reads it and continues. If not process sets message wait and exits.

Algorithm 4.10 Receive message

Get message number and transaction number

IF message sent THEN get *from message*

IF debug/test THEN calculate time for message to transfer

Reset waiting for message flag //done here and not in monitor to avoid race condition

Reset message sent flag in Message table

Reset local have to wait flag

ELSE

Set waiting for message flag in Message Table

Set local have to wait flag

Return (local have to wait)

ENDIF

Process should test local have to wait flag and exit if it has to wait

4.8.4 Release Message

Returns message resource to O.S.

4.8.5 Add to Circular Buffer

One process adds data to a circular buffer. When the circular buffer is full, new data overwrites old, so that the most recent n data values are available. An overflow flag is set to indicate that data has been lost.

Algorithm 4.11 Add to circular buffer

Get add index

Set allocated, buffer reference, time and add process number

IF not overflow THEN

Reset add index & remove index

Reset overflow flag

ENDIF

IF overflow THEN

Reset add index & remove index

Reset overflow flag

ENDIF

4.8.6 Remove from Circular Buffer

One process removes data from a circular buffer. If there is no data in the circular buffer, it will log error information.

Algorithm 4.12 Remove from circular buffer

```
Get remove index
IF Remove Index=0 THEN
    Log error
Return
ENDIF
Set allocated, buffer reference, time and remove process
IF not overflow THEN
    Set remove index
ENDIF
IF overflow THEN
    Set remove index
ENDIF
```

4.8.7 Wait

A form of self scheduling where a process is put into wait state for *n* clock ticks. Set the tick count for the process in the scheduler table for the given process.

4.8.8 Wait Event

Often we want a process to execute when an I/O event has occurred. The method sets up the event table to enable this process to run when the event occurs. Part of the initialization of the operating system is the setting up of the event table. All the events are entered into the table with their disabled flags set.

This method attaches a process to an event and enables the event. It has to be called every time you want the process to wait for the event. The time cost of doing this is

balanced by only events of interest are being monitored. Attaching an event to a process permanently stops other processes using that event. Attaching it for one event means that another process can register for that event at another time.

Algorithm 4.13 Wait event

IF event *n* is not enabled THEN //can wait on that event
    Put process number into event table
    Select event type
    Enable event
ELSE //another process is waiting on this event
    Log error
    Set flag for process to wait and try again
ENDIF
RETURN (event enabled)

4.8.9 Others

There are lots of other processes in Section 4.2 that are still to be designed.

4.9 Library of Event Handlers

These will depend on the I/O devices.

Chapter 5

Code Design of JARTOS

In this chapter, the architecture of TINI is introduced. We provide an overview of how the code of JARTOS fits together. We look at the issues of implementing our system design in Java. Also, the design of testing is introduced. In this chapter, we focus on the general code design of JARTOS. We will discuss the detailed implementation issues in Chapter 6.

5.1 TINI Architecture

A TINI (Tiny Internet Interface) is a microcontroller that runs a Java virtual machine (Figure 5.1). 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 [TINI, 2007].

Figure 5.1 Maxim TINI from Dallas Semiconductor

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 Figure 5.2, the JVM is running under an operating system called the TINI OS.

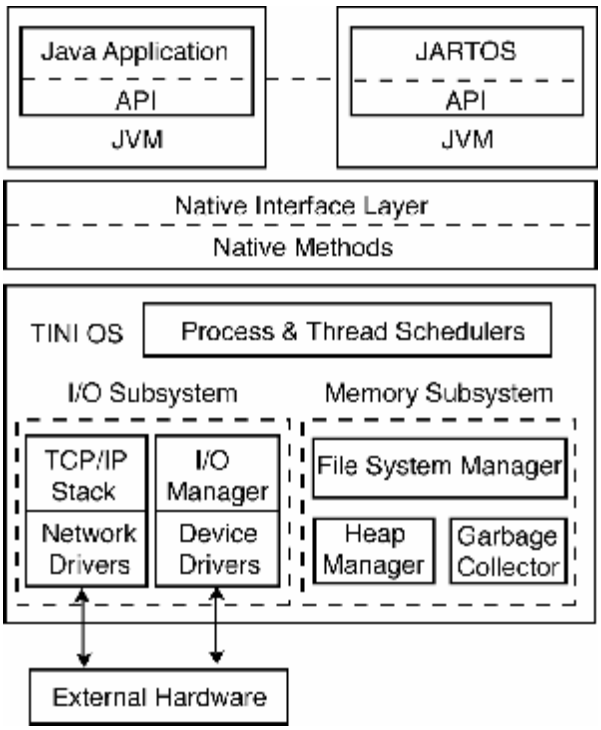

Figure 5.2 Runtime environment of TINI

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 microcontroller 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 very difficult to guarantee that any process running on TINI can meet a real-time deadline. A single process can utilize nearly all CPU on TINI.

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 sit on top of the JVM. The JVM supports the Java API and libraries.

We develop the software for JARTOS in XCode on a Macintosh and then download them into the TINI using the Slush command shell that runs on it. In our research we are running JARTOS as a single application on top of the JVM. We can also run it as an application in Mac OSX. Figure 5.3 shows the overall work flow of executing JARTOS on TINI. The number of application processes is twenty-two (this value can be changed by changing a content in the Process table). The maximum size of the OS.tini file is 512K, since TINI has a limited amount of RAM.

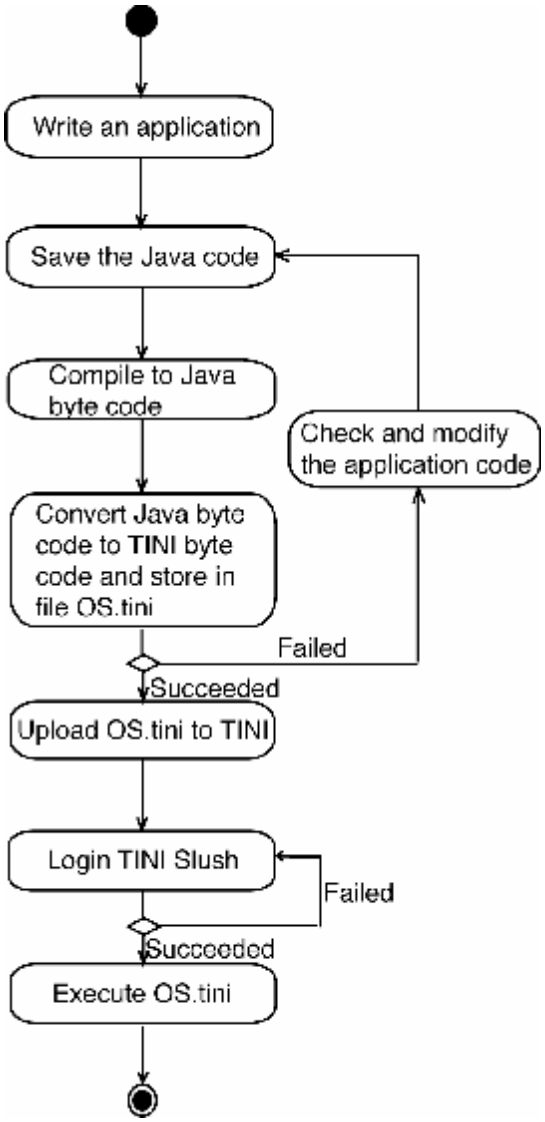

Figure 5.3 Activity diagram of running JARTOS on TINI

TINI provides a class named Clock to access the TINI Real-Time clock in two ways [TINI, 2007]. The faster way is to directly use the values minute, second, hundredth-second, etc. The clock resolution is in hundredth-second called by the getHundredth() method. The other way is to use getTickCount() method returning the long value in milliseconds that have passed since midnight, 1st January, 1970. The getTickCount() method is slower, since the Clock class has to convert the clock values to an amount of milliseconds. Although getTickCount() returns times in millisecond, the clock resolution

is in hundredth-second, not in millisecond. Therefore we decided to adopt the first way instead of calling the getTickCount() method.

5.2 Overview of Code Design

Application code is separated from the OS code, so that:

- the writer of the application only has to write application code and need make no changes to the OS. With this approach, they are better able to focus on programming the real-time task because many of the low-level details are abstracted away by the OS;
- the real-time task is 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;
- the OS can run as a stand alone executable for testing (may be a test application).

Figure 5.4 shows the overall class diagram of JARTOS. There are three java classes in our system: OS class, Process class and Application class. 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. Event Monitor process, Start Application process, Stop Application process and all user application processes are inner classes of the Application class, and inherit from the Process class as well.

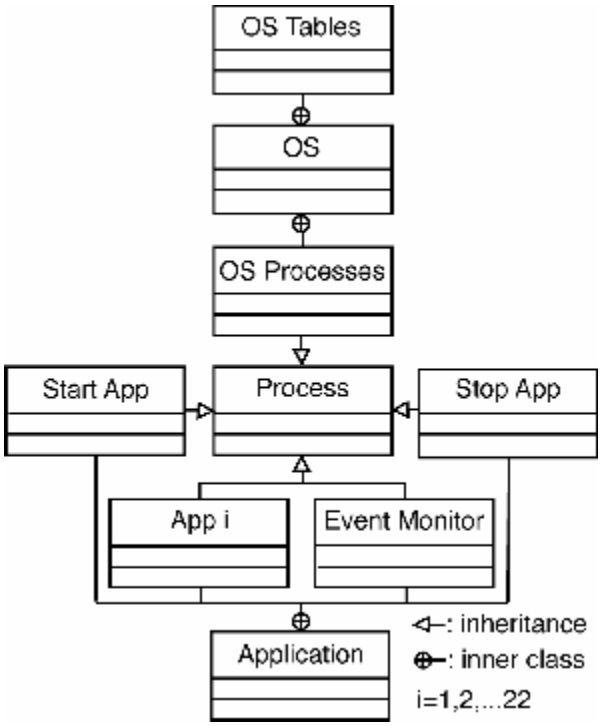

Figure 5.4 Overall class diagram of JARTOS

The Main method in the OS class starts the running of OS. The Main method constructs an instance of the OS class and an instance of the Application class. It constructs the OS processes, and sets their execute flag. Then the Main method enables the timer interrupt and calls the scheduler. The scheduler executes in loop until the Terminate process runs. On the first loop, the scheduler runs OS processes and the Start Application process. Each process is defined by declaring an instance of the Process class, and overriding the process method with their process specific code. The Process class contains a standard template for process methods, and methods for working with processes. The Application class contains the code for a specific task. This code includes Start Application process, application processes to carry out the task and Stop Application process. The amount of application processes is limited to 22 processes. This value can be changed by changing a content in the Process table (Table 4.10).

5.3 Can Java Implement the Design of JARTOS?

The majority of JARTOS is written in Java. Java can implement all high-level functions in the design. The OS tables are stored as Java arrays and accessed by the methods of the relevant table class.

Each process is written as a subclass of the Process class by using the inheritance of Java. Each OS process is constructed in the Main method by declaring an instance of its class. It is set to run according to the design of the JARTOS system. Each user process is constructed in the process method of the StartApplication class. It is loaded, enabled and set to run all in the process method of the StartApplication class. The loadProcess() method is a method of the ProcessTable class. The enableProcess method and the runProcess() method are methods of the Scheduler class.

The schedulerInfiniteLoop() method is a method of the Scheduler class. It is called by the Main method of OS class. The schedulerInfiniteLoop() method calls the OS processes on the first loop, then the StartApplication process enables each user process to run. The scheduler loop algorithm is implemented by a WHILE statement. Every process executes quickly and returns to the scheduler, then the scheduler will run the next process. The number of processes determines the maximum number of iterations of the scheduler loop. The scheduler will call processes using Java Reflection. “Reflection gives your code access to internal information for classes loaded into the JVM and allows you to write code that works with classes selected during execution, not in the source code [Sosnoski, 2003].” One of the ways of using reflection is to invoke a method of a specified name [McCluskey, 1998]. In the Process table (Table 4.10), the last column is called “reference to process method”. So, the scheduler can call the processes by their references.

Inter-process communication and supervisor calls can be implemented in Java as well. Suppose we pass a message from process A to process B. In the processMethod() of the process A class, the getMessage() method is called to get a message object for passing a message between two processes. Then the sendMessage() method is called to write a message and set its available flag in the Message table. In the processMethod() of process

B class, the receiveMessage() method is called to ask for the message. Finally the message resource is returned to OS by calling the releaseMessage() method. The getMessage() method, the sendMessage() method, the receiveMessage() and the releaseMessage() method are all methods of the Message class.

The circular list is read by calling the removeFromCircularList() method. It is written to by calling the addToCircularList() method. Both the removeFromCircularList() method and the addToCircularList() method are the methods of the CircularBuffer class.

A common data value is written by calling the writeCommonDataValue() method of the CommonData class. It is read by calling the readCommonDataValue() method of the CommonData class.

TINI (Section 5.1) supports I/O interfaces within its run-time environment including: Serial (RS232/485), SPI™, Parallel, I^2C*, 1-Wire and CAN. Dallas Semiconductor provides several TINI classes to assist with I/O access [TINI, 2007].

5.4 Low-level Issues

The timer interrupt handler should be connected to the hardware interrupt of TINI. In initial testing of JARTOS, the timer interrupt is simulated in the Idle process. The interrupt updates the clock value every tick.

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 called timerInterruptHandler(). 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. 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 JARTOS can take the processor away from that process.

As this type of operation is potentially dangerous and can cause failure to meet deadlines, the only time an interrupt handler is allowed to return to a different address is in a time out. When a time out occurs, the interrupt returns to the exit address of the interrupted process so that it returns to the scheduler in the normal way. These low-level functions in the timerInterruptHandler() method cannot be written in Java. TINI provides TNI (TINI Native Interface) for programmers to call native code in Java code. Therefore, we implement low-level functions in assembly language supported in TNI.

In TINI Native API, there is a function named System_SaveJavaThreadState used to save the Java state for the current thread. We have a native method called Native_SaveState() to call this function. When an interrupt occurs, the Native_SaveState() method will be called by the timerInterruptHandler() method to save the state of current running process.

There is a function named System_RestoreJavaThreadState used to restore the Java state for the current thread. We have a native method called Native_RestoreState() to call this function. When the interrupt handler completes servicing the interrupt, the Native_RestoreState() method will be called by the timerInterruptHandler() method to restore the state and continue the thread of execution of the current process.

We are still looking for the way to connect the timer interrupt handler to the hardware interrupt and change the return address to the exit address of the timed out process. We may write them in assembly language and save them as native methods called by the timerInterruptHandler() method.

Note:
During the period of time when the thesis was being examined, we designed ways to solve each of the low-level issues with Java language features. Specially, we found solutions to the timer interrupt handler and to the problem of cleanly killing a process in response to a timeout.

The timer interrupt is programmed as a single background thread by using the java.util.Timer and java.util.TimerTask classes. The timer interrupt is scheduled as a TimerTask object for repeated execution at regular intervals by a Timer.

It is possible for one method to stop/kill the execution of another in Java when that method is packaged in a Thread. So, our implementation would involve three threads: Main thread, the Timer interrupt thread and the Process thread. The design of each of these threads follows.

Timer Interrupt Thread (as designed, now running as a thread)
- handle timer interrupt
- IF timeout THEN set flags, etc., stop Process thread

Process Thread
- the process method becomes the run method of a thread
- as now it terminates, so when the method terminates the thread dies
- another thread can kill it by calling the interrupt method

Main Thread (i.e. current main)
- initialise as now
- Scheduler loop
    - timing measurement etc. as now
    - instead of call to process method, start Process Thread to call process's run method
    - join Process Thread (Main thread waits until Process Thread completes)
    - Main thread resumes after Process thread dies
    - rest of main as now – i.e. back around scheduler loop

5.5 Design of Testing

5.5.1 Test Harness

Each class of JARTOS OS has a test harness. A test harness is a program for a class that calls every function with test inputs, and then compares the outputs of the function

to those expected for the given test inputs. Test harnesses overcome a common problem. Often, the programmer thoroughly tests the code when it is written. But when it is modified, the programmer often does not run the same tests, so the quality of the code is reduced every time it is updated because the programmer does not run all the tests previously run. A test harness guarantees that all tests are run every time the code is updated. While test harnesses take time to write, they are easy to maintain and extend. These test harnesses are kept up to date and documented so that they can be run every time a change is made to a class, then we can say that the class passes a given set of tests.

5.5.2 Test Application

The JARTOS system has a set of the test applications (Appendix C) that are documented, extended and run every time any part of the OS is changed. These test applications carry out the following tests.

1. Test that the OS runs, tests the scheduler, and runs a single application process that prints out the contents of all OS tables.
2. Test each OS process (timer, etc) that they give expected results
3. Test timeout (We cannot test timeout now, since we have not worked out the timeout function.)
4. Test performance probes
5. Test multiple processes, including process to print out tables
6. Test Performance Analysis process

5.5.3 Assertions

The system has a set of assertions for checking conformance to design set out in the earlier phase of the development [Bartezko, 2001]. The purpose of assertions is to catch incorrect usage of functions, not to debug code. It is very helpful to program with assertions, as they are self documenting. We write assertions in any method with the IF-ELSE statement, the SWITCH statement and the FOR statement. Figure 5.5 shows the code of the Timer process. The Timer process is to operate for each process in the Scheduler table. The maximum process number is 32 (this value can be changed by

changing a content in the Process table) in the Scheduler table (Table 4.11). We place an assertion at line 3380 to check the value of "i"(assert i<33), which is the process number.

```
3364    class Timer extends Process{
3365        public Timer(String procName, int synWaitTime, int synWaitPhase,
3366            int timeout, int event1, int event2, int message1,
3367            int message2, long processExitAddress){
3368            super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
3369            message1, message2, processExitAddress);
3370        }
3371        public void processMethod(){
3372            //IF current time-previous time <> 1 tick THEN log error END
3373            if((getCurrentTime()-getPreviousTime())/20!=1){
3374                System.out.println("error occured:current time-previous time<>1");
3375            }
3376            //Previous time = current time
3377            refToOS.setPreviousTime(refToOS.getCurrentTime()); //Previous time=current time
3378            //for each process in scheduler table
3379            for(int i=1;i<refToScheduler.getProcessNumberInSchedulerTable();i++){
3380                assert i<33;//assert the process number is less than 33
3381                int tickCount=refToScheduler.getTickCount(i);
3382                //Decrement tick count
3383                refToScheduler.setTickCount(i,--tickCount);
3384                //IF tick count <= 0 THEN
3385                if(refToScheduler.getTickCount(i)<=0&refToScheduler.getProcessNumber(i)!=0){
3386                    //Set execute flag in scheduler table
3387                    refToScheduler.setExecuteFlag(i,true);
3388                    //reset tick count to wait time
3389                    refToScheduler.setTickCount(i,refToScheduler.getWaitTime(i));
3390                }
3391            }
3392        }
3393    }
```

Figure 5.5 Example code with assertion

Chapter 6

Code Implementation

In this chapter, we describe how to write OS tables and processes in Java. We introduce how to avoid repetitive object creation in code implementation. Also, we discuss how to write a test harness for each class in JARTOS.

6.1 Classes

All classes in JARTOS are declared with the public modifier, so that we can pass objects by its reference. All the fields are declared with the private modifier, making each field accessible only within its own class. We have set and get methods with the public modifier to write and read the fields. Supervisor calls and system methods are stored in relevant classes. Detailed descriptions of each class and methods are listed in Appendix A.

6.1.1 OS Tables

All the attributes of the OS tables are stored in arrays, so we can call any value of tables directly within the OS. Figure 6.1 shows the declaration code for the Process table.

```
518         int size=32;
519         int[] PROCESS_NUMBER=new int[size];
520         String[] PROCESS_NAME=new String[size];
521         boolean[] WAITING_ON_EVENT=new boolean[size];
522         int[] EVENT_NUMBER=new int[size];
523         boolean[] EVENT_OCCURRED=new boolean[size];
524         boolean[] WAITING_ON_MESSAGE=new boolean[size];
525         int[] MESSAGE_NUMBER=new int[size];
526         boolean[] MESSAGE_ARRIVED=new boolean[size];
527         boolean[] WAITING_ON_TIME=new boolean[size];
528         boolean[] TIME_IS_UP=new boolean[size];
529         String[] REFERENCE_TO_PROCESS_METHOD=new String[size];
```

Figure 6.1 The declaration code of Process table

6.1.2 Processes

The Main method is responsible for enabling the Start Application process to be run by the scheduler (Figure 6.2) to enable the user application processes to be run as required.

```
175     class StartApplication extends Process{
176         public StartApplication(String procName, int synWaitTime, int synWaitPhase,
177         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
178             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
179             message1, message2, processExitAddress);
180         }
181         public void processMethod(){
182             Application1 App1 = new Application1("Application1",1,1,2,0,0,0,0,0);
183             refToOS.passRef(App1);
184             passRef(App1);
185             refToProcessTable.loadProcess(App1);
186             refToScheduler.enableProcess(App1);
187             refToScheduler.runProcess(App1);
188         }
189     }
```

Figure 6.2 Code of Start Application process

```
268     class Application1 extends Process{
269         public Application1(String procName, int synWaitTime, int synWaitPhase,
270         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
271             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
272             message1, message2, processExitAddress);
273         }
274         public void processMethod(){
275             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
276             //insert your code here
277
278             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
279         }
280     }
```

Figure 6.3 Code of sample Application process

There are 2 steps to write and run an application process:

1. As shown in Figure 6.3, the code of an application process is written in the processMethod() of the application process class which inherits from Process class. The processMethod() method of the application process class overrides the processMethod() method of the Process class with the specific code.
2. We construct an instance of the application class in the processMethod() of the StartApplication class (Figure 6.2), then we call loadProcess(), enableProcess() and runProcess to set the application process to run.

6.2 Passing Object 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. We declare a reference to an object and define a method to pass the object reference;
2. After constructing the object, we call the reference passing method;
3. We call the method by the object reference.

For example, we need to call getClock() and setEnablePerformProbes() of OS class in the processMethod() of the PerformanceAnalysis class. Firstly, we declare a reference named refToOS to OS object and define a method called passRef() to pass the object reference (Figure 6.4).

```
10      OS refToOS;
11      public void passRef(OS myOS){
12          refToOS=myOS;
13      }
```

Figure 6.4 Example code of passing object by reference (1)

Secondly, we construct an instance of OS class, and then we call the reference passing method (Figure 6.5).

```
3602            OS myOS=new OS();
3603            myOS.passRef(myOS);
```

Figure 6.5 Example code of passing object by reference (2)

Thirdly, we call the methods, getClock() and setEnablePerformProbes(), by the object reference (Figure 6.6).

```
3449            if(refToOS.getClock()%20==0){
3450                refToOS.setEnablePerformProbes(false);
3451            }
```

Figure 6.6 Example code of passing object by reference (3)

6.3 Scheduling Processes

The scheduler is supposed to call processes using Java reflection. When we wrote the code for the scheduler, we found that Java reflection is not supported in the runtime environment of TINI. So, we had to find another way for the scheduler to call processes. The scheduler calls the processMethod() of each process, based on the process number, using the switch statement. There are some limitations for the scheduler to call processes. Every time we change the number of application processes we have to change the code in the switch statement. Also, it costs the performance of JARTOS. Figure 6.7 shows part of the scheduler code. We expect that Java reflection is supported in the future version of TINI runtime environment. Then we can use the Java reflection instead of the switch statement.

```
2332                switch (i) {//i is process number
2333                    case 1:  refToTimer.processMethod(); break;
2334                    case 2:  refToEventMonitor.processMethod(); break;
2335                    case 3:  refToStartApplication.processMethod(); break;
2336                    case 4:  refToApplication1.processMethod(); break;
2337                    ......
2338                }
```

Figure 6.7 Part of the switch code in the scheduler

6.4 Low-level Issues

Due to the time and documentation limitation, we have not solved the low-level issues of JARTOS (Section 5.4). The timer interrupt is simulated in the Idle process, updating the clock value every tick. The timeout function has not been implemented. We will try to solve them in future work.

6.5 Test Harnesses

All the test harnesses are written in the OS class as public methods with no return type, and are called in the Main method of the OS class. We wrote a test harness as a method

named testClassName() for each class of our system. In each test harness (Algorithm 6.1), we call each method of a certain class with test inputs. If the output is equal to the expected output, it will display the correct information. Otherwise, it will display the error information.

Algorithm 6.1 Test harness

IF Output=Expected Output THEN

    Correct

ELSE

    Error

END

```
3246      public void testCommonData(){
3247          refToCommonData.setupCommonData(0,"test",0,0,0);
3248          //test setupCommonData()
3249          if(refToCommonData.getValue()==0&refToCommonData.getType()=="test"
3250          &refToCommonData.getTime()==0&refToCommonData.getProcessNumber()==0
3251          &refToCommonData.getAdditional()==0){
3252              System.out.println("setupCommonDataValue():correct!");
3253          }
3254          else{
3255              System.out.println("writeCommonDataValue():error!");
3256          }
3257
3258          refToCommonData.writeCommonDataValue(1,"test",2,3);
3259          //test writeCommonData()
3260          if(refToCommonData.readCommonDataValue()==1){
3261              System.out.println("writeCommonDataValue():correct!");
3262          }
3263          else{
3264              System.out.println("writeCommonDataValue():error!");
3265          }
3266      }
```

Figure 6.8 A test harness for the CommonData class

Figure 6.8 shows a section of code from the test harness for the CommonData class. It shows tests for the setupCommonData() and writeCommonData() methods. The tests in the IF-ELSE statements encode the correct values to compare the results of the methods to.

Chapter 7

Performance Measurement

RTOSes have to guarantee that real-time processes execute within specified deadlines. Loss of synchronization and disruptions in control can occur when deadlines are not met. Timing problems are often very difficult to find. In JARTOS the decision to use polling and an event monitor rather than interrupts, and cooperative multiprocessing rather than preemptive multiprocessing ensures that an application does not loose control of process execution but it may introduce timing problems. In this chapter, we introduce a set of performance measurements to investigate the timing problems. These performance measurements are carefully designed to provide the right information at minimal cost in performance. Performance of TINI and JARTOS are measured and discussed.

7.1 Performance Measurement of Java Instructions Running on TINI

Before measuring the performance of JARTOS, we wrote three TINI applications to measure the performance of Java instructions running on TINI, since JARTOS runs on top of the JVM of TINI.

1. The getHundredth() method (Section 5.1), which is a method provided in TINI library, is used to access the real-time clock of TINI. We wrote a TINI application to measure the performance of getHundredth() reading the clock.
2. We need to obtain the execution time of a WHILE loop and a fundamental instruction unit, which can give us an idea of the performance that we can expect. So we wrote a TINI application with a WHILE loop to measure them.
3. The most common method of debugging is to add System.out.println(message) call to output a debugging message on the console. However, the time consumed

can seriously impact the performance of the RTOS. So we conducted tests to measure the time of this function call.

7.1.1 Testing the getHundredth()

We wrote a linear program (Figure 7.1) that calls getHundredth() 100 times and stores the values in an array. Then the TINI application prints the array out.

```
1  import com.dalsemi.system.*;
2  public class newProj {
3      public static void main (String args[]) {
4          com.dalsemi.system.Clock c=new com.dalsemi.system.Clock();
5          int[] hundredth=new int[1000];
6          c.getRTC();
7          hundredth[0]=c.getHundredth();
8          ...
9          c.getRTC();
10         hundredth[999]=c.getHundredth();
11         for(int i=0;i<1000;i++){
12             System.out.println(hundredth[i]);
13         }
14     }
15 }
16
```

Figure 7.1 A TINI application testing getHundredth()

The following data sequence is part of the raw test data that we collected:

…51,52,53,54,54,55,56,57,58,59,60,60…

The values are in hundredths of milliseconds, so we calculated the average over the 100 readings to get a time for the 2 function calls (c.getRTC() and c.getHundredth()). From the above data, we calculated that it takes 8 milliseconds to read the clock. This time is much longer than we expected, which alerted us to the fact that the performance of the TINI is poor. So in the next section we set out to obtain the execution time of a fundamental instruction unit, to give us an idea of the performance that we can expect.

7.1.2 Testing the WHILE loop

As shown in Figure 7.2, we wrote a TINI application (Algorithm 7.1) that reads the clock before and after a WHILE loop. There is only one instruction "i=i+1" in the WHILE loop, which makes the WHILE loop run. The application was tested with different WHILE loop execution counts (100 times, 1000 times and 10000 times respectively).

Algorithm 7.1 Testing WHILE loop on TINI

Reads clock

WHILE i<100 (1000) (10,000)

i=i+1

ENDWHILE

Reads clock

```
import com.dalsemi.system.*;
public class newProj {
    public static void main (String args[]) {
        com.dalsemi.system.Clock c=new com.dalsemi.system.Clock();
        int i=0;
        c.getRTC();
        int startTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        while(i<100){
            i=i+1;
        }
        c.getRTC();
        int finishTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        int executionTime=(finishTime-startTime);
        System.out.println("execution time:"+executionTime);
    }
}
```

Figure 7.2 A TINI application testing a WHILE loop

The raw test data we collected is listed in Table 7.1. As shown in Figure 7.3, the measurements do not match the line that we would expect based on the measurement of 100 loops.

Table 7.1 Result of testing WHILE loop on TINI without correction

Loops	Actual Time/msec
100	20
1000	140
10000	1360

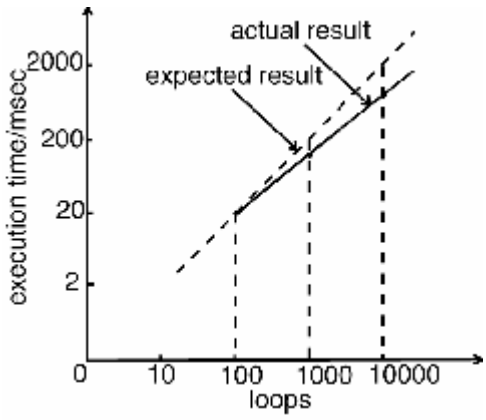

Figure 7.3 Result of testing WHILE loop without correction

We found that the instructions calculating the start time (code line 7 in Figure 7.2) costs the extra time. In order to get more accurate test data, we write a linear program (Figure 7.4) to calculate the time it takes, which is 7 msec (time=(s10-s1)/10≈7msec).

```
1  import com.dalsemi.system.*;
2  public class newProj {
3      public static void main (String args[]) {
4          com.dalsemi.system.Clock c=new com.dalsemi.system.Clock();
5          c.getRTC();
6          int s1=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
7          c.getRTC();
8          int s2=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
9          c.getRTC();
10         int s3=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
11         c.getRTC();
12         int s4=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
13         c.getRTC();
14         int s5=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
15         c.getRTC();
16         int s6=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
17         c.getRTC();
18         int s7=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
19         c.getRTC();
20         int s8=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
21         c.getRTC();
22         int s9=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
23         c.getRTC();
24         int s10=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
25         System.out.println(s1);
26         System.out.println(s10);
27     }
28 }
```

Figure 7.4 A TINI application testing the calculation time

Table 7.2 lists the new test data after considering the execution time of the time to calculate the time. As shown in Figure 7.5, the data is fairly close to the data line we expected. From the data in Table 7.2, we calculated that it takes 0.13 msec to execute a WHILE loop on TINI.

Table 7.2 Result of testing WHILE loop on TINI with correction

Loops	Actual time/msec	Expected time/msec
100	20-7=13	
1000	140-7=133	130
10,000	1360-7=1353	1300

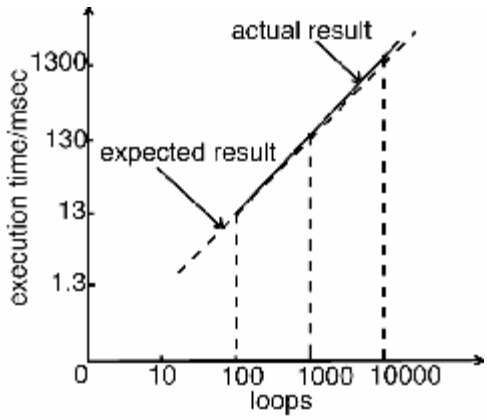

Figure 7.5 Result of testing WHILE loop on TINI with correction

We have defined "j=j+1" as a fundamental instruction unit in Java. The performance of a CPU is often defined as the execution time of a register to register add. We consider adding 1 to a variable to be a similar measure for Java. This measurement will provide a simple comparison when porting JARTOS to another embedded system (such as the Sun SPOT when they are available for purchase). Then, we can use it to scale all the other performance measurements reported here to predict the performance of JARTOS on the new hardware.

As shown in Figure 7.6, we added the instruction "j=j+1" into the WHILE loop (Algorithm 7.2). We set the loop execution counts to 10000 times, since the execution

time is much less than one hundredth of a second. We measured an execution time of 1990msec. Then we calculated the execution time of "j=j+1". It takes 0.063msec (time = (1990-1360)/10000 = 0.063msec).

Algorithm 7.2 Testing the fundamental instruction unit on TINI

Read clock

WHILE i<10000

i=i+1

j=j+1

ENDWHILE

Read clock

```
import com.dalsemi.system.*;
public class newProj {
    public static void main (String args[]) {
        com.dalsemi.system.Clock c=new com.dalsemi.system.Clock();
        int i=0;
        int j=0;
        c.getRTC();
        int startTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        while(i<100){
            i=i+1;
            j=j+1;
        }
        c.getRTC();
        int finishTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        int executionTime=(finishTime-startTime);
        System.out.println("execution time:"+executionTime);
    }
}
```

Figure 7.6 Testing the fundamental instruction unit on TINI

7.1.3 Testing the System.out.println()

In performance testing, we often use the function "System.out.println()" to display test information. Its execution affects the accuracy of the testing result, since it costs time. So we wrote a linear program (Figure 7.7) to test the execution time of "System.out.println()".

```
import com.dalsemi.system.*;
public class newProj {
    public static void main (String args[]) {
        com.dalsemi.system.Clock c=new com.dalsemi.system.Clock();
        int i=0;
        int j=0;
        c.getRTC();
        int startTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
        c.getRTC();
        int finishTime=((c.getHour())*360000+(c.getMinute())*6000+(c.getSecond())*100+c.getHundredth());
        int executionTime=(finishTime-startTime);
        System.out.println("time, execution time:"+executionTime);
    }
}
```

Figure 7.7 Testing System.out.println()

The result of the above program is 50msec. It takes 7msec to read the clock (Section 7.1.2). So, it actually takes 43msec to execute 10 instructions of "System.out.println()". That is 4.3msec to execute a "System.out.println()" instruction. We also tested the instruction "System.out.println()" printing out 20 characters, which is "System.out.println("we are testing print")". It takes 5.3msec to execute "System.out.println()" printing out 20 characters, i.e. an additional 0.05msec per character.

Table 7.3 Testing result of running Java instructions on TINI

Test Name	Test Instruction	Test Result
TINI method 1	getHundredth()	Takes 8msec to read the clock
TINI method 2	Get time	Takes 7msec to get the current time
WHILE loop	while	Takes 0.13msec to run a WHILE loop
Fundamental instruction unit	j=j+1	Takes 0.063msec to execute
Print	System.out.println()	Takes 4.3msec to execute
Print	System.out.println(within 20 characters)	Takes 5.3msec to execute

Table 7.3 lists the test result of running Java instructions on TINI. The developers of TINI claims that each thread is assigned an 8 msec time slice on TINI [TINI, 2007]. The instruction "j=j+1" takes 0.063 msec on TINI. That is 126 fundamental instructions per slice, which means it will not do much in any slice. So we think the time slice used in TINI OS should be longer.

7.2 Impact of JARTOS on Performance of Java instructions

After measuring the performance of Java instructions running on TINI, we measured the performance of Java instructions running on JARTOS, since we wanted to check whether JARTOS has any impact on the performance of Java instructions.

7.2.1 Testing the WHILE loop

```
309     class Application3 extends Process{
310         public Application3(String procName, int synWaitTime, int synWaitPhase,
311         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
312             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
313             message1, message2, processExitAddress);
314         }
315         BitPort bp = new BitPort(BitPort.Port3Bit5);
316         public void processMethod(){
317             System.out.println("test process is running!");
318             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
319             int i=0;
320             int startTime=refToOS.getCurrentTime();
321             while(i<100){
322                 i=i+1;
323             }
324             int finishTime=refToOS.getCurrentTime();
325             int executionTime=(finishTime-startTime);
326             System.out.println("execution time:"+executionTime);
327             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
328         }
329     }
```

Figure 7.8 A Test process testing the WHILE loop on JARTOS

We wrote a Test process (Figure 7.8) running a WHILE loop (Algorithm 7.1). The WHILE loop also runs 100 times, 1000 times and 10000 times respectively. As shown in Table 7.4, the test results are the same as the test result in Table 7.1.

Table 7.4 Results of testing the WHILE loop on JARTOS without correction

Loops	Actual Time/msec
100	20
1000	140
10000	1360

Then we tested the execution time of calling getCurrentTime() on JARTOS, which is a method getting the current time. We wrote a linear program in the Test process, which calls getCurrentTime() 10 times (Figure 7.9). Then we calculated the time from the test data that we collected. It takes 7msec (time = (s10-s1)/10 ≈ 7msec). Again, there has been no change.

```
309     class Application3 extends Process{
310         public Application3(String procName, int synWaitTime, int synWaitPhase,
311         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
312             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
313             message1, message2, processExitAddress);
314         }
315         BitPort bp = new BitPort(BitPort.Port3Bit5);
316         public void processMethod(){
317             System.out.println("test process is running!");
318             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
319             int s1=refToOS.getCurrentTime();
320             int s2=refToOS.getCurrentTime();
321             int s3=refToOS.getCurrentTime();
322             int s4=refToOS.getCurrentTime();
323             int s5=refToOS.getCurrentTime();
324             int s6=refToOS.getCurrentTime();
325             int s7=refToOS.getCurrentTime();
326             int s8=refToOS.getCurrentTime();
327             int s9=refToOS.getCurrentTime();
328             int s10=refToOS.getCurrentTime();
329             System.out.println(s1);
330             System.out.println(s10);
331             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
332         }
333     }
```

Figure 7.9 Testing the execution time of calling getCurrentTime() on JARTOS

Table 7.5 shows the corrected test result of running WHILE loops on JARTOS. As shown in Figure 7.10, the data is fairly close to what we expected. From the test data of Table 7.5, we calculated the execution time of running a WHILE loop on JARTOS. It takes 0.13msec, which is the same as that in Table 7.2.

Table 7.5 Result of testing WHILE loop on JARTOS with correction

Loops	Actual time/msec	Expected time/msec
100	20-7=13	
1000	140-7=133	130
10,000	1350-7=1343	1300

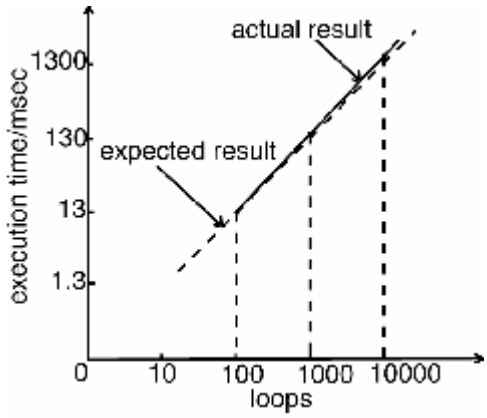

Figure 7.10 Result of testing WHILE loop on JARTOS

Then we also wrote the instruction "j=j+1" in the WHILE loop (Figure 7.11) to obtain the execution time of a fundamental instruction unit on JARTOS. It takes 0.063msec, which is the same as we measured previously (Section 7.1.2).

```
309     class Application3 extends Process{
310         public Application3(String procName, int synWaitTime, int synWaitPhase,
311         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
312             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
313             message1, message2, processExitAddress);
314         }
315         BitPort bp = new BitPort(BitPort.Port3Bit5);
316         public void processMethod(){
317             System.out.println("test process is running!");
318             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
319             int i=0;
320             int j=0;
321             int startTime=refToOS.getCurrentTime();
322             while(i<100){
323                 i=i+1;
324                 j=j+1;
325             }
326             int finishTime=refToOS.getCurrentTime();
327             int executionTime=(finishTime-startTime);
328             System.out.println("execution time:"+executionTime);
329             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
330         }
331     }
```

Figure 7.11 Testing the fundamental instruction unit on JARTOS

7.2.3 Testing the System.out.println()

We wrote a Test process (Figure 7.12) that execute "System.out.println()" 10 times, which takes 50msec. The execution time of calling getCurrentTime() is 7msec. So it takes 43msec to execute 10 instructions of "System.out.println()". That is 4.3msec to execute a "System.out.println()" instruction, which is same as running on TINI.

```
272     class Application3 extends Process{
273         public Application3(String procName, int synWaitTime, int synWaitPhase,
274         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
275             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
276             message1, message2, processExitAddress);
277         }
278         public void processMethod(){
279             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
280             int s1=refToOS.getCurrentTime();
281             System.out.println();
282             System.out.println();
283             System.out.println();
284             System.out.println();
285             System.out.println();
286             System.out.println();
287             System.out.println();
288             System.out.println();
289             System.out.println();
290             System.out.println();
291             int s2=refToOS.getCurrentTime();
292             System.out.println(s2-s1);
293             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
294         }
295     }
```

Figure 7.12 Testing System.out.println() on JARTOS

From the above tests, we claim that there is no impact of JARTOS on the performance of Java instructions.

Note:

During the process of thesis examination, we moved all the work to SunSPOT. The performance of SunSPOT is much better than TINI. The performance data of SunSPOT is listed in Table 7.5-1.

Table 7.5-1 Updated performance data.

Instruction	TINI	SunSPOT	JARTOS on SunSPOT
j=j+1	63microsec	0.47microsec	0.44microsec

while	130microsec	1.01microsec	1.02microsec
get time	7millisec	5.80microsec	6.4microsecond

7.3 Performance Measurement of JARTOS

In this section, the performance of JARTOS is measured and discussed. We measure the execution time of clock simulation in the Idle process and of the timer interrupt handler, since we want to completely characterize the timing performance of JARTOS and evaluate what duration between clock ticks is appropriated on TINI. This duration sets the base for the performance of JARTOS, as it specifies the maximum frequency of any process.

Then we test the flow of control of JARTOS. A test template is developed for the testing of flow of control. The test data produced by the test template application can be used to validate that the code achieves our system design.

Finally, we developed a reliability test template to evaluate the reliability of JARTOS. We want to measure the ability of JARTOS working for a long time. The test template application runs 24 hours and produces a performance record.

7.3.1 Testing Clock Simulation

We wrote a test program (Figure 7.13) to calculate the time that clock simulation takes. Clock tick simulation code is in the Idle process (line 3581,3582 in Figure 7.13). We measured 30msec. It takes 7msec to get the current time (Section 7.2.1). So it takes 23msec to simulate a clock tick.

```
3568    class Idle extends Process{
3569        public Idle(String procName, int synWaitTime,
3570                int synWaitPhase, int timeout, int event1,
3571                int event2, int message1, int message2,
3572                long processExitAddress){
3573            super(procName, synWaitTime, synWaitPhase,
3574                    timeout, event1, event2, message1,
3575                    message2, processExitAddress);
3576        }
3577
3578        public void processMethod(){
3579            int start=refToOS.getCurrentTime();
3580            //say 1 tick = 20hundredthsec
3581            if(refToOS.getCurrentTime()-refToOS.getPreviousTime()>20){
3582                timerInterruptHandler();
3583            }
3584            int finish=refToOS.getCurrentTime();
3585            int time=finish-start;
3586            System.out.println("clock simulation takes "+time);
3587            refToScheduler.runProcess(refToIdle);
3588        }
3589    }
```

Figure 7.13 Testing the clock simulation

7.3.2 Testing the Timer Interrupt Handler

We wrote a test program (Figure 7.14) in the Idle process to test the execution time of the timer interrupt handler. It takes 20msec. After subtracting the execution time of reading the clock (7msec), we calculated that it takes 13msec to execute the timer interrupt handler.

```
3567    class Idle extends Process{
3568        public Idle(String procName, int synWaitTime,
3569                int synWaitPhase, int timeout, int event1,
3570                int event2, int message1, int message2,
3571                long processExitAddress){
3572            super(procName, synWaitTime, synWaitPhase,
3573                    timeout, event1, event2, message1,
3574                    message2, processExitAddress);
3575        }
3576        public void processMethod(){
3577            //say 1 tick = 20hundredthsec
3578            if(refToOS.getCurrentTime()-refToOS.getPreviousTime()>20){
3579                int start=refToOS.getCurrentTime();
3580                timerInterruptHandler();
3581                int finish=refToOS.getCurrentTime();
3582                int time=finish-start;
3583                System.out.println("timer interrupt handler takes "+ time);
3584            }
3585            refToScheduler.runProcess(refToIdle);//sets Idle process to run
3586        }
3587    }
```

Figure 7.14 Testing timer interrupt handler

7.3.3 Testing the Process Overhead Time

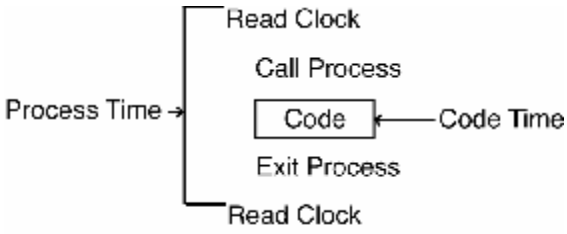

Figure 7.15 Process time

As shown in Figure 7.15, Process time consists of reading the clock time, code execution time and process overhead time. The read clock calls are to obtain the time data to measure the performance, so they represent the time overhead of performance probes. To measure the overhead time of calling a process, we wrote a Test process (Figure 7.16) without any code in the processMethod(). We measured that it takes 10msec to execute the Test process. After subtracting the execution time of reading clock (7msec), we worked out that the process overhead time is 3msec (10-7=3msec).

```
272     class Application3 extends Process{
273         public Application3(String procName, int synWaitTime, int synWaitPhase,
274         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
275             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
276             message1, message2, processExitAddress);
277         }
278         public void processMethod(){
279             //run nothing
280         }
281     }
```

Figure 7.16 Testing process time

7.3.4 Testing the Flow of Control of JARTOS

In this section we discuss the test of the flow of control in JARTOS. We can use the performance data to validate that the code achieves our system design. Three synchronous processes are required to run to maintain the JARTOS running. These are Timer process, Idle process and Test process. Test process is an application process on JARTOS, which we wrote for performance measurement. After running test process a

certain number of times, the scheduler runs the Performance Analysis process, which prints out the measured data.

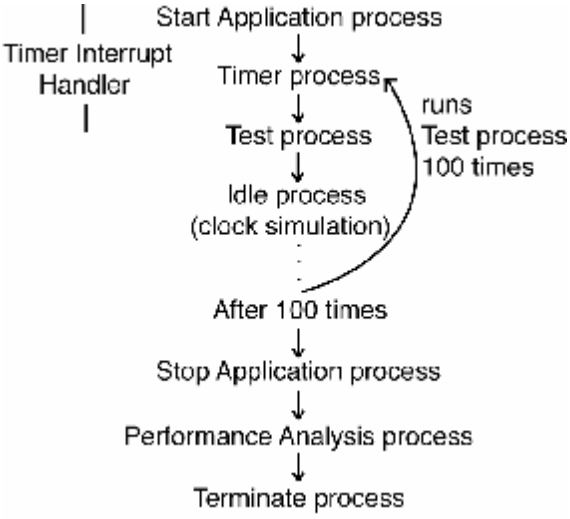

Figure 7.17 Performance evaluation template

Figure 7.17 shows a template of the test system. This test template can be used to measure the performance of any application simply by running the application processes as the Test process. The Test process (Figure 7.18), which is set to run by Start Application process, runs every 2 clock ticks. We wrote many different test processes to test the flow of control of JARTOS. One of the test processes is a process that turns on or turns off a LED on the TINI. The clock is simulated in the Idle process, which is set to tick every 200msec. After 100 executions of the scheduler loop, the Stop Application process stops the Test process. In the scheduler loop, performance probes (Section 4.6.2) are called before and after the execution of each process to collect performance data. The performance probes put process number and current time onto a circular buffer. Then the scheduler runs Performance Analysis process (Section 4.7.9) to produce the performance trace, and finally the Terminate process to terminate the JARTOS system.

```
309     class Application3 extends Process{
310         public Application3(String procName, int synWaitTime, int synWaitPhase,
311         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
312             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
313             message1, message2, processExitAddress);
314         }
315         BitPort bp = new BitPort(BitPort.Port3Bit5);
316         private boolean blinkyState=false;
317         public void setState(boolean state){
318             blinkyState=state;
319         }
320         public boolean getState(){
321             return blinkyState;
322         }
323         public void processMethod(){
324             System.out.println("test process is running!");
325             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
326             if(getState()==false){//if LED is off
327                 setState(true);
328                 bp.clear();//turn on LED
329             }
330             else{//if LED is on
331                 setState(false);
332                 bp.set();//turn off LED
333             }
334             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
335         }
336     }
```

Figure 7.18 A Test process testing the flow of control of JARTOS

As shown in Figure 7.19, a scheduler loop consists of scheduling time and the process time of a process. Scheduling time is the time that the scheduler takes to select which process is to run. Process time is the execution time of a process. The execution time of the start performance probe is included in the scheduling time, and the execution time of the finish performance probe is included in the process time. Previously we measured the probe time to be 7msec. In order to maintain consistency of data, we have left this overhead in the calculations in Table 7.6 and 7.7. Table 7.6 shows part of the raw test data that we collected. The first entry for each process is its start time, and the second entry is its finish time.

$$\text{Process time}=\text{Finish}_{p_n}-\text{Start}_{p_n}$$

$$\text{Schedule time}=\text{Start}_{p_{n+1}}-\text{Finish}_{p_n}$$

From the timing data (Column 3 in Table 7.5), we calculated the process time (Column 4) and scheduling time of each process execution (Column 5).

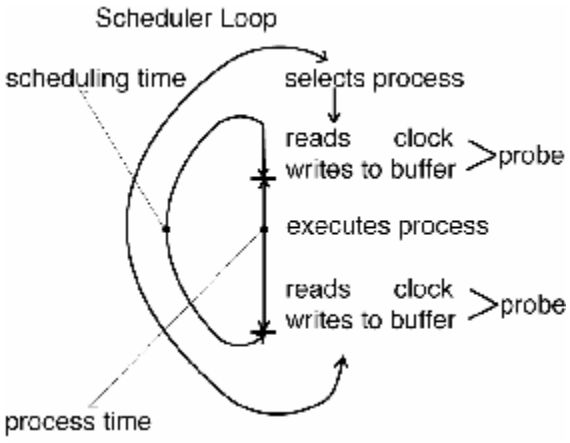

Figure 7.19 Scheduler loop

Table 7.6 Part of test data collected by Performance Analysis process

Process no.	Process name	Time/msec	Process time/msec	Scheduling time/msec
1 start	Timer process	5600740		
1 finish	Timer process	5600860	120	
6	Test process	5600880		20
6	Test process	5600900	20	
31	Idle process	5600930		30
31	Idle process	5600970	40	
31	Idle process	5601000		30
31	Idle process	5601040	40	
1	Timer process	5601060		20
1	Timer process	5601180	120	
31	Idle process	5601210		30
31	Idle process	5601250	40	
31	Idle process	5601280		30
31	Idle process	5601330	50	
1	Timer process	5601340		10
1	Timer process	5601460	120	
6	Test process	5601480		20
6	Test process	5601510	30	
31	Idle process	5601540		30
31	Idle process	5601570	30	
31	Idle process	5601600		30

31	Idle process	5601650	50	

Table 7.7 lists the average process time and average schedule time for each process. These times are all constant to the precision of our time measurement. We found that a process always has the same scheduling time, and that the process execution time of a process is fairly consistent. We also note that scheduling time varies with process number, because the scheduler iterates down the scheduler table until it finds a process to run. Consequently, scheduling time increases with the process number.

Table 7.7 Result of testing the flow of control of JARTOS

Process no.	Process name	Process time/msec (medium ± max)	Scheduling time/msec (medium ± max)
1	Timer process	120±10	10±10
6	Test process	30±10	20±10
31	Idle process	40±20	30±10

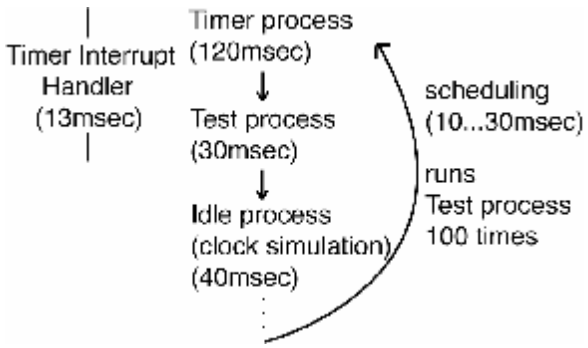

Figure 7.20 Flow of control with time on it

Figure 7.20 shows the flow of control of JARTOS with the execution time of each process. We observe that performance data that we collected conforms to the design of JARTOS.

From Table 7.7 we produced Figure 7.21, which shows the time relationships of the processes running on JARTOS in the test. The Timer process (p1) runs every clock tick, the Test process (p6) runs every 2 clock ticks, the Idle process (p31) runs the rest of time.

From the data in Table 7.6 we see that the Idle process runs twice every clock tick. Also, we see that when there are no processes to run, JARTOS runs the Idle process.

p1: Time = 1, Phase = 1

p2: Time = 2, Phase = 2

idle: rest of time – OS is checking for events etc.

Time means the number of clock ticks between executions. Phase is the clock tick relative to first.

Figure 7.21 Time relationships of process in the test

7.3.5 How Long should the Clock Tick be?

A significant design parameter in JARTOS is the time between clock ticks, whether the clock is simulated or a hardware interrupt. We chose a target, based on the 20/80 rule, that JARTOS spends 20% of time performing OS tasks including running the Timer process, leaving 80% of time for applications (Figure 7.22).

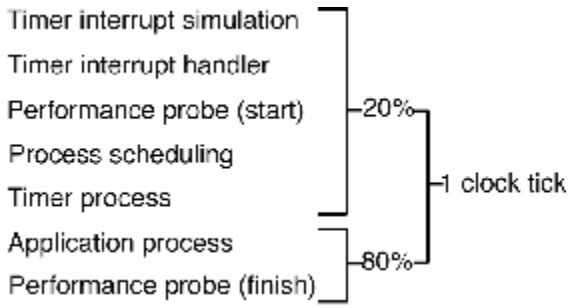

Figure 7.22 How long should the clock tick be?

We have obtained that the timer interrupt simulation and the timer interrupt handler take 23msec, process scheduling takes 20msec, and the Timer process and one performance probe take 120msec (Table 7.8).

OS tasks overhead = 163msec

Table 7.8 The time of performing OS tasks

OS tasks	Time/msec
Timer interrupt simulation+ Timer interrupt handler	23
Process scheduling	20
Timer process+ One performance probe	120

On these basis, the time between clock ticks should be

Duration=overhead/20%=815msec =>800msec

leaving 637msec (Duration-overhead=800-163=637msec) for application to run.

When using the simulated clock tick, the Idle process has to run to simulate the clock. To get a regular tick, we should allow sufficient application time for an integral number of ticks.

Number of ticks=application time/Idle process time=637/40=16

So the duration is sufficient to allow multiple checking for clock tick even when running a number of processes.

7.3.6 Reliability Testing of JARTOS

Finally, we evaluate the reliability of JARTOS working for a long time. Figure 7.23 depicts a template for reliability testing.

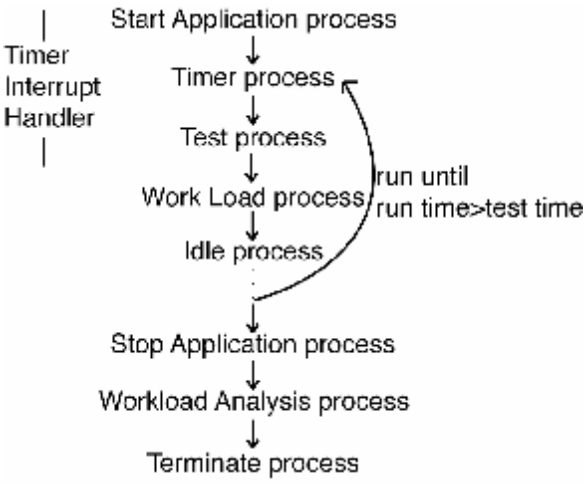

Figure 7.23 Reliability testing template

To maintain the JARTOS running, four synchronous processes are required to run, which are Timer process, Idle process, Test process (Figure 7.18), and Workload process (Algorithm 7.3). Test process and Workload process are set to run every 2 clock ticks. A simple Test process is a process that turns on or turns off the LED on the TINI. The Workload process counts the executions of itself, not them of the Test process. The Test process simply provides a workload to exercise OS functions and is not modified. To count the execution of the Test process it would have to be modified.

In the Workload process, counters and execution parameters of the Workload process are maintained. These counters include the number of execution of the Workload process. The parameters include the average, minimum and maximum times between executions of the Test process. If JARTOS runs for the full test time, the scheduler will set the Stop Application process, the Workload Analysis process (Algorithm 7.4) and the Terminate process to run. The Workload Analysis process is a process that prints out the run time, execute count and timing parameters.

Algorithm 7.3 Workload process

Execute count=Execute count+1

Average time=Run time/Execute count

IF time>Maximum time //time is the time between execution of the Test process

```
        Maximum time=time
    ENDIF
    IF time<Minimum time

        Minimum time=time
    ENDIF
    IF Run time>Test time THEN
        Set Stop Application process to run
        Set Workload Analysis process to run
        Set Terminate process to run
    ENDIF
```

Algorithm 7.4 Work Analysis process

```
    Print out the Run time
    Print out the Execution count
    Print out the Average time
```

In Section 7.3.5, we suggested that the clock tick should be 800msec. However, we ran a test with a 200msec clock tick to learn whether the reliability test would confirm that there is a problem. We set the Test process and Workload process to run every 2 clock ticks or 400msec.

We set the test time to 24 hours. That is 86,400,000msec. After running 24hours, we obtained the test result of the reliability test. The measured runtime is 86,400,070msec, and the execution count is 157,596. The average time between executions of the Workload process is 548msec, which is 37% longer than the expected 400msec.

As shown in Table 7.9, the total execution time is 265msec per clock tick with a 200msec clock tick or a 32.5% time overload. Therefore, we show that the 37% longer execution time is due to overload. If we set the clock tick to 400msec or 800msec, the total time per clock tick is less than the clock tick time.

Table 7.9 Analysis of the problem in the reliability testing

		200msed/tick	400msec/tick	800msec/tick
Timer process	Scheduling time	10msec	10msec	10msec
	Timer process	120msec	120msec	120msec
Simulated timer interrupt	Scheduling time	30msec	30msec	30msec
	Idle process	40msec	40msec	40msec
Test process	Scheduling time	10msec (20/2)	20msec	40msec (20*2)
	Test process	15msec (30/2)	30msec	60msec (30*2)
Workload process	Scheduling time	10msec (20/2)	20msec	40msec (20*2)
	Workload process	30msec (60/2)	60msec	120msec (60*2)
Total Time		265msec (>200msec)	310msec (<400msec)	460msec (<<800msec)

So reliability testing showed a problem and analysis showed the cause. The good news is that overloading JARTOS did not cause it to crash, the executive overload simply caused it to run late, i.e. it slowed down gracefully even though it failed to meet the deadlines.

Chapter 8

Conclusion and Future Work

The work performed during this thesis has showed that an RTOS named JARTOS has been developed in a safe language, Java. The thesis examined the advantages (Section 3.4.4) and associated problems (Section 3.4.4) of writing real-time operating systems (RTOSes) in a safe language, namely Java.

The design of JARTOS is a time-sharing design switching tasks on a timer interrupt. The scheduling of JARTOS is cooperative multiprocessing. Each application task is decomposed into several interacting processes to run on JARTOS. As each process is small relative to the task, the complexity of the code is reduced and its reliability is increased. A user process executes quickly and gives up the processor. Otherwise it will be timed out. To implement a timeout, JARTOS supports a timer interrupt handler that regularly updates a clock and checks for timeouts. There is a small-fast event monitor that polls I/O and sets event flags to tell the scheduler to run another process to respond to the event. To keep the number of interrupts to a minimum, input/output is done using polling where possible. Also, interrupt code is designed to be transparent to the processes. An interrupt handler sets flags and values, and then returns to the process it interrupted.

We introduced how we used Java constructs to implement the design of JARTOS. The majority of JARTOS is written in Java. Java can implement all high-level functions in the design. However, there are some low-level operations that cannot be coded in Java. The interrupt handler cannot be connected to hardware interrupt in Java. Also, the return address of the timed out process cannot be changed in Java. These can only be coded in

assembly language. We did not solve the low-level issues of JARTOS. We are still doing research on TINI hardware and TINI Native Interface.

In JARTOS, application code is separated from the OS code, so that the programmer of the application only has to write application code and make no changes to the OS. Thus, they can focus on programming the real-time task because many of the low-level details are abstracted away by the OS. JARTOS has passed a given set of test applications, which are documented, extended and run every time any part of the OS is changed. Test harnesses and assertions are also used to test JARTOS.

The final stage of the work was the performance measurement of JARTOS. We investigated the timing problems by a set of performance measurements. Performance on the TINI and JARTOS are measured and discussed. We note that the performance data we collected conforms to the design of JARTOS.

One surprise was how poor the performance of the TINI JVM is. The developers of TINI claim that each thread is assigned an 8 msec time slice on TINI [TINI, 2007]. The test data shows that it takes 0.063 msec to execute a fundamental instruction unit on TINI. That is 126 fundamental instructions per slice, which means it will not do much in any slice. So we think the time slice used in TINI OS should be longer. Also, the performance data of TINI shows that the speed of TINI is much slower than we expected.

As a safe language, Java is suitable for coding a safety-critical RTOS. The Java compiler handles potentially unsafe operations rather than the programmer. Also, Java includes run-time support to catch and handle run-time errors. However, the low-level operations cannot be coded in Java, which is a main problem of writing an RTOS in Java. We can only rely on the native interface provided by the JVM. So, the relevant documentation should be sufficient for the developers to study.

The documentation of TINI is poor, which is inconvenient for doing research on the TINI board. Also, there are some omissions in the TINI API, such as reflection [TINI, 2007].

These hampered the development of JARTOS. So, we conclude that TINI is better suited to developing stand alone programs than to developing an RTOS.

8.1 Future Work

8.1.1 Low-level issues

During the period when the thesis was being examined, we ported JARTOS to the SunSPOT and designed ways to solve the remaining low-level issues (Section 5.4). We will implement our new design on the SunSPOT and finish all the performance testing of JARTOS running on the SunSPOT in future research.

8.1.2 Network

We plan that JARTOS will run on a multiple processors connected by a network. For example, the control of a mobile robot may be decomposed into motion control, ultrasonic sensing, vision and task planning, each running on a separate processor. The application design will distribute processing to multiple processes over the network, for example on sensor networks.

8.1.3 Sun SPOT

During the process of the thesis examination, we moved JARTOS from TINI to Sun SPOT embedded microcontrollers. As discussed in Section 7.1, the design of TINI has some problems, for example, each process is only assigned an 8msec time slice, which is too short for running a process to completion. Also, TINI provides poor documentation, which is not convenient for research. The performance data of SunSPOT is much better than TINI (Chapter 7).

There are two questions of interest in our future research. Does the design of the Sun SPOT enable the use of an RTOS or is it best suited to stand alone programs? What performance can be achieved by an RTOS running on a Sun SPOT?

With the release of the Sun SPOT in April 2007, Sun [SPOT, 2007] 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 *IEEE802.25.4* wireless links. As shown in Figure 8.1, the left part is the suite creator that runs on the host, and the right part is the architecture of the embedded device.

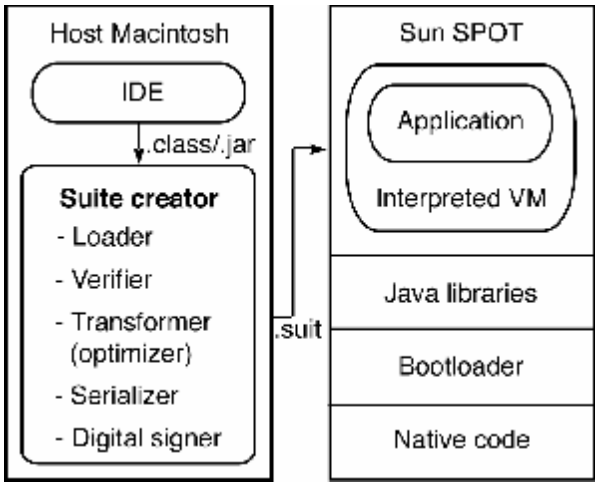

Figure 8.1 Sun SPOT

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 (Figure 8.1). 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 to machine code, improving performance and removing the need for just-in-time compilation [Shaylor, *et. al.*, 2003].

Applications are represented as objects that are instances of the Isolate class to isolate them from one another. Sun [SPOT, 2007] 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 multiprocessing. 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, which seems to be more appropriate for our future research on JARTOS.

Bibliography

1 Anderson, D.A. (1981), Operating Systems, *IEEE Trans. Computers*, June 1981.

2 Barrett, S. F. and Park, D. J. (2005), *Embedded systems design and applications with the 68HC12 and HCS12*, Pearson/Prentice Hall, 2005.

2 Bartezko, D., Fischer C., Möller M., Wehrheim H. (2001), *Jass – Java with Assertions*, Electronic Notes in Theoretical Compuer Science, Proceedings of RV 01, Paris, France, Volume 55, Issue 2, July 2001

3 Bluebotics (2007), *Bluebotics: mobile robots at your service*, http://www.bluebotics.com/

4 Brega, R. (2002), *A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications*, PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 2002.

5 Brega R., Tomatis N., Arras K.O. (2000), The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion, *IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, Takamatsu, Japan, 2000.

6 Broekman, B., Notenboom, E. (2003), Testing Embedded Software, Addision-Wesley, 2003.

7 Burns, A., and Wellings, A.J. (2001), *Real-Time Systems and Programming Languages*, 3rd edition, Addison-Wesley, 2001.

8 Buttazzo, G. C. (1997), *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, 1997.

9 Cheng, A. (2002), *Real-time Systems: Scheduling, Analysis, and Verification*, John Wiley & Sons, 2002.

10 Cox, I.J., Kapilow D.a., Kropfl W.J., and Shopiro J.E. (1988), Real-time Software for Robotics, *AT&T Technical Journal*, March/April 1988, Volume 67, Issue 2.

11 Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D. and Morris, R. (2002), Event-driven Programming for Robust Software, *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, pages 186-189, September 2002.

12 Edwards, S. A. (2000), *Languages for Digital Embedded Systems*, Kluwer Academic Publisher, 2000.

13 Ethernut (2007), *Ethernut Software Manual*, <http://www.ethernut.de/en/documents/index.html>.

14 Evesham, D.E. (1990), *Developing Real-time Systems*, Wilmslow: Sigma, 1990.

15 Ford, W. and Topp, W. (1988), *MC68000 Assembly Language and Systems Programming*, DC Heath and Company, 1988.

16 Fritzinger, J. S. and Mueller, M. (1996), *Java Security*, Sun Microsystems White Paper, 1996.

17 Glass, R.L. (1980), Real-Time: The "Lost-World" Of Software Debugging and Testing, *Communication of the ACM*, Volume 28 Number 5, May 1980.

18 Golm, M., Felser, M., Wawersich, C. and Kleinoeder J. (2002), A Java Operating System as the Foundation of a Secure Network Operating System, *Technical Report TR-I4-02-05*, August 2002

19 Golm, M., Felser, M., Wawersich, C. and Kleinoeder J. (2002), the JX operating system. *In Proceedings of the USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

20 Golm, M., Kleinoeder, J., Bellosa, F. (2001), Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System, *8th Workshop on Hot Topics in OS* (HotOS-VIII), 2001.

21 Gosling, J., Joy, B. and Steele, G. (1996), *The Java Language Specification*, Addison-Wesley: Reading, MA, 1996.

22 Gries, D. (2006), What have we not learned about teaching Programming?, *IEEE Computer*, October, 2006, pp81-pp82.

23 Gritzalis, S. and Iliadis, J. (1998), Addressing security issues in programming languages for mobile code, DEXA *'98 9th Workshop of Database and Expert Systems Applications*, R. Wagner (Ed.), pp. 288-293, August 1998, Vienna, Austria, IEEE Computer Society Press.

24 Gustafsson, A. (2005), Threads without the pain, *ACM Queue*, Vol3, No 9, 2005.

25 Holzmann, G.J. (2006), The Power of 10: Rules for Developing Safety-Critical Code, *IEEE Computer*, June 2006, pp95-97.

26 Horton, I. (2000), *Beginning Java 2*, Wrox Press, Birmingham, UK, 2000.

27 Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fahndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B. (2005), An Overview of the Singularity Project, *Technical Report MSR-TR-2005-135*, Microsoft Research, 2005.

28 Justice, B., Osborne, S. and Wills, S. (1987), Concurrent Operating System, Build It With Modula-2, *Micro Cornucopia*, No 34, Feb-Mar, 1997, pp26-32.

29 Krishna, C. M. and Shin, K. G (1997), *Real-time systems*, McGraw-Hill, 1997.

30 Labrosse, J. J. (1999), *µC/OS-II: the Real-Time Kernel*, R&D Publications, 1999.

31 Laplante, P.A. (2004), *Real-Time Systems Design and Analysis*, 3rd Edition, Wiley-IEEE Press, 2004.

32 Lee, E. A. (2000), What's Ahead for Embedded Software, *IEEE Computer*, vol. 33, pp. 18-26, September 2000.

33 Lee, E. A. (2006), The problem with threads, *IEEE Computer*, 39(5), May 2006, pp 33-42.

34 Li, Q., Yao, C. (2003), *Real-Time Concepts for Embedded Systems*, CMP Books, 2003.

35 Li, Y., Potkonjak, M. and Wolf, W. (1997), Real-Time Operating Systems for Embedded Computing, *Proc. Int'l Conf. Computer Design*, Oct. 1997.

36 Liu C.L. and Layland J.W. (1973), Scheduling Algorithms for. Multiprogramming in a Hard Real-Time Environment, *Journal of the ACM*, Vol.20, No.1, January 1973.

37 Liu, J.W.S. (2000), *Real-time systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

38 Marwedel, P. (2003), *Embedded System Design*, Kluwer Academic Publisher, 2003.

39 McCluskey, G. (1998), Using Java Reflection, *Java Developer Connection*, 1998.

40 McGraw, G. and Felton, E. W. (1999), *Securing Java: Getting Down to Business with Mobile Code*, 2nd Edition, John Wiley & Sons, 1999.

41 M^cKerrow, P.J. (1988), *Performance Measurement of Computer Systems*, Addison-Wesley, 1988.

42 M^cKerrow, P.J. and Antoune, S. (2007), Research into Navigation with CTFM Ultrasonic Sensors, *Proceedings 63rd Annual Meeting of the Institute of Navigation*, Boston, April 23-25, 2007, CD paper.

43 M^cKerrow, P.J., Lu, Q., Zhou, Z.Q. and Chen, L. (2007), Developing real-time systems in Java on Macintosh, *Submitted to AUC'07*, Apple University Consortium, Gold Coast, September, 23-26, 2007.

44 M^cKerrow, P.J., Lu, Q., Zhou, Z.Q. and Chen, L. (2007), Software development of embedded systems on Macintosh, *Submitted to AUC'07*, Apple University Consortium, Gold Coast, September, 23-26, 2007.

45 Mosse, D., Gudmundsson, O. and Agrawala, A. K. (1990), Prototyping Real Time Operating Systems: A Case Study, *Proceedings of the First International Workshop on Rapid System Prototyping*, pp. 144-154, Research Triangle Park, North Carolina, June 1990.

46 Mössenböck, H. P. and Wirth, N. (1991), The Programming Language Oberon-2, *Structured Programming*, 12:4, pp.179-195, 1991.

47 Nikitin,E. (1997), Into the Realm of Oberon, *Springer*, New York, 1997.

48 Nilsen, K. (1996), Issues in the Design and Implementation of Real-Time Java, *Java Developer's Journal*, 1996. 1(1): pp. 44-57, Sun Microsystems Inc.

49 Nissanke, N. (1997), *Real-time systems*, Prentice Hal, 1997.

50 Oaks, S. (2001), *Java Security*, O'Reilly, 2001.

51 Ousterhout, J. (1996), Why Threads Are A Bad Idea (for most purposes). In USENIX Technical Conference (Invited Talk), Austin TX, January 1996.

52 Pottie, G. and Kaiser, W. (2005), *Principles of Embedded Networked Systems Design*, Cambridge University Press, 2005.

53 Plösch, R. (2002), Evaluation of Assertion Support for the Java Programming Language, *Journal of Object technology (JOT)*, Vol. 1, No. 3, August 2002.

54 Puschner, P. and Burns, a. (2000), A review of WCET analysis, *Real Time System*, vol. 18, no. 2/3, 2000.

55 Purser,W.F.C., Jennings, D.M. (1975), The Design of a Real-Time Operating System for a Minicomputer. Part 1, *Software-Practice and Experience*, Vol.5, 147-167, 1975.

56 Richard W. S. (1999), *UNIX Network Programming, Volume 2: Interprocess Communications*, Second Edition. Prentice Hall, Englewood Cliffs, NJ, 1999

57 Rogers P. (2001), *J2SE 1.4 premieres Java's assertion capability* – Part 1, JavaWorld, November 2001, < http://www.javaworld.com/javaworld/jw-11-2001/jw-1109-assert_p.html >

58 Rogers P. (2001), *J2SE 1.4 premieres Java's assertion capability* – Part 2, JavaWorld, November 2001, < http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-assert_p.html >

59 Rizk, A. and Halsall, F.(1987), Design and Implementation of a C-based Language for Distributed Real-time Systems, *SIGPLAN NOTICES*, V22 #6 1987.

60 Sethi, H. (2002), *Java Security*, Premier Press, 2002.

61 Shaylor, N., Simon, D. and Bush, B. (2003), A Java Virtual Machine Architecture for Very Small Devices. *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM Press, June 2003.

62 Silberschatz, A., Galvin, P. and Gagne, G. (2000), *Applied Operating Systems Concepts*, 1st Edition,

Wiley, 2000.

63 Simon, D., Cifuentes, C., Cleal, D., Daniels, J. and White, D. (2006), Java(TM) on the Bare Metal of Wireless Sensor Devices -- The Squawk Java Virtual Machine, *VEE'06*, Ottawa, July 2006.

64 Smith, R., Cifuentes, C. and D. Simon (2005), Enabling Java for Small Wireless Devices with Squawk and SpotWorld, *In Building Soft. for Pervasive Computing '05*, 2005.

65 Sosnoski, D. (2003), *Java programming dynamics, Part 2: Introducing reflection*, < http://www.ibm.com/developerworks/library/j-dyn0603/>.

66 SPOT, *Sun SPOT: Programming the Real-world*, <http://sunspotworld.com/docs/>

67 Stankovic, .J.A. (1982), Software Communication Mechanisms: Procedure Calls Versus Messages, *IEEE Trans. Computers*, April 1982.

68 Sun Microsystems, *Java Assertion Facility*, <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>.

69 Szyperski, C. and Gough, J. (1995), The Role of Programming Languages in the Life-Cycle of Safe Systems, *Second International Conference on Safety Through Quality (STQ'95)*, Kennedy Space Center, Cape Canaversal, Floria, USA, October 1995.

70 Tanenbaum A.S., Herder, J.N. and Bos, H. (2006), Can we make operating systems reliable and secure, *IEEE Computer*, May 2006, pp 44-51.

71 TINI (2007), *TINI networked microcontroller*, <http://www.maxim-ic.com/products/microcontrollers/tini/>.

72 Tomatis, N., Brega, R., Jensen, B., Arras, K., Moreau, B., Persson, J., Siegwart, R.(2001), A Complex Mechatronic System: From Design to Application. , *Proc. IEEE/ASME Int. Conf. on Advanced Intelligent Mechatronics (AIM)*, Como, Italy, July 8-11, 2001.

73 Tomatis N., Terrien G., Piguet R., Burnier D., Bouabdallah S**.,** Arras K.O., Siegwart R. (2003), Designing a Secure and Robust Mobile Interacting Robot for the Long Term, *IEEE International Conference on Robotics and Automation (ICRA'03)*, Taipei, Taiwan, 2003.

74 Ulrickson, R.W. (1977), Real-time systems often use interrupts, *Electronic Design*, May 10,1977.

75 Venners, B. (1996), Java's garbage-collected heap, *JavaWorld*, August 1996, <http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html>.

76 Williams, R. (2006), *Real-time systems development*, Elsevier Butterwork-Heinemann, 2006.

77 Wellings, A. (2004), *Concurrent and real-time programming in Java*, Wiley, 2004.

78 Worth, N. (2006), Good Ideas, through the Looing Glass, *IEEE Computer*, January, pp28-39.

Appendix A

System Library

Class OS

public class OS

In the OS class, there are main method, constructor, OS tables, OS methods, OS processes and supervisor calls. OS class is responsible for maintaining the system running. The operating system is an instance of an OS class. The operating system is started by running the main() method.

Field Summary	
int	Clock The clock tick of the OS.
boolean	enableTimeoutsFlag Timeout flag.
boolean	enableDebugFlag Debug flag.
int	timeoutCounter Timeout counter.
boolean	timeoutReportFlag Timeout report flag.
int	numberOfProcTimeout Total number of process that is timed out.
boolean	enablePerformProbes Perform probes flag.
int	currentProcessNumber The current running process number on OS.
int	startOfMemoryBlocks The start address of memory blocks.
long	commonDataAddress

	The common data address.
long	circularListsAddress The circular list address.
long	previousTime The previous time.
long	currentTime The current time of OS.

Method Summary	
void	setClock(int clock) Sets the clock value of OS with the given clock.
int	getClock() Gets the clock value of OS.
void	setEnableTimeoutsFlag(boolean flag) Enables or disables timeouts flag.
boolean	getEnableTimeoutsFlag() Gets enable timeouts flag value.
void	setEnableDegugFlag(boolean flag) Enables or disables debug flag.
boolean	getEnableDegugFlag() Gets enable debug flag value.
void	setTimeoutCounter(int t) Sets timeout counter for the process.
int	getTimeoutCounter() Gets timeout counter.
void	setTimeoutReportFlag(boolean flag) Sets timeout report flag.
boolean	getTimeoutReportFlag() Gets timeout report flag.
void	setNumberOfProcessTimeout(int n) Sets the total number of processes that are timed out.
int	getNumberOfProcessTimeout() Gets the total number of processes that are timed out.
void	setEnablePerformProbes(boolean is) Enables or disable performance probes.

boolean	getEnablePerformProbes() Gets enable performance probes value.
void	setCurrentProcessNumber(int procNum) Sets the process number of current running process.
int	getCurrentProcessNumber() Gets the process number of current running process.
void	setStartOfMemoryBlocks(int addr) Sets the start address of memory blocks.
int	getStartOfMemoryBlocks() Gets the start address of memory blocks.
void	setCommonDataAddress(long commonDataAddress) Sets the address of common data.
long	getCommonDataAddress() Gets the address of common data.
void	setCListsAddress(long address) Sets the address of circular list.
long	getCListsAddress() Gets the address of circular list.
void	setPreviousTime(long p) Sets the previous time.
long	getPrevousTime() Gets the previous time.
void	setCurrentTime(long c) Sets the current time of OS.
long	getCurrentTime() Gets the current time of OS.
void	setupOSTable (int clock, boolean enableTimeoutFlag, boolean enableDebugFlag, int timeoutCounter, boolean timeoutReportFlag, int numberOfProcTimeout, boolean enablePerformProbes, int currentProcessNumber,int startOfMemoryBlocks, long commonDataAddress, long CListAddress, long previousTime) Sets up OS table.
void	addToOSTable(Process proc) Adds a process to OS table.
void	getOSTable() Returns OS table.
void	waitState(int n) Goes into wait state for n clock ticks.

void	changePriority() Changes the priority of user processes by moving them in process table.
void	simulateEvent() Switches from hardware event to software event simulator for testing.
void	getOSTables() Returns the current value of all OS tables for use in debugging, testing and performance measurement.
void	libraryOfEventHandlers() Provides the library of event handlers.
void	timerInterruptHandler() Sets flag to run timer process and handles time out.
void	enableTimerInterrupt(boolean is) Enables or disables the timer interrupt.
void	performanceProbe() Collects performance data.
void	testOS() A test harness for OS class
void	testProcessTable() A test harness for ProcessTable class
void	testScheduler() A test harness for Scheduler class
void	testEvent() A test harness for Event class
void	testMessage() A test harness for Message class
void	testCircularBuffer() A test harness for CircularBuffer class
void	testCommonData() A test harness for CommonData class
public static void	main(String[] args) Enables timer interrupt, sets up OS tables and call scheduler to start OS.

Class ProcessTable (Inner class of OS class)

ProcessTable class is used to store process table and relevant methods.

Field Summary	
int	Size The size of process table.
int[]	PROCESS_NUMBER The number of the process
String[]	PROCESS_NAME The name of the process.
boolean[]	WAITING_ON_EVENT If the process is waiting on an event.
int[]	EVENT_NUMBER The event number of an event that a process is waiting on.
boolean[]	EVENT_OCCURRED If the event has occurred.
boolean[]	WAITING_ON_MESSAGE If the process is waiting on a message.
int[]	MESSAGE_NUMBER The message number of a message that the process is waiting on.
boolean[]	MESSAGE_ARRIVED If the message is available.
boolean[]	WAITING_ON_TIME If the process is waiting on time.
boolean[]	TIME_IS_UP If the time is up.
String[]	REFERENCE_TO_PROCESS_METHOD The reference to the process method.

Method Summary	
void	setupProcessTable() Sets up process table.
void	addToProcessTable(Process proc) Adds a process to process table.
void	removeFromProcessTable(String procName) Removes a process from process table.

int	getProcessNumber(String procName) Gets the process number of the process with the given process name.
void	setEventOccurred(int procNum, boolean is) Sets the eventOccurred value to indicate if the event has occurred.
boolean	getEventOccurred(int procNum) Gets the eventOccurred value to check if the event has occurred.
void	setWaitingOnEvent(int procNum, boolean is) Sets the waitingOnEvent value to indicate if there is a process waiting on the event.
boolean	getWaitingOnEvent(int procNum) Gets the waitingOnEvent value to check if there is a process waiting on the event.
void	setEventNumber(int procNum, int eventNum) Sets the event number which process is waiting for.
int	getEventNumber(int procNum) Gets the event number which process is waiting for.
void	setWaitingOnMessage(int procNum, boolean is) Sets the waitingOnMessage value to indicate if there is a process waiting on the message.
boolean	getWaitingOnMessage(int procNum) Gets the waitingOnMessage value to check if there is a process waiting on the message.
void	setMessageNumber(int procNum, int messageNum) Sets the message number which process is waiting for.
int	getMessageNumber(int procNum) Gets the message number which process is waiting for.
void	setMessageArrived(int procNum, boolean is) Sets the messageArrived value to indicate if the message has arrived.
boolean	getMessageArrived(int procNum) Gets the messageArrived value to check if the message has arrived.
void	setWaitingOnTime(int procNum, boolean is) Sets the waitingOnTimeValue.
boolean	getWaitingOnTime(int procNum) Gets the waitingOnTimeValue.
void	setTimeIsUp() Sets the timeIsUp value to indicate if time is up.
boolean	getTimeIsUp() Gets the timeIsUp value to check if time is up.
void	setReferenceToProcessMethod(int procNum, String reference) Sets the reference to the process.
String	getReferenceToProcessMethod(int procNum)

	Gets the reference to the process.
int	getIndexInProcessTable(String procName) Gets the index of the process by the given process name in process table.
void	resetProcessTable() Resets the process table.
void	getProcessTable() Returns the current value of process table.
void	loadProcess(Process proc) Loads a process in the process table.
void	removeProcess(String procName) Removes a process from the process table.

Class Scheduler (Inner class of OS class)

Scheduler class is used to store scheduler method, scheduler table and relevant methods.

Field Summary	
int	Size The size of scheduler table.
int[]	PROCESS_NUMBER The number of the process.
boolean[]	EXECUTE_FLAG Execute flag of the process.
boolean[]	LOADED_FLAG Loaded flag of the process.
int[]	WAIT_TIME Wait time of the process.
int[]	WAIT_PHASE Wait phase of the process.
int[]	TICK_COUNT Tick count of the process.
boolean[]	WAITING_ON_EVENT If the process is waiting on an event.

Method Summary	
void	setupSchedulerTable(int processNumber, bololean executeFlag, boolean loadedFlag, int waitTime, int waitPhase, int tickCount, boolean waitingOnEvent) Sets up the scheduler table.

void	addToSchedulerTable(int processNumber) Adds a process to the scheduler table.
void	addToSchedulerTable(Process proc) Adds a process to the scheduler table.
void	removeFromSchedulerTable(String procName) Removes a process from the scheduler table.
void	removeFromSchedulerTable(int processNumber) Removes a process from the scheduler table.
int	getIndexInSchedulerTable(int processNumber) Gets the index of the process by given process number in the scheduler table.
void	getSchedulerTable() Returns the current value of scheduler table.
int	getProcessIndexInSchedulerTable(int processNumber) Gets the index of the process by given process number in the scheduler table.
int	getProcessNumberInSchedulerTable() Gets the number of processes in scheduler table.
boolean	getIsProcessInSchedulerTable(int processNumber) Checks if the process is in the scheduler table.
int	getProcessNumber(int procNum) Gets the processes number.
void	setExecuteFlag(int procNum, boolean flag) Sets the execute flag value.
boolean	getExecuteFlag(int procNum) Gets the execute flag value.
void	setLoadedFlag(int procNum, boolean flag) Sets the loaded flag value.
void	setWaitTime(int procNum, int time) Sets the wait time value.
void	setWaitPhase(int procNum, int phase) Sets the wait phase value.
void	setTickCount(int procNum, int tick) Sets the tick count value.
int	getTickCount(int procNum) Gets the tick count value.
void	setWaitingOnEvent(int procNum, boolean is) Sets the waitingOnEvent value to check if the process is waiting on event.
void	enableProcess(Process proc)

	Enables a process by adding it to scheduler table.
void	enableProcess(int processNumber) Enables a process by adding it to scheduler table.
void	disableProcess(String procName) Disable a process by removing it from scheduler table.
void	disableProcess(int processNumber) Disable a process by removing it from scheduler table.
void	runProcess(Process proc) Sets the execute flag to true in the scheduler table for a process.
void	runProcess(String processName) Sets the execute flag to true in the scheduler table for a process.
void	stopProcess(int processNumber) Sets the execute flag to false in the scheduler table for a process.
void	schedulerInfiniteLoop() Decides which process is to run and dispatches it.

Class Message (Inner class of OS class)

Message class is used to store message table and relevant methods.

Field Summary	
int	Size The size of message table.
int[]	MESSAGE_NUMBER The number of the message.
boolean[]	ALLOCATED If the message object is allocated.
int[]	FROM_PROCESS The process that is sending the message.
int[]	TO_PROCESS The process that is receiving the message.
boolean[]	WAITING_FOR_MESSAGE The process that is waiting for the message.
boolean[]	MESSAGE_SENT If the message has been sent.
String[]	MESSAGE_REFERENCE The reference to the message.
String[]	MESSAGE_TYPE

	The type of the message.
int[]	TRANSACTION_NUMBER The transaction number of the message.
long[]	MESSAGE_SENT_TIME The time when the message is sent.
boolean[]	OVERFLOW_FLAG The overflow flag of the message.

Method Summary	
void	setupMessageTable(boolean allocated, int fromProcess, int toProcess, boolean waitingForMes, boolean messageSent, String messageReference, String messageType, int transactionNumber, long messageSentTime, boolean overflowFlag) Sets up message table.
int	getMessageNumberInMessageTable() Gets the number of messages in message table.
int	getMessageIndexNumber(int toProcessNumber) Gets the index of message in message table.
int	getMessageNumber(int messageNum) Gets message number.
void	setAllocated(int messageNum, boolean is) Sets the allocated value in message table.
boolean	getAllocated(int messageNum) Gets the allocated value to check if the message has been allocated.
void	setFromProcess(int messageNum, int processNumber) Sets the process sending a message.
int	getFromProcess(int messageNum) Gets the process sending a message.
void	setToProcess(int messageNum, int procNumber) Sets the process receiving a message.
int	getToProcess(int messageNum) Gets the process receiving a message.
void	setWaitingForMessage(int messageNum, boolean is) Sets waitingForMessage value.
boolean	getWaitingForMessage(int messageNum) Checks to see if there is a process waiting for message.
void	setMessageSent(int messageNum, boolean is)

	Sets messageSent value.
boolean	getMessageSent(int messageNum) Checks to see if the message has been sent.
void	setMessageType(int messageNum, String type) Sets the type of message.
String	getMessageType(int messageNumber) Gets the type of message.
void	setMessageReference(int messageNum, String messageReference) Sets the reference of message.
String	getMessageReference(int messageNum) Gets the reference of message.
void	setMessageSentTime(int messageNum, long time) Sets the time of sending message.
long	getMessageSentTime(int messageNum) Gets the time of sending message.
void	setTransactionNumber(int messageNum, int transactionNumber) Sets the transaction number of message.
int	getTransactionNumber(int messageNum) Gets the transaction number of message.
void	setOverflowFlag(int messageNum, boolean is) Sets the overflow flag.
boolean	getOverflowFlag(int messageNum) Gets the overflow flag.
int	getIndexInMessageTable(int messageNum) Get the message index in message table.
void	getMessageTable() Returns the current value of message table.
int	getMessage(String fromProcessName, String toProcessName, String messageType) Gets a message object.
void	sendMessage(int msgNumber, String toProcessName, String messageReference) Writes message into message buffer and sets available flag.
void	receiveMessage(String toProcessName) Process asks for message. If it has been sent process reads it and continues. If not process sets message wait and exit.
void	releaseMessage(int messageNumber) Returns message resource to OS.

Class Event (Inner class of OS class)

Event class is used to store event table and relevant methods.

Field Summary	
int	Size The size of the event table.
int[]	EVENT_NUMBER The number of the event.
String[]	EVENT_TYPE The type of the event.
String[]	INTERRUPT_OR_POLLED Is the event an interrupt event or polled event.
boolean[]	EVENT_ENABLED If the event is enabled.
boolean[]	EVENT_OCCURED If the event has occurred.
int[]	PROCESS_WAITING_ON_EVENT The process that is waiting on the event.
String[]	REFERENCE_TO_EVENT_METHOD The reference to the event method.
String[]	REFERENCE_TO_INTERRUPT_HANDLER The reference to the interrupt handler.

Method Summary	
void	setupEventTable(int eventNumber, String eventType, String interruptOrPolled, boolean eventEnabled, boolean eventOccured, int processWaitingOnEvent, String referenceToEventMethod, String referenceToInterruptHandler) Sets up event table.
void	addToEventTable(String eventType, String interruptOrPolled, Boolean eventEnabled, Boolean eventOccurred, String procName, String refToEventMethod, String refToHandler) Adds an event to event table.
void	getEventTable() Returns current value of event table.
int	getEventNumberInEventTable() Gets the total number of event in event table.
void	setEventNumber(int eNum, int eventNumber) Sets eventNumber value.

void	setEventType(int eNum, String eventType) Sets the type of event.
String	getEventType(int eNum) Gets the type of event.
void	setInterruptOrPolled(int eNum, String iop) Sets the value to display that the event is an interrupt event or polled event.
void	setEveEnabled(int eNum, boolean is) Enables or disables event.
boolean	getEveEnabled(int eNum) Checks if the event is enabled or not.
void	setEveOccured(int eNum, boolean is) Sets the event has occurred or not.
boolean	getEveOccured(int eNum) Checks if the event has occurred.
void	setProcessWaitingOnEvent(int eNum, int processNumber) Sets the process number that is waiting on an event.
int	getProcessWaitingOnEvent(int eNum) Gets the process number that is waiting on an event.
void	setReferenceToEventMethod(int eNum, String method) Sets the reference to an event method.
String	getReferenceToEventMethod(int eNum) Gets the reference to an event method.
void	setReferenceToInterruptHandler(int eNum, String method) Sets the reference to an interrupt handler.
int	getIndexInEventTable(int processNumber) Gets the index in event table.
void	waitEvent(int eventNumber, int processNumber, String eventType) Waits for an event.
void	interruptHandler(int eventNumber) Handles the interrupt.
void	enableEvent(int eventNumber) Enables an event.
void	disableEvent() Disables an event.

Class CircularBuffer (Inner class of OS class)

CircularBuffer is used to store circular buffer table and relevant methods.

Field Summary	
int	Size The size of circular buffer table.
int[]	BUFFER_NUMBER The number of the buffer.
boolean[]	ALLOCATED If the buffer has been allocated.
String[]	BUFFER_REFERENCE The reference to the buffer.
int[]	ADD_PROCESS The process that adds the buffer.
int[]	REMOVE_PROCESS The process that removes the buffer.
boolean[]	OVERFLOW_FLAG The overflow flag.
int[]	ADD_INDEX The index of adding buffer.
int[]	REMOVE_INDEX The index of removing buffer.

Method Summary	
void	setupCircularBufferTable(boolean allocated, String bufferReference, int addProcess, int removeProcess, boolean overflowFlag, int addIndex, int removeIndex) Sets up circular buffer table.
void	addToCircularBufferTable(int bufferNumber, String bufferReference,int addProcNumber, int removeProcNumber,int addIndex, int removeIndex) Adds a buffer to circular buffer table.
void	removeFromCircularBufferTable(int bufferNum) Removes a buffer from a circular buffer table.
void	setBufferNumber(int bufferNum, int bufferNumber) Sets the buffer number.
void	getBufferNumber(int bufferNum) Gets the buffer number.
void	setAllocated(int bufferNum, boolean is) Sets the allocated value to display if the buffer has been allocated.

boolean	getAllocated(int bufferNum) Checks if the buffer has been allocated.
void	setBufferReference(int bufferNum, String bufferReference) Sets the reference to the buffer.
String	getBufferReference(int bufferNum) Gets the reference to the buffer.
void	setAddProcess(int bufferNum, int addProcessNumber) Sets the process that adds a buffer.
int	getAddProcess(int bufferNum) Gets the process that adds a buffer.
void	setRemoveProcess(int bufferNum, int removeProcessNumber) Sets the process that removes a buffer.
int	getRemoveProcess(int bufferNum) Gets the process that removes a buffer.
void	setOverflowFlag(int bufferNum, boolean overflowFlag) Sets the overflow flag.
boolean	getOverflowFlag(int bufferNum) Gets the overflow flag.
void	setAddIndex(int bufferNum,int addIndex) Sets the index of added buffer.
int	getAddIndex(int bufferNum) Gets the index of added buffer.
void	setRemoveIndex(int bufferNum,int removeIndex) Sets the index of removed buffer.
int	getRemoveIndex(int bufferNum) Gets the index of removed buffer.
void	getCircularBufferTable() Gets the circular buffer table.
int	getCircularBuffer(int addProcessNumber, int removeProcessNumber) Gets a circular buffer object.
void	addToCircularList(int bufferNumber, String bufferReference,int addProcNumber, int removeProcNumber, int addIndex, int removeIndex) Adds a buffer to circular list.
String	removeFromCircularList(int bufferNumber) Removes a buffer from circular list.

Class CommonData (Inner class of OS class)

CommonData is used to store common data table and relevant methods.

Field Summary	
int	size The size of the common data area.
int[]	VALUE The value of the common data.
String[]	TYPE The type of the common data.
int[]	TIME_WRITTEN The time when the common data is written.
int[]	NUMBER_OF_WRITING_PROCESS The process that is writing the common data.
int[]	ADDITIONAL_DATA The additional data of the common data.

Method Summary	
void	setupCommonData (String value, String type, long timeWritten, int numberOfWritingProcess, int additionalData) Sets up common data.
int	getValue() Gets common data value.
String	getType() Gets common data type.
int	getTime() Gets the time when writing the common data value.
int	getProcessNumber() Gets the process number which writes the common data.
int	getAdditional() Gets the additional value.
void	addToCommonData(String value, String type, long timeWritten, int numberOfWritingProcess, int additionalData) Adds the value to common data.
void	getCommonDataTable() Gets the current value of common data table.
void	writeCommonDataValue(String value, String type, int processNumber,int additionalData) Writes the common data value.

String	readCommonDataValue() Reads the common data value.

Class Timer (Inner class of OS class)

Timer class inherits form Process class. In Timer class, there are a constructor and a process method.

Constructor Summary
Timer Creates a timer process that inherits from Process class.

Method Summary	
void	processMethod() Timer process's method. Maintains the timer table and sets flags for processes to run

Class MessageMonitor (Inner class of OS class)

MessageMonitor class inherits form Process class. In MessageMonitor class, there are a constructor and a process method.

Constructor Summary
MessageMonitor Creates a message monitor process that inherits from Process class.

Method Summary	
void	processMethod() Message monitor process's method. Checks for the arrival of messages.

Class PerformanceAnalysis (Inner class of OS class)

PerformanceAnalysis class inherits form Process class. In PerformanceAnalysis class, there are a constructor and a process method.

Constructor Summary
PerformanceAnalysis Creates a performance analysis process that inherits from Process class.

Method Summary

void	processMethod() Performance analysis process's method. Analyses data collected by performance probes

Class TimeoutReport (Inner class of OS class)

TimeoutReport class inherits form Process class. In TimeoutReport class, there are a constructor and a process method.

Constructor Summary
TimeoutReport Creates a timeout report process that inherits from Process class.

Method Summary	
void	processMethod() Timeout report process's method.

Class GarbageCollector (Inner class of OS class)

GarbageCollector class inherits form Process class. In GarbageCollector class, there are a constructor and a process method.

Constructor Summary
GarbageCollector Creates a garbage collector process that inherits from Process class.

Method Summary	
void	processMethod() Garbage collector process's method. Runs when time available to clean up heap.

Class Terminate (Inner class of OS class)

Terminate class inherits form Process class. In Terminate class, there are a constructor and a process method.

Constructor Summary
Terminate Creates a terminate process that inherits from Process class.

Method Summary	
void	processMethod() Terminate process's method. Disables timer interrupt and resets tables to stop scheduler.

Class Idle (Inner class of OS class)

Idle class inherits form Process class. In Idle class, there are a constructor and a process method.

Constructor Summary
Idle Creates an idle process that inherits from Process class.

Method Summary	
void	processMethod() Idle process's method.

Class Process

Process class contains a standard template for process methods and methods for working with processes.

Field Summary	
String	processName The name of the process.
int	synchronousWaitTime The wait time of the process.
int	synchronousWaitPhase The wait phase of the process.
int	timeout Number of ticks to be timed out
int	event1 The event that the process is waiting on.
int	event2 The other event that the process is waiting on.
int	message1 The message that the process is waiting on.
int	message2 The other message that the process is waiting on.

long	processExitAddress The exit address of the process.

Constructor Summary
Process(String procName, int synWaitTime, int synWaitPhase, int timeout, int event1, int event2, int message1, int message2, long processExitAddress) Creates a process.

Method Summary	
void	setProcessName(String processName) Sets the name of the process.
String	getProcessName() Gets the name of the process.
void	setSynchronousWaitTime(int waitTime) Sets the waitTime value of the process.
int	getSynchronousWaitTime() Gets the waitTime value of the process.
void	setSynchronousWaitPhase(int waitPhase) Sets the waitPhase value of the process.
int	getSynchronousWaitPhase() Gets the waitPhase value of the process.
void	setTimeout(int timeout) Set number of ticks to be timed out.
int	getTimeout() Get number of ticks to be timed out.
void	setEvent1(int event1) Sets the event that the process is waiting on.
int	getEvent1() Gets the event that the process is waiting on.
void	setEvent2(int event2) Sets the other event that the process is waiting on.
int	getEvent2() Gets the other event that the process is waiting on.
void	setMessage1(int message1) Sets the message that the process is waiting on.
int	getMessage1()

	Gets the message that the process is waiting on.
void	setMessage2(int message2) Sets the other message that the process is waiting on.
int	getMessage2() Gets the other message that the process is waiting on.
void	setProcessExitAddress(long processExitAddress) Sets the exit address of the process.
long	getProcessExitAddress() Sets the exit address of the process.
void	processMethod() The method of the process.

Class Application

In the Application class, there are Event Monitor process, Start Application process, Stop application process and user application processes. The Application class contains the code for a specific user task.

Class EventMonitor (Inner class of Application class)

EventMonitor class inherits form Process class. In EventMonitor class, there are a constructor and a process method.

Constructor Summary
EventMonitor Creates an event monitor process that inherits from Process class.

Method Summary	
void	processMethod() Event monitor process's method. Polls for i/o event.

StartApplication class (Inner class of Application class)

Constructor Summary
StartApplication Creates a start application process that inherits from Process class.

Method Summary

void	processMethod() Start application process's method. Sets up processes to get the application to be run by the OS

StopApplication class (Inner class of Application class)

Inner class of Application class

Constructor Summary
StopApplication Creates a stop application process that inherits from Process class.

Method Summary	
void	processMethod() Stop application process's method. Stops all the running processes.

Application1 class (Inner class of Application class)

Constructor Summary
Application1 Creates an application process that inherits from Process class.

Method Summary	
void	processMethod() The method of application 1 process.

Appendix B

Processes Provided with OS

1. Timer process

```
3375    class Timer extends Process{
3376        public Timer(String procName, int synWaitTime, int synWaitPhase,
3377        int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
3378            super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
3379            message1, message2, processExitAddress);
3380        }
3381
3382        public void processMethod(){
3383            //IF current time-previous time <> 1 tick THEN log error END
3384            if((getCurrentTime()-getPreviousTime())/40!=1){
3385                System.out.println("error occured:current time-previous time<>1");
3386            }
3387            //Previous time = current time
3388            refToOS.setPreviousTime(refToOS.getCurrentTime()); //Previous time=current time
3389            //for each process in scheduler table
3390            for(int i=1;i<refToScheduler.getProcessNumberInSchedulerTable();i++){
3391                int tickCount=refToScheduler.getTickCount(i);
3392                //Decrement tick count
3393                refToScheduler.setTickCount(i,--tickCount);
3394                //IF tick count <= 0 THEN
3395                if(refToScheduler.getTickCount(i)<=0&refToScheduler.getProcessNumber(i)!=0){
3396                    //Set execute flag in scheduler table
3397                    refToScheduler.setExecuteFlag(i,true);
3398                    //reset tick count to wait time
3399                    refToScheduler.setTickCount(i,refToScheduler.getWaitTime(i));
3400                }
3401            }
3402        }
3403    }
```

2. Start Application process

```
175     class StartApplication extends Process{
176         public StartApplication(String procName, int synWaitTime, int synWaitPhase,
177         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
178             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
179             message1, message2, processExitAddress);
180         }
181         public void processMethod(){
182             Application1 App1 = new Application1("Application1",1,1,2,0,0,0,0,0);
183             refToOS.passRef(App1);
184             passRef(App1);
185             refToProcessTable.loadProcess(App1);
186             refToScheduler.enableProcess(App1);
187             refToScheduler.runProcess(App1);
188         }
189     }
```

3. Message Monitor process

```
3457     class MessageMonitor extends Process{
3458         public MessageMonitor(String procName, int synWaitTime, int synWaitPhase,
3459         int timeout, int event1, int event2, int message1, int message2,
3460                 long processExitAddress){
3461             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
3462             message1, message2, processExitAddress);
3463         }
3464         public void processMethod(){
3465             //for each process in message table
3466             for(int i=0; i<refToMessage.getMessageNumberInMessageTable(); i++){
3467                 //IF wait flag is set and message has been sent THEN
3468                 if(refToMessage.getWaitingForMessage(i) & refToMessage.getMessageSent(i)){
3469                     //Set process execute flag in scheduler table
3470                     refToScheduler.setExecuteFlag(refToMessage.getToProcess(i),true);
3471                 }
3472             }
3473         }
3474     }
```

4. Performance Analysis process

```
3487     class PerformanceAnalysis extends Process{
3488         public PerformanceAnalysis(String procName, int synWaitTime, int synWaitPhase,
3489         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
3490             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
3491             message1, message2, processExitAddress);
3492         }
3493         public void processMethod(){
3494             refToScheduler.stopProcess(27);
3495             refToScheduler.disableProcess(27);
3496
3497             //On each run, reads data from circular buffer and produces analysis trace;
3498             //Displays results to and interacts with user.
3499             refToCircularBuffer.getPerformanceCircularBufferTable();
3500         }
3501     }
```

5. Stop Application process

```
244     class StopApplication extends Process{
245         public StopApplication(String procName, int synWaitTime, int synWaitPhase,
246         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
247             super(procName, synWaitTime, synWaitPhase, timeout,
248             event1, event2, message1, message2, processExitAddress);
249         }
250 
251         public void processMethod(){
252             //stop, disable and remove for each process in the application
253             refToScheduler.stopProcess(4);
254             refToScheduler.disableProcess(4);
255             refToProcessTable.removeProcess("Application1");
256 
257             //start a new process by setting a start application process to run
258             //or enabe terminate process to run
259             OS.Terminate terminate=refToOS.new Terminate("Terminate",1, 1, 2, 0, 0, 0, 0, 0);
260             refToOS.passRef(terminate);
261             refToProcessTable.loadProcess(terminate);
262             refToScheduler.enableProcess(terminate);
263             refToScheduler.runProcess(terminate);
264         }
265     }
```

6. Idle process

```
3568     class Idle extends Process{
3569         public Idle(String procName, int synWaitTime,
3570                 int synWaitPhase, int timeout, int event1,
3571                 int event2, int message1, int message2,
3572                 long processExitAddress){
3573             super(procName, synWaitTime, synWaitPhase,
3574                     timeout, event1, event2, message1,
3575                     message2, processExitAddress);
3576         }
3577 
3578         public void processMethod(){
3579             //say 1 tick = 20hundredthsec
3580             if(refToOS.getCurrentTime()-refToOS.getPreviousTime()>20){
3581                 timerInterruptHandler();
3582             }
3583             refToScheduler.runProcess(refToIdle);
3584         }
3585     }
```

7. Terminate process

```
3536      class Terminate extends Process{
3537          public Terminate(String procName, int synWaitTime, int synWaitPhase,
3538              int timeout, int event1, int event2, int message1, int message2,
3539              long processExitAddress){
3540              super(procName, synWaitTime, synWaitPhase, timeout,
3541                      event1, event2, message1, message2,
3542                      processExitAddress);
3543          }
3544
3545          public void processMethod(){
3546              //disable the timer interrupt
3547              enableTimerInterrupt(false);
3548
3549              //resets all execute flags in the scheduler table so that the scheduler stops
3550              for(int i=0;i<refToScheduler.getProcessNumberInSchedulerTable();i++){
3551                  refToScheduler.setExecuteFlag(i,false);
3552              }
3553          }
3554      }
```

8. Other processes

We are still working on the design of other processes, which are Event Monitor process, Timeout Report process and Garbage Collector process.

Appendix C

OS Kernel

Main method

```
//enables timer interrupt, enables initialize process and calls scheduler to start OS
public static void main(String[] args){
    //construct an instance of OS Class(this) class
    OS myOS=new OS();
    myOS.passRef(myOS);

    //construct an instance of Application class
    Application myApp=new Application();
    myOS.passRef(myApp);
    myApp.passRef(myOS);

    //Declares instances of all tables
    OS.ProcessTable processTable=myOS.new ProcessTable();
    myOS.passRef(processTable);
    myApp.passRef(processTable);
    OS.Scheduler scheduler=myOS.new Scheduler();
    myOS.passRef(scheduler);
    myApp.passRef(scheduler);
    OS.Event event=myOS.new Event();
    myOS.passRef(event);
    myApp.passRef(event);
    OS.Message message=myOS.new Message();
    myOS.passRef(message);
    myApp.passRef(message);
    OS.CircularBuffer circularBuffer=myOS.new CircularBuffer();
    myOS.passRef(circularBuffer);
    myApp.passRef(circularBuffer);
    OS.CommonData commonData=myOS.new CommonData();
    myOS.passRef(commonData);
    myApp.passRef(commonData);
    myOS.testCommonData();
```

```
        //Initialises all table values to zero
        processTable.setupProcessTable(0, "", false, 0, false, false, 0, false, false, false, "");
        scheduler.setupSchedulerTable(0, false, false, 0, 0, 0, false);
        event.setupEventTable(0, "", "", false, false, 0, "", "");
        message.setupMessageTable(false, 0, 0, false, false, "", "", 0, 0, false);
        circularBuffer.setupCircularBufferTable(false, 0, 0, 0, false, 0, 0);
        commonData.setupCommonData("", "", 0, 0, 0);

        //initialises the OS table
        myOS.setupOSTable(0, false, false, 0, false, 0, false, 0, 0, 0l, 0l, 0);

        //reads current time and sets to previous time to correctly initialise the timer
        myOS.setPreviousTime(myOS.getCurrentTime());
        //myOS.setClock(0);
        myOS.setStartTime(myOS.getCurrentTime());

        //constructs the OS processes-timer,garbage collector,timeout report,idle
        OS.Timer timer=myOS.new Timer("Timer", 1, 1, 2, 0, 0, 0, 0, 0);
        myOS.passRef(timer);


        OS.Idle idle=myOS.new Idle("Idle",1, 1, 2, 0, 0, 0, 0, 0);
        myOS.passRef(idle);

        //loads and enables the following processes, which are considered to be part of the OS
        processTable.loadProcess(timer);
        //processTable.loadProcess(garbageCollector);
        //processTable.loadProcess(timeoutReport);
        processTable.loadProcess(idle);
        scheduler.enableProcess(timer);
        //scheduler.enableProcess(garbageCollector);
        //scheduler.enableProcess(timeoutReport);
        scheduler.enableProcess(idle);

        //loads and enables the Performance Analysis Process
        OS.PerformanceAnalysis performanceAnalysis=myOS.new PerformanceAnalysis("PerformanceAnalysis",1, 1, 2, 0, 0, 0, 0, 0);
        myOS.passRef(performanceAnalysis);
        processTable.loadProcess(performanceAnalysis);


       //loads and enables the Start Application Process
       Application.StartApplication startApplication=myApp.new StartApplication("StartApplication",1, 1, 2, 0, 0, 0, 0, 0);
       myOS.passRef(startApplication);
       processTable.loadProcess(startApplication);
       scheduler.enableProcess(startApplication);

       //sets the starts Application process and the Idle process to run
       scheduler.runProcess(startApplication);
       scheduler.runProcess(idle);

       myOS.enableTimerInterrupt(true); //enables timer interrupts
       //myOS.setEnablePerformProbes(true); //enables performance probes
       myOS.setClock(0);
       scheduler.schedulerInfiniteLoop(); //call scheduler

       System.exit(0); //exit OS

   }
```

Scheduler loop

```
/* Scheduler-decides which process is to run and dispatches it */
private void schedulerInfiniteLoop(){
    // set process number to zero-loop invariant i<=number of processes
    // loop will only exit when a call to the terminate process resets
    // all execute flags in scheduler table
    int i=0;
    while(i<32){

        //if process i is ready to run
        if (EXECUTE_FLAG[i]==true){

            //reset process execute flag in scheduler table
            //something else has to set it
            EXECUTE_FLAG[i]=false;

            //Set timeout counter in OS table
            refToOS.setTimeoutCounter(2);

            //Set current process number in OS table
            refToOS.setCurrentProcessNumber(i+1);


            //if performance testing flag set in OS table
            if (refToOS.getEnablePerformProbes()==true){
                //then call probe method
                performanceProbe();
            }

            //call process,  start process and pass state to it
            //process executes and returns to here
            switch (i) {
                case 0:  refToTimer.processMethod();
                    break;
                case 1:  refToEventMonitor.processMethod();
                    break;
                case 2:  refToStartApplication.processMethod();
                    break;
                case 3:  refToApplication1.processMethod();
                    break;
                case 4:  refToApplication2.processMethod();
                    break;
                case 5:  refToApplication3.processMethod();
                    break;
                case 6:  refToApplication4.processMethod();
                    break;
                case 7:  refToApplication5.processMethod();
                    break;
                case 8:  refToApplication6.processMethod();
                    break;
                case 9:  refToApplication7.processMethod();
                    break;
                case 10: refToApplication8.processMethod();
                    break;
                case 11: refToApplication9.processMethod();
                    break;
                case 12: refToApplication10.processMethod();
                    break;
                case 13: refToApplication11.processMethod();
                    break;
                case 14: refToApplication12.processMethod();
                    break;
```

```
                    case 15: refToApplication13.processMethod();
                        break;
                    case 16: refToApplication14.processMethod();
                        break;
                    case 17: refToApplication15.processMethod();
                        break;
                    case 18: refToApplication16.processMethod();
                        break;
                    case 19: refToApplication17.processMethod();
                        break;
                    case 20: refToApplication18.processMethod();
                        break;
                    case 21: refToApplication19.processMethod();
                        break;
                    case 22: refToApplication20.processMethod();
                        break;
                    case 23: refToApplication21.processMethod();
                        break;
                    case 24: refToApplication22.processMethod();
                        break;
                    case 25: refToMessageMonitor.processMethod();
                        break;
                    case 26: refToPerformanceAnalysis.processMethod();
                        break;
                    case 27: refToTimeoutReport.processMethod();
                        break;
                    case 28: refToGarbageCollector.processMethod();
                        break;
                    case 29: refToStopApplication.processMethod();
                        break;
                    case 30: refToIdle.processMethod();
                        break;
                    case 31: refToTerminate.processMethod();
                        break;

                    default: System.out.println("Invalid process number in scheduler loop.");
                    break;
                }

            //(if timeout not disabled by process potential
            //for error if timer interrupt occus here)
            //DISABLE timeouts
            //should have been done by process-belt and bracers
            refToOS.setEnableTimeoutsFlag(false);

            //if performance testing flag set in OS table
            if (refToOS.getEnablePerformProbes()==true){
                //then call probe method
                performanceProbe();
            }
        }
        i++;
        if((i==31)&(EXECUTE_FLAG[31]==false)){
            i=0;
        }
    }
```

Timer interrupt simulation

```
class Idle extends Process{
    public Idle(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 processMethod(){
        //say 1 tick = 20hundredthsec
        if(refToOS.getCurrentTime()-refToOS.getPreviousTime()>20){
            timerInterruptHandler();
        }
        refToScheduler.runProcess(refToIdle);
    }
}
```

Appendix D

Test Applications

1. Test that OS runs, tests scheduler, runs a single application process that prints out table contents

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260 
261             refToOS.getOSTables(); //prints out all the OS tables
262 
263             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
264         }
265     }
```

2. Test each OS process –Timer process etc that they give expected results

see Section7.2.2

3. Test timeout

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260 
261             for(;;){
262                 System.out.println("Application1 is running!");
263             }
264 
265             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
266         }
267     }
```

4. Test each data communication method: messages, circular list, and common data

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260             int randomNumber=getRandomNumber(); //gets a random number
261 
262             //gets and sends a message
263             int messageNumber=0;
264             messageNumber=refToMessage.getMessage("Application1","Application2","int");
265             refToMessage.sendMessage(messageNumber,"Application2",String.valueOf(randomNumber));
266             //adds to circular list
267             refToCircularBuffer.addToCircularList(String.valueOf(randomNumber),4);
268             //writes to common data area
269             refToCommonData.writeCommonDataValue(String.valueOf(randomNumber), "String", 4, 0);
270 
271             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
272         }
273         public int getRandomNumber(){
274             Random generator=new java.util.Random();
275             int randomNumber=generator.nextInt();
276             return randomNumber;
277         }
278     }
```

```
280     class Application2 extends Process{
281         public Application2(String procName, int synWaitTime, int synWaitPhase,
282         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
283             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
284             message1, message2, processExitAddress);
285         }
286         public void processMethod(){
287             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
288 
289             refToMessage.receiveMessage("Application2"); //reveives the message
290             refToMessage.releaseMessage(1); //release the message
291             refToCircularBuffer.removeFromCircularList(5); //removes from circular list
292             refToCommonData.readCommonDataValue(); //reads from common data
293 
294             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
295         }
296     }
```

5. Test performance probes

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260 
261             refToOS.performanceProbe();
262             refToCircularBuffer.getCircularBufferTable();
263 
264             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
265         }
266     }
```

6. Test multiple processes, including process to print out tables

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260             int randomNumber=getRandomNumber(); //gets a random number
261 
262             //gets and sends a message
263             int messageNumber=0;
264             messageNumber=refToMessage.getMessage("Application1","Application2","int");
265             refToMessage.sendMessage(messageNumber,"Application2",String.valueOf(randomNumber));
266             //adds to circular list
267             refToCircularBuffer.addToCircularList(String.valueOf(randomNumber),4);
268             //writes to common data area
269             refToCommonData.writeCommonDataValue(String.valueOf(randomNumber), "String", 4, 0);
270 
271             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
272         }
273         public int getRandomNumber(){
274             Random generator=new java.util.Random();
275             int randomNumber=generator.nextInt();
276             return randomNumber;
277         }
278     }
```

```
280     class Application2 extends Process{
281         public Application2(String procName, int synWaitTime, int synWaitPhase,
282         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
283             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
284             message1, message2, processExitAddress);
285         }
286         public void processMethod(){
287             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
288 
289             refToMessage.receiveMessage("Application2"); //reveives the message
290             refToMessage.releaseMessage(1); //release the message
291             refToCircularBuffer.removeFromCircularList(5); //removes from circular list
292             refToCommonData.readCommonDataValue(); //reads from common data
293 
294             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
295         }
296     }
```

```
309     class Application3 extends Process{
310         public Application3(String procName, int synWaitTime, int synWaitPhase,
311         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
312             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
313             message1, message2, processExitAddress);
314         }
315         BitPort bp = new BitPort(BitPort.Port3Bit5);
316         public void processMethod(){
317             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
318             int i=0;
319             while(i<1){
320                 i=i+1;
321                 //Turn on LED
322                 bp.clear()
323                 //Turn off LED
324                 bp.set
325             }
326             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
327         }
328     }
```

```
328     class Application4 extends Process{
329         public Application4(String procName, int synWaitTime, int synWaitPhase,
330         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
331             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
332             message1, message2, processExitAddress);
333         }
334         public void processMethod(){
335             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
336             refToOS.getOSTables(); //prints out all the tables
337             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
338         }
339     }
```

7. Test performance analysis process

```
252     class Application1 extends Process{
253         public Application1(String procName, int synWaitTime, int synWaitPhase,
254         int timeout, int event1, int event2, int message1, int message2, long processExitAddress){
255             super(procName, synWaitTime, synWaitPhase, timeout, event1, event2,
256             message1, message2, processExitAddress);
257         }
258         public void processMethod(){
259             refToOS.setEnableTimeoutsFlag(true); //enables timeouts for duration of process
260 
261             refToScheduler.runProcess(refToOS.refToPerformanceAnalysis);
262 
263             refToOS.setEnableTimeoutsFlag(false); //disables timeouts
264         }
265     }
```