Real-Time Operating System With Non-Real-Time Simulation For the Power PC

by

Binh C. Truong

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2000

© 2000 Binh C. Truong. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Signature of Author

Department of Electrical Engineering and Computer Science

May 19, 2000

Certified by

Roger Racine
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by

Nancy Leveson
T Professor
Thesis Advisor

Accepted by

Chairman, Department Committee on Graduate Theses
Real-Time Operating System With Non-Real-Time Simulation For the Power PC

by

Binh C. Truong

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 2000
in partial fulfillment of the requirements for the Degrees of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis describes the implementation of a real-time operating system for the Power PC processor. This implementation includes a non-real-time simulation for the Linux host system. The simulation system provides for a variety of capabilities, such as task scheduling. In particular, the rate monotonic and earliest deadline first scheduling algorithms are examined.

Technical Supervisor: Roger Racine
Title: Draper Laboratory Technical Staff

Thesis Advisor: Nancy Leveson
Title: MIT Professor
ACKNOWLEDGEMENT
(June 2000)

I would like to express my appreciation to Roger Racine of Draper Lab for his technical supervision over the course of the project, and to Professor Nancy Leveson for her support as the advisor to this thesis.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Draper Laboratory IR&D Project, Common Core Software Architecture; Project Number 15022, Contract Number 00-1-1047.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(author's signature)
## CONTENTS

1 INTRODUCTION .......................................................................................................................... 6

   1.1 OPERATING SYSTEMS ........................................................................................................... 6
       1.1.1 Overview ....................................................................................................................... 6
       1.1.2 Examples ....................................................................................................................... 7

1.2 REAL-TIME SYSTEMS ........................................................................................................... 7

   1.2.1 Overview ......................................................................................................................... 8
   1.2.2 Scheduling ...................................................................................................................... 9

1.3 DRAPER LAB REQUIREMENTS ............................................................................................... 11

   1.3.1 Sample Real-Time Operating Systems ............................................................................ 12
   1.3.2 Real-Time Operating System for Draper Lab ................................................................. 12
   1.3.3 The Ada Programming Language .................................................................................... 13

1.4 SCHEDULING TECHNIQUES FOR THE OS ............................................................................ 14

   1.4.1 Rate Monotonic .............................................................................................................. 14
   1.4.2 Earliest Deadline First ..................................................................................................... 15

2 REAL-TIME OPERATING SYSTEM .............................................................................................. 17

   2.1 TOP LAYER ADA ARCHITECTURE ....................................................................................... 17

       2.1.1 General implementation ............................................................................................... 18
       2.1.2 Overview of Ada Packages for Runtime System .......................................................... 18

2.2 POWERPC SPECIFICATIONS ................................................................................................. 20

       2.2.1 Assembly Language File ............................................................................................. 20
       2.2.2 Auxiliary C File ............................................................................................................ 20

2.3 EXCEPTION HANDLING FOR RTOS ..................................................................................... 21

       2.3.1 Interrupts ...................................................................................................................... 21
       2.3.2 Exceptions .................................................................................................................... 22

2.4 COMPILATION PROCESS ....................................................................................................... 23

2.5 CURRENT STATUS OF RTOS ............................................................................................... 23

3 NON-REAL-TIME SIMULATION .................................................................................................. 26

   3.1 SPECIFICATIONS .................................................................................................................. 26
   3.2 IMPLEMENTATION .............................................................................................................. 27

       3.2.1 Simulated Clock ............................................................................................................ 27
       3.2.2 Scheduler ..................................................................................................................... 34

4 CONCLUSION .............................................................................................................................. 45

   4.1 SUMMARY ........................................................................................................................... 45
   4.2 FURTHER WORK ................................................................................................................... 45

5 REFERENCES ............................................................................................................................... 47
1 Introduction

The operating system serves as the system program that interfaces the computer hardware with the application software. The operating system is loaded into the computer's RAM through a bootstrap program, after which the OS takes command of managing all other application programs [11].

This thesis will discuss the purpose and method of implementing a real-time operating system (RTOS) on the Power PC, including the implementation of a non-real-time simulation system with task scheduling capabilities. The main goal is to start with an existing operating system that was developed for another processor, and customize it to work with the PowerPC. Several features can be implemented in the system, such as multiprogramming, snapshot-rollback, fault tolerance, and a cyclic task architecture. An important aspect to the thesis is the task architecture consisting of the scheduler, which can be implemented in a variety of ways.

1.1 Operating Systems

1.1.1 Overview

The operating system for a computer serves as the connection between the applications programmer and the hardware. The hardware-level system generally consists of the central processing unit (CPU), the memory, and the input/output (I/O)
devices, while the programmer defines and uses such programs as compilers, databases, games, and other applications. The operating system, then, allows the programmer to utilize the services provided by the hardware in a convenient and efficient way [11].

1.1.2 Examples

A wide variety of operating systems are available for the different types of computer systems. These include commercial systems such as Microsoft Windows and UNIX, and open-source systems such as Linux and NetBSD. The choice of which operating system to use depends on several factors, such as cost, adaptability, quality of support, and purpose of use. For instance, a system that includes professional support usually comes with high cost and limited flexibility, while a customizable, open-source system usually has no type of support. Furthermore, some applications require specialized features that must be included with the operating system. One specific example is a real-time system, which is a system that not only needs to produce correct results in the logical sense, but must also adhere to particular time requirements.

1.2 Real-Time Systems

There are numerous examples of systems that are considered real-time. For instance, a navigation system on a vehicle might need to continuously monitor certain data obtained from its external sensors. The data is then analyzed to determine whether any action needs to take place, such as adjusting the vehicle’s position, or updating a console display. It is important that the data analysis is done in real-time, since a lag in
the data collection could result in mal-function of the system. Thus, the navigation system must use an operating system that is able to support real-time features so that the appropriate actions can take place within a specific time interval. In order to accurately distinguish between real-time and non-real-time systems, we can define a real-time system to be one that consists of a deadline, which is the time that the system must meet in completing its execution. Furthermore, a cyclic (or periodic) system is one that repeats itself every time period. This type of system can be said to have an execution start time, which corresponds to its previous deadline. Figure 1 illustrates the typical characteristics of a cyclic system [5].

![Figure 1 Timing Characteristics of a Cyclic System](image)

### 1.2.1 Overview

Real-time systems can be divided into two main categories according to how they consider timing for their tasks, which we define as any computational activity that can be
scheduled and executed by the CPU. The first type is hard real-time systems, which must meet all timing constraints for their tasks in order to insure proper functionality. If a task misses its deadline, then a system failure has occurred. Such examples include nuclear power plants, military command and control systems, and navigation systems. The second type is known as a soft real-time system, in which finishing the task by the soft deadline is advantageous, but missing the deadline does not cause system failure.

A real-time operating system is required to support the various types of time-constrained systems discussed above, especially those that have hard real-time constraints. On the other hand, those that resemble soft real-time systems may be able to use a general-purpose operating system. In order to satisfy the needs of hard real-time systems, real-time operating system must have additional features not normally found in the common operating system. One example is the ability to provide multi-tasking. However, due to the existence of timing constraints, one task might need to completely take over the CPU in order to meet a deadline. Thus, the operating system must have an appropriate scheduler that allows an important task to run, while stalling a less important task until resources are available for that task to start running.

1.2.2 Scheduling

The scheduler is one of the most crucial components of a real-time operating system. There are many implementations for a scheduling scheme, each with its own advantages. The simplest type is first-come, first-served scheduling, in which the oldest task completely finishes executing before the next available task can begin to execute. Another implementation involves shortest-job-first scheduling. This type of scheduler
tries to ensure that the average waiting period for all tasks is as short as possible. A third approach to scheduling involves a priority-based preemptive algorithm. The idea behind this method is that the most important tasks should always be executed first, while less critical tasks can wait in the background until resources are available. A final approach, known as round-robin scheduling, is similar to first-come first-served, but instead of allowing a task to finish executing, each task is assigned a period of time for executing, known as a time quantum or time slice. Once a task’s time quantum is used up, it is stalled until the rest of the tasks have taken their turn. A combination of these methods can be used to implement a scheduler. For example, if priority-based scheduling is used and there are two or more tasks that have the same priority, then the scheduler can use the first-come first-served approach for these particular tasks [11].

The scheduling of tasks in real-time systems depends on the type of system that is being used. For instance, a system might involve only tasks that are periodic, whose initial deadlines and periods are known. Another system might include both periodic and aperiodic, or sporadic, tasks. These sporadic tasks do not necessarily occur at a specific time or frequency, although it is usually known that a sporadic task will definitely occur once during a time interval, and will occur again at the next interval. Given these types of systems, there are two common scheduling schemes that are commonly used, the rate monotonic algorithm (RMA) and earliest deadline first (EDF).

The rate monotonic scheduler uses the priority-based algorithm to assign priorities. For tasks that are periodic, it assigns the highest priority to the task with the shortest period, and continues to assign decreasing priority to longer-period tasks until the task with the longest period has been assigned the lowest priority. Aperiodic tasks can be handled in a couple of ways. For example, all aperiodic tasks can run in the background, and when resources are free (i.e. all periodic tasks are finished executing) then the aperiodic tasks can start to execute. Another method is to assign an aperiodic
task a certain time slice and to allow preemption to take place. This aperiodic task can then preempt a periodic task if all timing constraints can be met [10].

The earliest deadline first algorithm also uses priority-based scheduling, but in general does not worry about differentiating between periodic and aperiodic tasks. Instead, the algorithm simply assigns the highest priority to the task with the earliest deadline in the current time frame. Since deadlines can change over time, EDF assignment must be done dynamically, whereas static assignment is performed with rate monotonic scheduling. Hence, rate monotonic scheduling is usually easier to implement and analyze, and thus is more widely used [5].

1.3 Draper Lab Requirements

Many Draper Laboratory projects require the use of a real-time operating system. However, the choice of which operating system to use involves many of the factors discussed previously, such as cost, adaptability, and availability of support. A variety of commercial off-the-shelf (COTS) real-time operating systems exist in the market today, but they contain a number of deficiencies. These deficiencies lead to decreased productivity in performing embedded software development projects at Draper Laboratory. For instance, one major deficiency is the lack of host-based simulation of the operating system. The typical development process at Draper Laboratory starts with software being implemented on a host, in which there is a simulation environment and debugging tools to perform integration and testing. This usually requires the ability to stop at a breakpoint, to restart, and to single-step through the software program. Once the
software is completed on the host, it can then be compiled, integrated, and tested on the target.

1.3.1 Sample Real-Time Operating Systems

Several real-time commercial and open-source operating systems are considered. The open-source operating systems are based on the gcc compiler. One such operating system is called eCos, which is written in C++ and is supported by Cygnus. It is highly configurable, but has neither Ethernet capability nor a simulation system. ACT supports the Ada version of the gcc compiler, called GNAT. Since the Ada language supports multi-tasking, GNAT includes a tasking kernel. RTEMS, by OAR Corp., is written in C and is supported for C and Ada. However, it is not configurable, does not have Ethernet capability, and has no simulation system. The commercial operating system, VxWorks, supports multiprocessor systems and has a non-real-time version that is available on Sun and HP computers. However, it is very expensive to acquire the source to its kernel. Given the deficiencies of the open-source operating systems and the costliness of commercial systems, it has been determined that an existing open-source real-time operating system should be used as the starting point, and missing features should be added to the operating system to eliminate deficiencies.

1.3.2 Real-Time Operating System for Draper Lab

Given Draper Lab’s requirements for an operating system, the primary goal of this project is to implement an operating system that can be freely used in the public
domain, in which there is the availability of the source code and the allowance to modify the code. In terms of its features, the operating system will contain a variety of useful capabilities such as priority scheduling for real-time systems, host-based simulation for testing and debugging, multiprogramming ability, and fault tolerance. Furthermore, a non-real-time version will be created that allows for simulations and for deliverable software to run.

1.3.3 The Ada Programming Language

The real-time operating system used in this project is written in the Ada programming language. This language is particularly appropriate in this case due to its strong typing and its built-in real-time capabilities that are exploited in the development of the real-time operating system.

To briefly describe the language, Ada programs are composed of one or more entities called packages, each of which consists of one or more related functions. Each package generally contains two parts, a body and a specification. The body contains the function definitions for the package, and the specification defines the interface for functions that are called by other packages. Finally, there is a main program that uses the functions defined by the packages, and it is this program that is compiled and linked with the packages to produce an executable file.

One feature of Ada that is usually not included in other languages is tasking. Tasking provides the ability to build concurrent systems in an efficient manner. The Ada tasking construction allows the user to define a set of tasks and to work with various features such as task priority and communication among tasks [3].
1.4 *Scheduling Techniques for the OS*

As discussed previously, the scheduler is a crucial part of a real-time operating system that allows for multi-tasking. For the purposes of this project, the scheduler is implemented and analyzed using the two most common algorithms, rate monotonic and earliest deadline first. The real-time features of the run-time environment, such as priority tasking and concurrency, are used to implement the scheduler. The scheduler is tested using a set of sample tasks gathered from representative real-time systems, such as an inertial guidance system.

A non-real-time version of the Ada run-time system is implemented that allows for testing and debugging on the host machine. This system has a simulated clock that is updated only when all tasks are blocked, and any task that has reached its wake-up time will be released. Thus, the scheduler is first implemented and tested on this simulation system, and then ported to the real-time system.

1.4.1 Rate Monotonic

As mentioned previously, the rate monotonic algorithm is a static procedure that assigns a priority to each task according to task period; tasks with shorter periods receive higher priorities. One major advantage of this algorithm is that the schedulability of a set of tasks can be determined before actual execution, given that the execution time is
known beforehand. The following theorem determines whether all tasks in a set can meet their deadlines:

A set of $n$ independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines if:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n\left(2^{1/n} - 1\right)U(n)$$

where $C_i$ and $T_i$ are the execution and period of task $\tau_i$ respectively.

The value $\sum C_i/T_i$ represents the fraction of time that the processor requires to execute tasks $\tau_i$. This value is termed utilization $U(n)$, and its bound approaches 69% ($\ln 2$) as the number of tasks, $n$, approaches infinity. Thus, processor utilization should generally be less than 69% for successfully scheduling of tasks [10].

1.4.2 Earliest Deadline First

The earliest deadline first algorithm also deals with a set of independent tasks. However, it sometimes uses a dynamic procedure to assign priorities to the tasks. The advantage to this scheme is that it allows for new tasks to be introduced to the original set of tasks. This, however, requires the scheduler to continuously monitor and update the task priorities. Thus, one possible disadvantage is that the scheduling process cannot be entirely accomplished in the beginning of the program execution. This might arise in a situation where a task has varying deadlines, or in a case where new tasks are being introduced into the system after the system has started executing, thus creating the need for the scheduler to continue scheduling during system execution. If there is a need to simplify the system, however, various assumptions can be made about the tasks to
guarantee that deadlines are met. For instance, it can be specified that if all incoming tasks have a deadline later than a set deadline, then those tasks can be scheduled successfully.

One basic approach to this algorithm is to put the tasks in order of non-decreasing deadlines, and to assign the higher priorities to the earlier deadlines. Since the deadlines will be updated after every task execution, the scheduler must determine the appropriate time to update the priorities according to the new deadlines. A more complicated case involves not only the changing deadlines of existing tasks, but also the arrival of new tasks that must be factored into the scheduling process [5].

The choice of whether to use a static or dynamic version of the earliest deadline first algorithm depends on whether all timing parameters are known before the system starts executing. Furthermore, the decision of which scheduler to use becomes more complicated with the addition of the rate monotonic algorithm. A comparison of these various implementation techniques are discussed later in this paper.
2 Real-Time Operating System

2.1 Top Layer Ada Architecture

The starting point for the operating system is a kernel that was designed by Top Layer Networks, Inc. for use in their data switch. The kernel is written in the Ada programming language and is designed to execute on a bare-machine with no existing operating system. Features that are supported include priority-based preemptive scheduling, time-slicing within priority, and execution of modules written in C. The main components of the kernel are thread management, process management, interrupt and exception handling, memory management, and time operations. The general architecture of the TopLayer Ada runtime system is illustrated in Figure 2.

![Figure 2 Architecture of the Ada Runtime System](image_url)

(From Top Layer Networks [6])
2.1.1 General implementation

The Ada run-time system is modified from the existing GNAT Ada system to exclude features that were not needed for the data switch, but can still be supported by the GNAT compiler interface and debugger. Thus, the interfaces that are kept in the system include, protected objects, exceptions, and the standard library and parameters. Additional characteristics of the run-time system include multi-tasking features and memory management. For instance, in the task management area, thread control blocks are physically separated from the Ada task control block so that the underlying kernel can support both Ada tasks and C threads. Also, both dynamic memory management and dynamic allocation of protected objects are permitted, while dynamic creation and termination of tasks can be implemented.

The original target for the operating system was the ARC processor. Therefore, all the assembly code was written and tailored for this specific processor. Modifications are made to the assembly code files to make it compatible with the PowerPC processor. Such modifications include interrupt-handling routines, register manipulation, and low level programming changes to the kernel. However, very little, if any, of the Ada run-time system routines were changed in order for the operating system to start running on the Power PC. A more detailed explanation of the system and major modifications to the system are presented in the next sections.

2.1.2 Overview of Ada Packages for Runtime System

The files for the runtime system are grouped into two main layers, the Ada-dependent layer that supports much of the Ada programming features, and the underlying
kernel layer which handles the low-level operations. The Ada-dependent layer includes the following packages:

System.Tasking System.Tasking.Stages
System.Tasking.Protected_Objects
System.Task_Specific_Data System.Task_Info
Ada.Exceptions System.Exceptions
System.Exception_Table System.Tasking_Soft_Link
System.Standard_Library System.Parameters

The layer consisting of the underlying kernel includes the following packages [6]:

- Kernel.Threads, which performs thread management in a processor independent manner.
- Kernel.Process, which performs process management in a processor independent manner.
- Kernel.CpuPrimitives, which encapsulates all the machine dependent operations to handle machine state and interrupts and acts as the lowest layer in Ada runtime system. Its specification is known only to the underlying kernel.
- Kernel.Memory, which performs real memory management.
- Kernel.Time, which provides low-level time type and operations.
- Kernel.Parameters, which contains constants for use only by the Kernel layer.
- Kernel.Exceptions, which performs low-level exception management.
- Kernel.Crash, which logs significant machine state before resetting.
- Kernel.IO, which is the interface to a flash disk device.

A few of these files, Kernel.CpuPrimitives in particular, are modified to run correctly in the PowerPC processor. These changes include exception and interrupt routines, register settings, and other low-level function calls.
2.2 PowerPC Specifications

2.2.1 Assembly Language File

There are several elements of the Top Layer Ada architecture that are processor-dependent. Thus, in order to port the operating system to the PowerPC processor, some of these low-level elements are also modified. However, the files containing the low-level code for the Top Layer RTOS (assembly files) are designed for the ARC processor, not for the PowerPC. Therefore, the main source for these files originates from a PowerPC-specific Linux kernel, which is contained in an assembly language file named head.s. This file includes most of the low-level function calls required by the RTOS, such as handling interrupts, saving registers during various exceptions, and restoring the processor to its original state after an exception. The details of how these functions work can be obtained by examining the RTOS kernel files described in the previous section.

2.2.2 Auxiliary C File

Because the Ada architecture uses a few C library routines, the cross-compilation process (described later) requires that these libraries be present in order for a final executable to result from the compilation. Some system calls, though, are not built into libc.a, which is the C library file that is included in the compilation process. The reason is that they are too dependent upon the particular target processor board that is being used. Therefore, a separate file, named functions.c, contains these missing library routines. In general, many of these function calls are not needed by the RTOS, so
they are included in functions.c as null functions so that the compiler does not generate errors concerning missing functions

2.3 Exception Handling for RTOS

As an example of how these packages provide the low-level functionality for the runtime system, the steps in processing interrupts and exceptions are examined. The PowerPC processor groups interrupts and exceptions into distinct categories: interrupts include External, Decrementer, and System Management, while exceptions include Machine Check, System Reset, Instruction-Caused, DSI, and Floating-Point [7].

2.3.1 Interrupts

The following process illustrates how an interrupt is handled:

-- The procedure Kernel-threads.AttachHandlers needs to be called first. All of the user-defined handlers are attached in this function.

-- The procedure Kernel-threads.Initialize needs to be executed (by a 'with' in the main program). This allows the appropriate interrupt handler to be called, after which a function pointer is assigned to the handler.

Stepping through an External Interrupt:

1) Hardware detects the interrupt and goes to vector address 0x0500

2) At "External" label, the code begins to initialize the machine’s state by saving the interrupt registers.

3) At this point, a thread is created for this handler. The thread is then put onto a queue to wait for execution.

4) Once the interrupt thread has finished executing, the return process takes place with a call to the return_from_int assembly language procedure.
5) Various registers, such as link and floating point registers, are restored.

6) rfi is called to signal a return from interrupt/exception.

2.3.2 Exceptions

Using DSI as an example, the following illustrates the handling of an exception:

1) Start at vector address (0x0300 for DSI)

2) Branch to DSI, which does the machine initialization by saving the exception registers, and then branches to s-stalib.Raise_DSI_Error.

3) Raise_DSI_Error then makes several calls.
   a) KE.CaptureState fills the stack with the values of the exception registers.
   b) The currently running thread is accessed by several functions which examine its data record, and determines if the thread data needs to be updated.
   c) Finally, KE.RaiseExcept is called, which activates the following procedures:
      i) KCD.LogError is called, which updates the error count.
      ii) builtin_longjmp (imported from C) is called, which jumps to the address pertaining to the appropriate exception handler.
      iii) If no handlers exist, then KCP.ResetProcessor is called. Otherwise the appropriate handler is activated.

4) Once the exception handling is finished, the return process takes place with a call to the return_from_excp assembly language procedure.

5) Various registers, such as link and floating point registers, are restored.

6) rfi is called to signal a return from interrupt/exception.
2.4 Compilation Process

Compiling the entire operating system into an executable file is made easier by creating a directory that contains all of the necessary files. These files include the Ada system files, the low-level kernel files, the assembly language file, and any auxiliary files created by the user, such as functions.c. At this point, the compiler can be run with the command:

`power-eabi-ada-gnatmake -g -a -02 -gnatapgv -f -nostdinc -I- -I. -v gmain -bargs -f -largs -v head.o functions.o`.

To briefly explain the command, `power-eabi-ada-gnatmake` invokes the cross-compiler, which compiles the source code into PowerPC machine code. The additional command options allow the compiler, binder, and linker to perform various optimizations. For instance, the compilation options allow for debugging capabilities, compilation of all system files, and ignoring system files that are included in the default directory. The other options indicate to the binder that detailed messages should be displayed upon encountering an error, and that other object files should be searched during the linking process.

2.5 Current Status of RTOS

A simple main program, named `gmain`, is created to test the RTOS and is compiled with the rest of the operating system files. The `gmain` program simply attempts to print out a line of text. After obtaining an Executable and Linking Format 32-bit, Most-Significant-Bit PowerPC executable `gmain` file in the compilation process, a
VME board with the PowerPC is setup to run the executable. However, before downloading it to the processor, the file has to be put into the correct format by using the `powerpc-rtems-objcopy` command. Once this is accomplished, the executable is loaded and the Motorola internal debugger is used to step through the machine code line by line.

At this point, the test program is debugged with the PowerPC internal debugger mentioned previously. A higher-level Ada debugger to be used with the PowerPC is currently being developed, which should greatly speed up the debugging process. By stepping through the machine code using the PowerPC debugger, it is found that the test program does not finish its execution. This program halt seems to be caused by a couple of errors in the machine code, which are explained below.

The first error involves the initialization process that the Ada run-time environment goes through before reaching the main program. The process includes setting Ada tags, checking the stack, performing elaboration, and going through kernel processes such as threads and memory. The entire initialization process, however, is not performed. Instead, it stops at the kernel process of memory elaboration. It then goes to a procedure called `kernel_crash_dump`, which is an indication that there is an ‘Illegal memory configuration,’ or that a memory address is null. At this point, it is difficult to discover the source and reason for the error that is causing `kernel_crash_dump` to be called. Hopefully, the high-level Ada debugger can be implemented and used to debug this particular error.

The second error comes from calling `kernel_crash_dump`, which eventually calls `Shutdown_Hardware_Print`. This procedure, which simply prints
“shutdownhardware_print procedure” to the screen, was added to notify the user that a


 crash-dump had occurred. However, when attempting to execute this procedure, the


 processor gives back an “Alignment Exception” for the instruction ‘Lfd fR0, $0, (R3).’ Thus, the print procedure has an error itself, and will need to be debugged with


 the Ada debugger.
3 Non-Real-Time Simulation

As described previously, the purpose of the non-real-time simulation for the real-time operating system is to test the implementation of the components for the RTOS, namely the scheduler as it relates to the thesis. The development of this simulation system is done on the Linux host, using the standard Ada libraries (libada) that have the same functionalities as those for the RTOS.

3.1 Specifications

The simulation system consists of a non-real-time mode that allows all services to have identical functionalities as in the real-time case except that time is simulated, i.e. there is a simulated clock. When running in simulated clock mode, the simulated clock is incremented by a configurable number of microseconds (100 microseconds - 1 second) each time all application tasks have become blocked and are waiting for their next execution start time. Additionally, after updating the simulated clock, the tasks that have reached their execution start time are released for execution as if the actual time had become the simulated clock’s value.

If all tasks are still blocked after the simulated clock has been updated, the clock will be incremented again, and those tasks which have reached their execution start times will be released for execution. All tasks are allowed to run until they become blocked, using a priority-based, non-preemptive scheduler with priority ceiling protocol for locks.
Thus, the system will look to the software as if the processor speed is infinite (a cyclic loop will take no time to execute).

**3.2 Implementation**

The simulation system allows the user to create application tasks either in Ada or in the C programming language. Although the construction of the tasks is similar, the user must be conscious of the slight differences in the creation and management of tasks in each language. The implementation details of the simulation system, along with the differences in Ada and C, are described below.

### 3.2.1 Simulated Clock

Given a set of application tasks created by the user, a queue is created to put these tasks in order of priority. On every clock increment, or tick, the queue is checked for tasks that have reached their execution start time. The tasks that are ready for execution are dequeued and released for execution, in order of priority. After execution, the tasks that are periodic are put back into the queue again. The next sections describe the specific Ada packages that implement this system.

#### 3.2.1.1 Ada Packages

There are several Ada packages that make up the simulation system. Each package consists of a group of related functions or tasks that perform a part of the simulation.
3.2.1.1 Clock Packages

The first set of packages is concerned with updating the simulated clock. These include `sim_clock`, `sim_clock_manager`, and `sim_clock_ticker`. Figure 3 illustrates the basic interaction between the clock packages and the tasking architecture of the Ada run-time system.

![Figure 3 Clock Packages](image)

The package `sim_clock` defines the types for several clocking elements, such as the unit of time and the priority queue. In `sim_clock_manager`, a task called `clock_manager` is created and is assigned the highest possible priority. Once this task is activated during the execution of the main program, it waits in a suspended mode at an entry (an Ada tasking function) named `clock_tick`. It becomes activated when this
entry is called by sim_clock_ticker. Once clock_manager becomes unsuspended, it increments the simulated clock by one time unit, as defined in sim_clock. It then examines the priority task queue to determine which task to release for execution. Since the queue is ordered by decreasing priority, clock_manager loops through the queue from head to tail and releases the tasks whose execution start times are equal to or less than the current simulated clock time. Once clock_manager has finished its job, it once again becomes suspended at the clock_tick entry and waits for the entry call. The reason the clock_manager task is assigned the highest possible priority is that once this task starts running (i.e. starts updating the simulated clock), we do not want another task to preempt it.

In contrast with the clock_manager task, the sim_clock task in sim_clock_ticker is assigned the lowest priority since we want all the application tasks to be blocked (queued) before incrementing the simulation clock. Therefore, since sim_clock has the lowest priority, it will not run until all application tasks are blocked. Once this occurs, sim_clock will call the entry clock_tick, which activates clock_manager, as described previously. After this entry call, sim_clock then goes back to waiting until all other application tasks are blocked.

3.2.1.1.2 Pthreads Interface

Another package that makes up the simulation system is actually part of the Ada run-time system, namely the package OS_Interface. One purpose of this package is to serve as an interface between the Ada run-time system and the low-level operating services. For instance, it allows for the creation of Ada tasks through the use of a thread
package standard called Pthreads [8]. Pthreads is defined for the family of IEEE operating system interface standards known as POSIX, or Portable Operating System Interface. In order for Ada to use these interface standards in setting up its own tasking architecture, the Pthreads interfaces must first be “imported” into the Ada language.

To briefly describe the process of importing, a function that is defined in a “foreign” language (i.e. C) can be used in Ada only if it is first imported. To accomplish this, Ada provides a pragma called Import, which allows the compiled C function to be recognized and used by an Ada program. By importing the Pthreads functions, the Ada run-time environment can use all of the facilities provided by Pthreads to create its tasking architecture.

Several of the existing functions in the os_Interface package are modified to create the simulation system. One of these is pthread_cond_timedwait(), which originally imports the cond_timedwait function directly from Pthreads. This function is called by a thread, which passes in a condition variable and a time value as arguments. The function then places the calling thread into a wait state, and when the specified condition variable is signaled or when the system time is greater than the time value specified by the thread, the function resumes the calling thread.

Although the RTOS simulation system utilizes cond_timedwait in a similar way, this function must be modified in order for the simulation to work correctly. In order to understand the changes made to the function, it is appropriate to briefly analyze what occurs when a delay statement is used in an Ada program. Taking a cyclic Ada task as an example, a user might use a delay statement in the body of this task for purposes of simulating an execution start time. For instance, a 25 Hz cyclic task could be
implemented with a repeating loop containing a delay of 0.04 seconds (the task’s period). Once the delay statement is reached, the Ada run-time system immediately calls the Timed_Delay function, which eventually calls pthread_cond_timedwait(). In order to correctly implement the RTOS simulation system, pthread_cond_timedwait() is modified so that the function releases the calling thread (or Ada task) only when the condition variable is signaled (in this case, with the function pthread_cond_signal() in clock_manager) and not when the time value specified by the thread has been reached. In other words, the objective is to simply put the thread into the priority queue and then to suspend the thread until the condition variable is signaled. The time at which the thread should be released is taken into account in the enqueueing function. Thus, pthread_cond_timedwait() consists of the following modifications: first, the enqueueing is performed by the function enqueue (described later), after which the function pthread_cond_wait() is called to suspend the calling thread.

3.2.1.1.3 Queueing Functions

The remainder of the functions that support the simulation system is also contained in the OS_Interface package.

- Get_Priority uses the function pthread_getschedparam to obtain a thread’s current priority.
- Wakeup uses pthread_cond_signal to release a thread on a condition variable.
- Dequeue_Head dequeues the head of the priority queue.
- Enqueue puts a thread into the priority queue.
The `Enqueue` function sets up a priority queue with which to add and remove tasks. Once a task calls `Enqueue`, the function determines where to add the task onto the queue by examining the task's current priority.

3.2.1.1.4 Test Program

An Ada main program is created to test the RTOS simulation system. This test program is similar to one that would be created by a user to implement various application tasks. It creates three cyclic tasks, each with a different frequency of cycles. Furthermore, priority is assigned to the tasks in order of highest to lowest frequency. To simplify the analysis of the test program, a text output is displayed every time a task has completed one cycle.

The program runs successfully and displays the expected text messages. Before running through the cycles, each task displays a message stating that the task has been created and is ready to execute. Once the cyclic loops begin, the task with the higher frequency begins its execution. In this test, the chosen frequencies are integral multiples of each other, namely tasks of 25 Hz, 5 Hz, and 1 Hz. Thus, at the simulated clock time of 0.2 seconds, when both the 25 Hz and 5 Hz tasks are scheduled to execute, the 25 Hz tasks correctly executes first due to its higher priority. Similarly, at 1 second, the execution proceeds in the order of the 25 Hz, 5 Hz, and 1 Hz tasks.
3.2.1.2 C Program

To illustrate the RTOS simulation system’s ability to support the C programming language, a C main program is also written to serve as a test mechanism. Although this program provides for similar test cases as the Ada test program, it requires additional modifications. Firstly, several Ada functions have to be imported into the C program. Most of these functions are part of the standard Pthreads interface, while others are part of the Ada run-time environment. For instance, in order to create C threads for use as Ada tasks, information must be provided to the run-time system so that it has knowledge of both Ada tasks and threads created by foreign code. This is accomplished through the use of the run-time system function `create_thread`. This function takes in the name of the C thread and the thread’s priority, and creates the corresponding Ada task.

Other run-time system functions include `get_thread`, which allows the user to access the identification of a thread that was created using `create_thread`. `get_sim_time`, which is added to the set of run-time functions, allows the test program to use Ada’s simulated clock time as a C float type. The function `clock_proc` is implemented as a null procedure, and serves only to activate the simulated clock in the Ada packages.

An additional feature in the C test program that differs from the Ada program is that the Ada tasking environment automatically waits for all current tasks to terminate before the main program is allowed to finish. In C, however, the main program must be explicitly suspended to allow any currently running threads to finish executing. This is done with separate calls to the Pthreads function `pthread_join()`.
One further supplement to the C program involves the addition of two Ada run-time defined routines, namely adainit() and adafinal(). adainit(), which must be called in the C program before using any Ada subroutine, initializes the Ada portions of the program by calling the appropriate elaboration routines. adafinal(), which must be called before the C program terminates, performs any library-level finalization required by the Ada subprograms.

The final implementation of the C test program executes in the same manner as the previous Ada test program.

3.2.2 Scheduler

The scheduler that is developed for RTOS simulation system allows the user to preview the execution of the applications tasks on the host before implementing the applications on the actual real-time operating system.

3.2.2.1 General Architecture

The scheduler is designed to allow the user to create multiple tasks and to specify various characteristics for these tasks. These characteristics include the frequency of a cyclic task (or period of execution), the number of cycles that the task will execute, and the amount of time required for the task to perform its execution. The latter characteristic requires a modification to the original RTOS simulation system, which assumed that the processor speed is infinite. This assumption is no longer valid, since a task will require some amount of time to execute. Once the user sets up the application tasks and calls the
appropriate scheduling functions, the scheduler will then assign priorities to the task and enable the tasks to run.

3.2.2.2 Modifications to Simulated Clock

The original implementation of the RTOS simulation system needs to be modified to allow the scheduler to conform to the specifications. One modification involves the technique of checking the priority queue for tasks that need to be released for execution. The original non-real-time simulation system orders the queue by the tasks’ priorities. When it is time to check if the tasks are ready to be dequeued (released), the head of the queue (containing the highest priority task) is always checked first, and the rest of the queue is not checked if the head is not ready to be dequeued. This is problematic in the case where the head of the queue is not ready to be released, but some task of lesser priority might be ready. Thus, this task will not be released until the task at the head of the queue is released. Note that this original implementation does not cause a problem, if the frequencies of the cyclic tasks are an integral multiple of each other, which was originally assumed. In the original case, the 69% utilization attributed to the rate monotonic algorithm now reaches 100%, and thus, is generally the norm. To provide for the general case of tasks not having integral multiple frequencies, the simulation system simply searches through the entire queue to determine which tasks need to be released, regardless of whether the head of the queue is released or not.

One further complication involving the above modification is a situation in which two or more tasks are ready to be released. In this case, the simulation system must first release the higher priority task, and then allow it to execute to completion. Thus, the
lower priority task is released for execution only after the first task has finished running. Although they could both be released at the same time, the implementation of the simulated clock is made easier by only releasing a single task.

3.2.2.3 The Scheduler Package

In addition to the modifications to the simulation system, several additional functionalities are needed to implement the scheduler. Most of the functions are defined in the scheduler package. Figure 4 shows the interactions between the existing simulation system with the new scheduler package.

![Scheduler Interaction Diagram]

Figure 4 Scheduler Interaction

The general implementation of the scheduler involves setting up a separate queue of tasks, named TPQ (task priority queue), which is distinct from the queue (sim_time_queue, or STQ) of the original simulation system. It is necessary to create
the TPQ, although this queue does not actually interact with the clock in the simulation system. The queue interacts with the scheduler package for task data manipulation, and interfaces with the simulated clock through the STQ. The STQ does the job of telling the simulation system what the current priority of the tasks are, and indicates the task’s execution frequency to the simulated clock. Although it is not essential to have two distinct queues, it is useful to keep the STQ unchanged from the original simulation system. Furthermore, a new queue would need to be implemented anyway when creating a test program to run on the real-time system. This new queue for the real-time test can now be adapted from the current TPQ on the simulation system.

The function that sets up the TPQ is \texttt{add\_task().} It allows all the tasks to be put into the queue in order of decreasing priorities, given that the priority of the task is known beforehand. However, this information is usually not known until the scheduler assigns the tasks’ priorities according to a selected algorithm. In this case, once the assignment of priorities is done, the TPQ is re-arranged using the \texttt{order\_TPQ} function. This function simply looks at the current priority of each task and arranges the TPQ in order of decreasing priorities.

In the case of running the scheduler in the simulation system, it is not necessary for the Ada runtime system to have knowledge of the current priorities of the tasks. The clock packages take care of queueing and releasing tasks according to task priority and wake-up times. However, when running the scheduler with application tasks in the real-time operating system, it is necessary to inform the run-time system of the current priorities of all tasks, since the clock packages no longer exist in the system. This job is
performed by the `prioritize_tpq` function, which relays the task priorities to the run-time system by using the `set_priority` function.

Once the selected scheduling algorithm assigns the task priorities in the TPQ, the STQ must also be updated with these priorities. This is accomplished with the function `adjust_priorities_stq`, which assigns the priority of each task in the STQ equal to the priority of the corresponding task in the TPQ. The function `order_stq` then reorders the STQ in decreasing task priorities. Figure 5 illustrates the general steps in the scheduling process.

**Figure 5 Steps in Task Scheduling**

```plaintext
Task is created

TPQ is updated by scheduler package

Creation of tasks complete

TPQ prioritized by scheduling algorithm

Inform run-time system of priorities.

Task Execution

STQ is updated by clock package

Set priorities in STQ equal to those in TPQ

Re-order STQ according to new priorities
```
The right side of the figure involving the STQ would be eliminated in the real-time case.

The user is able to select which scheduling algorithm is used in the simulation system, either rate monotonic or earliest deadline first. The rate monotonic algorithm is selected by calling the procedure rate_monotonic, which first puts the TPQ in order of decreasing task frequencies and then assigns priorities to each task, starting with the highest priority for the highest frequency task. Once this is done, the task with the greater frequency (higher priority) is chosen to be released first if it has reached its execution start time. Since there is no preemption, all tasks are allowed to run to completion.

The earliest deadline first algorithm is selected by calling the procedure EDF. Like the rate monotonic algorithm, the EDF algorithm orders the TPQ from highest to lowest priority. However, the highest priority is given to the task with the earliest deadline. Furthermore, unlike the static nature of the rate monotonic algorithm, the EDF algorithm is dynamic since the deadlines of the tasks are continuously changing. Therefore, the EDF procedure must always be called after every change in the deadline values. Once a task begins executing, it runs to completion since preemption does not occur.

### 3.2.2.4 Test Program

An Ada main program is created to test the scheduler for the simulation system. This program is similar to the one used to test the original simulation system. However, it activates the scheduling facilities using the aforementioned scheduler package. Furthermore, in order to simulate actual execution time, the processor speed can no
longer look infinite. Thus, each task simulates execution by simply updating the clock by its execution time.

The test program correctly schedules the task according to the selected algorithm and indicates the occurrence of a missed deadline. To illustrate a typical missed-deadline scenario, take for example a cyclic task of frequency 25 Hz (period of 0.04 seconds and has highest priority). If it misses its deadline due to the execution of another task that has taken too long, then the 25 Hz task will immediately start executing as soon as possible, i.e. right after the task that caused the delay. It will then continue to run at its regular frequency.

3.2.2.5 Comparisons and Results

To illustrate the differences between the two scheduling algorithms, a sample scheduling scenario is used. Consider a set of cyclic tasks with the following rates: 9 Hz, 5 Hz, and 1 Hz, each with deadlines 0.11 secs, 0.2 secs, and 1 sec respectively. Assuming that each task has an execution time of 0.05 secs, applying either the rate monotonic or earliest deadline first algorithm results in giving the highest priority to the 9 Hz task, and then to the 5 Hz and 1 Hz tasks. In this case, both algorithms give the same satisfactory results, and both will usually be equally efficient in the scheduling process assuming that they are performed using static scheduling implementations.

By examining a slightly modified scenario, we can examine how the algorithms can provide different results. Consider the same set of tasks with the same rates, execution times, and deadlines as above. However, instead of the 1 Hz task having a deadline of 1 sec, its deadline has now changed to 0.075 secs. Using the rate monotonic
algorithm, the tasks will have the same priorities as in the previous scenario: 9 Hz, 5 Hz, and 1 Hz in descending order. This scheduling arrangement, however, does not allow all the tasks to meet their deadlines. In particular, once the 9 Hz and 5 Hz tasks complete their execution (each having 0.05 secs execution times), 0.1 secs have already elapsed, which is past the 1 Hz task’s deadline of 0.075 secs. Given that these are hard real-time deadlines, this situation is unacceptable. By applying the earliest deadline first algorithm, however, the resulting order of decreasing priorities is 1 Hz, 9 Hz, and 5 Hz. This arrangement allows all the tasks to execute without violating their deadline constraints.

The situation of having a task with a deadline that is earlier than its execution period (as with the 1 Hz task in the previous example) can come into play in a variety of circumstances. For instance, such a task might have a frequency of 1 Hz, in which it only needs to perform its job every second. However, the task could be required to finish a particular part of its job earlier than one second so that the results can be immediately used elsewhere. In this case, the earliest deadline first algorithm performs the scheduling process satisfactorily.

For situations in which all the timing parameters are known beforehand, static scheduling can be implemented for both algorithms. The static situation is generally advantageous to all scheduling procedures, since the scheduler only needs to do work at the beginning of system execution. Thus, implementations for the static rate monotonic and static earliest deadline first algorithms are generally equally efficient in utilizing processor resources. The choice of which algorithm to use, then, is based solely on which algorithm produces no missed deadlines. The choice becomes more difficult if one
or more of the algorithms use dynamic implementations, a topic which is mentioned later in this section.

There are situations in which deadline misses are not fatal to the system, but in which minimizing the misses would be optimal. In this case, the two scheduling algorithms can be examined to find the more efficient one. Consider a scenario consisting of four cyclic tasks with rates of 9 Hz, 5 Hz, 4 Hz, and 1 Hz. If each of these tasks take a small amount of time to execute, then both the rate monotonic and earliest deadline algorithms work in the same manner and allow all the tasks to run correctly without delays. However, in the simple case of one task taking a large amount of time to execute, one algorithm might prove to be more effective than the other. For example, consider a 9 Hz task that has an execution time such that it finishes execution at 1.244 seconds from the starting time of zero seconds. Using the rate monotonic algorithm, the 9 Hz task will run first and finish at 1.244 seconds, and then the 5 Hz task will start running. If this task has an execution time of 0.005 seconds, then it will finish at 1.249 seconds. Assuming that the 4 Hz task has the same execution time of 0.005 seconds, then it will finish at 1.254 seconds. In this scenario, the 5 Hz task has missed five of its deadlines (with the most recent deadline at 1.4 seconds) and the 4 Hz task has missed four deadlines (with the most recent at 1.25 seconds). However, if the earliest deadline first algorithm is used, the 9 Hz task will again run first and finish at 1.249 seconds. At this point, the 4 Hz task has an earlier deadline (1.25 seconds) than the 5 Hz task (1.4 seconds). Therefore, after the 9 Hz task has finished, the 4 Hz will immediately start executing, and will finish at 1.254 seconds. This results in the 4 Hz task missing only three deadlines, and the 5 Hz tasks still misses five deadlines as before. In this simple
case, the two tasks switch the order in which they begin executing, thus enabling one of the tasks to miss one less deadline.

Although the earliest deadline first algorithm can be more efficient in some cases, a definite tradeoff exists with its use. If this algorithm is implemented dynamically, then it must run at various times throughout the execution of the system. Thus, more calls will be made to the scheduling functions with the earliest deadline first algorithm than with the static rate monotonic algorithm, resulting in more processor usage and potentially longer execution time of the program. For instance, in the previous example, the earliest deadline first algorithm resulted in fewer deadline misses. However, examination of the program's profile data (using GNAT's gprof utility) shows that when the rate monotonic algorithm is used, there is only one call to the scheduling functions that modify the task queue, while there are significantly more calls to those functions when using the earliest deadline first algorithm.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Period (Tₙ)</th>
<th>Exec. (Cₙ)</th>
<th>Util. (Cₙ/Tₙ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>Update Ship Attitude</td>
<td>2.56 ms</td>
<td>0.5 ms</td>
<td>19.53 %</td>
</tr>
<tr>
<td>P₂</td>
<td>Update Ship Velocity</td>
<td>40.96 ms</td>
<td>5.0 ms</td>
<td>12.21 %</td>
</tr>
<tr>
<td>P₃</td>
<td>Send Attitude Message</td>
<td>61.44 ms</td>
<td>15.0 ms</td>
<td>24.41 %</td>
</tr>
<tr>
<td>P₄</td>
<td>Send Navigation Message</td>
<td>983.04 ms</td>
<td>30.0 ms</td>
<td>3.05 %</td>
</tr>
<tr>
<td>P₅</td>
<td>Update Console Display</td>
<td>1024.0 ms</td>
<td>50.0 ms</td>
<td>4.88 %</td>
</tr>
<tr>
<td>P₆</td>
<td>Update Ship Position</td>
<td>1280.0 ms</td>
<td>1.0 ms</td>
<td>0.08 %</td>
</tr>
</tbody>
</table>

Figure 6: Periodic Tasks from INS
(From Real-Time Software Engineering [4])
Figure 6 provides a sample set of tasks for an inertial navigation system (INS) [4]. This is a typical example for real-time systems that have a variety of jobs to accomplish. The rate monotonic algorithm is appropriate for this set of tasks, which has a total utilization value of 64.16%. However, it must be noted that several assumptions are made. Firstly, context-switching overhead is either assumed to be negligible, or that it has been factored in to the tasks’ execution times. Also, there is negligible overhead for activating the tasks, and the tasks are independent.

Given the above assumptions, the tasks are prioritized in decreasing order, from $P_1$ to $P_6$. Thus, the rate monotonic simulation scheduler is able to schedule these tasks with no deadline misses. However, if the utilization value changes due to a modification in timing parameters, such as a change in execution time or unexpected delays in the execution of a task, then another algorithm, such as earliest deadline first, might be more appropriate. Tradeoffs must always be evaluated according to the needs of the system.
4 Conclusion

4.1 Summary

This thesis has described the implementation of a real-time operating system for the Power PC. Ada was the language used in the development of the operating system, and the GNAT cross-compiler was used to port the system files to the Power PC. Although an executable file was produced to test the operating system, a couple of errors were encountered during execution that stopped the program from completing.

The simulation system for the Power PC RTOS was developed for the Linux host system. It consists of a simulated clock that allows for quicker program testing and for the ability to step through program execution in debugging mode. A scheduler was developed in conjunction with the simulation system. It allows for task scheduling according to the rate monotonic and the earliest deadline first algorithms, and indicates the times at which the tasks have finished executing and whether a task has missed any of its deadlines.

4.2 Further Work

There are many aspects of the project that can be continued. The executable file for the real-time operating system can be further worked on, especially if the high-level Ada debugger can be implemented to work with the low-level Power PC debugger.
Furthermore, the scheduling algorithms developed for the simulation system can be completely ported to the real-time system for implementation and testing.

The scheduler for the non-real-time simulation system can be further modified to adapt to more complicated scheduling scenarios, such as combining several different algorithms into a single, efficient scheduler. This scheduler can then be ported to the real-time operating system. Other aspects of the RTOS and simulation system that can be implemented are memory management, snapshot/rollback capabilities, and fault tolerance. Such additions would make the system more valuable to projects that require real-time capabilities.
5 References


