
133
637
THS

THESIS
1

MICHIGAN STATE LIBRARIES
3 1293 01774 9551

LIBRARY
Michigan State
University

This is to certify that the

thesis entitled

Implementation and Evaluation of MC/OS
for the Handy Board

presented by

Anthony E. Pappas

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Diane J Rover
Major professor

Date 8/10/98

O-7639

MSU is an Affirmative Action/Equal Opportunity Institution

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

1/98 c:/CIRC/DateDue.p65-p.14

IMPLEMENTATION AND EVALUATION OF μC/OS FOR THE HANDY BOARD

By

Anthony E. Pappas

A THESIS

Submitted to
Michigan State University
In partial fulfillment of the requirements
For the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1998

ABSTRACT

IMPLEMENTATION AND EVALUATION OF μC/OS FOR THE HANDY BOARD

By

Anthony E. Pappas

RTOS (Real-Time Operating System) technology is a viable design option for many embedded systems and applications. It has proven itself as a reliable and useful tool in developing systems that have hard and soft real-time requirements. Many engineers have written their own RTOS because nothing else has been available. However, this often takes away from the application development time. Their RTOS would often be inefficient, requiring maintenance and would not be scalable or portable.

Commercial RTOSs are now available. More engineers chose them over writing their own because commercial RTOSs are proven technology. They are written efficiently, scale well to meet the real-time requirements and port easily to new processors. These RTOSs allow the application to remain abstract and they take the responsibility of handling the system details away from the application.

We investigate μC/OS, an RTOS for the MC68HC11, with the following key objectives. First, μC/OS must be ported to the MC68HC11 and there must be a basic understanding of how to use and expand its features for real-time embedded applications. Second, applications must be developed to illustrate these features. Also, a prototype environment must be created to test them. Finally, recommendations for curricular use of the RTOS must be made and a description of how an RTOS fits into co-design must be given.

DEDICATION

This paper is dedicated to my loving wife, Rachelle Pappas, who has given me full support and encouragement on this thesis and to my parents, Greg and Gloria Pappas to whom I owe so much.

ACKNOWLEDGEMENTS

I would like to thank my family and friends for understanding the importance of what this thesis means to me and for giving me their full support. I would also like to thank my advisor, Dr. Diane Rover, for her support and leadership throughout my project. She has been very flexible and understanding of my work situation and I greatly appreciate it.

I would also like to thank Seth Mosier and Andy Huang for the work on the train simulator that is used in conjunction with my bridge control application.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

Real-time operating systems (RTOS) live at the core of any system that requires applications to be logically correct and meet real-time constraints. These operating systems support real-time applications and systems via kernel services, task and resource management, and interrupt handling. There are two types of real-time systems, hard and soft.

Hard real-time tasks operate correctly and in a timely fashion[1]. Hard real-time requirements put strict time constraints on the system where the success of its operation hinges on the timeliness of tasks.

Soft real-time requirements have some time constraints but a "best effort" by the tasks to meet their deadlines is sufficient. The real time operating system must support one of these two types of real-time systems.

To support either hard or soft real-time systems, the RTOS must possess a number of capabilities. It must be written efficiently, supporting and optimizing the use of the available hardware. It must have real-time kernel services that aid the application in meeting the real-time requirements. In essence, the RTOS must allow the application to be more abstract, meet real-time requirements and avoid hardware or system details. To accomplish this, the RTOS interfaces the application to resources, interrupts and timing processes. The RTOS provides this interface through kernel services. These services allow the application to concentrate on the specifications of the problem and the design rather than deal with underlying system details.

The application can remain abstract by being broken up into tasks. These tasks can have priorities, communicate among each other and have access to resources. They also have access to the timing services that help meet the application's needs.

Application portability can be maintained by choosing the proper RTOS. Keeping the application abstract leaves the RTOS to handle the system details and provide a standard system interface for the application. Re-writing an application for an RTOS may not be simple but once this has been done, running it on a new system could be trivial. Only the RTOS needs to support the new system and most are written to be portable. This abstraction allows the application to take on new features more easily. It could be as simple as writing a new task or using a resource in a unique way. Also, future designs and upgrades need not be concerned with portability or system interfaces because of a new processor. This is made possible by the fact that RTOS kernel services remain the same while the underlying system may change. Most of the time, a new system or processor means re-working the applications (not written for an RTOS). This often introduces new problems or extends application development time. Thus, an RTOS may also reduce the time to market in an industry where timing, in every sense of the word, is crucial.

1.1 Problem Statement

8-bit and 16-bit microprocessors support increasingly more complex embedded applications and require RTOS support to manage them. This thesis presents the implementation of an RTOS written for the Handy Board, which incorporates Motorola's popular 8-bit embedded microprocessor, the MC68HC11. The MC68HC11 thrives in many of the industries' real-time applications and has many features that fit well with an

RTOS. Many educators have also embraced this processor because of its industrial popularity and wide spread use in educational books and publications.

Also in this thesis, we discuss the tools that aid in the implementation and development process of porting and writing applications for an RTOS. μC/OS by Jean Labrosse is a well-documented and portable RTOS chosen for this thesis[1]. A framework application for a classroom environment at Michigan State University has been developed using μC/OS. This thesis intends to support future labs or assignments on RTOS application development in embedded systems. A course that may have such labs or assignments would be an operating systems class or a computer engineering capstone course on embedded systems. This thesis may also serve as a tool for co-design research involving such tools as POLIS from UC Berkeley[2].

1.2 Approach

In this thesis, we present how to implement and use an RTOS in an 8-bit embedded environment. Implementing an RTOS such as μC/OS requires the proper tools and RTOS knowledge as well as good programming skills. Research in the RTOS market gives insight to what standard features are used in the development process. For this thesis to be useful in an instructional environment, the expectations of the student and teacher were explored and understood. We present a real-time train application developed for μC/OS and other case studies to demonstration the usefulness μC/OS.

1.3 Contributions

The following lists the contributions of this thesis.

- An implementation of μC/OS for the Handy Board.
- The development of real-time applications with μC/OS.
- An analysis of the features and performance of μC/OS.
- An environment that makes RTOS development part of MSU's real-time systems education and research.

1.4 Organization of Thesis

Chapter 2 gives background information on RTOS technology and the hardware design choices that relate to them. This gives enough information to understand how μC/OS works and what it provides to real-time applications. Chapter 3 discusses all the components of implementing and using μC/OS. Chapter 4 explains the train application written for μC/OS. The results of using μC/OS and a performance analysis of it are given in Chapter 5.

Chapter 2 Background

2.1 RTOS Introduction

"When used properly, an RTOS changes your design from one of main functions calling lots of subroutines to an event-driven, interrupt-responsive system organized by tasks and time."[3]. A transition of an application to tasks that make use of a real-time operating system may not be easy but it will have great benefits.

If an application can be broken into tasks then the RTOS optimizes them for the system and runs them in a multitask environment. Each task may not need continuous use of the CPU allowing other tasks to execute. This may not have been possible in a non-RTOS system because the application lacked any abstraction that would allow for context switching.

Task communication can allow for synchronization of the tasks or provide a data path among them via kernel services such as semaphores, message boxes and queues. Tasks can also use timing services provided by the kernel to ensure that they meet their real-time specifications or requirements. A task may delay its self for a specified amount of time or obtain the current time. This gives a task some sense of time and allows it to take time intensive action i.e. sounding an alarm at 6:30 A.M. or waiting 30 seconds before attempting to gain access to a printer.

To avoid resource corruption by tasks, the RTOS handles the resources of the system. Two or more tasks may attempt to modify a resource and the RTOS will maintain the integrity of the resource during this process. The RTOS must provide this service in a simple and standard way, allowing the tasks to focus on their true function. Thus, a resource can be shared among tasks without concern for misuse. The main functions of the kernel provide context switching and management of tasks, handling the communication among them.

To accomplish task management, the RTOS must have a scheduler that handles various states of the task. The task states usually consist of the following: Current, Ready, Delaying, Suspended, Timed or Blocked. Figure 1 is an example of how these tasks move among the states.

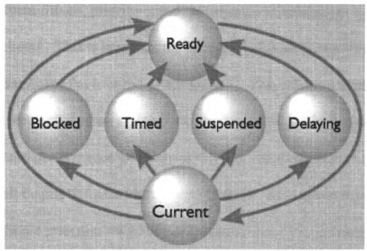


Figure 1. Task states.

A task begins in the Ready state and the scheduler will allow it to run when the processor becomes available and another task is not deemed worthier based on the

scheduling algorithm. Once it begins to run, it may continue until the scheduler decides that it must be moved to one of the other states such as Timed or Suspended. This could occur as part of the scheduling algorithm. The scheduler would move the task to the Timed state because all tasks are time-sliced to run on the processor. A task could be moved to the Suspended state while the RTOS performs some kernel services or handles an interrupt. The scheduler may have nothing to do with changing the state of the task, the task may move itself to a Delayed or Blocked state. The task may need to wait for a given amount of time; thus, changing its state to Delayed. It may also attempt to obtain a resource that another task holds; thus, changing its state to Blocked. The processing of the states must be fast and efficient to keep overhead to a minimum. Scheduling decisions are made based on the priority and state of the task.

An RTOS must implement one of several available scheduling algorithms and chooses one based on the expectations of the kernel. Some of the more advanced algorithms can be event-driven, priority-based, preemptive scheduling or can incorporate a rate monotonic scheme. This is based on the thought that what is required of the real-time software will vary over the lifetime of the system.[4] This scheduling causes lower priority tasks to be preempted when a higher, time-critical task needs to execute and ensures the application performance goals are achieved. Others implement a round robin approach or may still be priority based but not preemptive. Task priorities can either be static or dynamic. Static priorities only have one priority that can not change throughout its life. A dynamic priority based task can be changed during its lifetime.

Priority inheritance can be a crucial feature for resource and task management. This scheme has the task holding a resource temporarily inherit the priority of any higher

priority task attempting to gain access to the same resource. Without this scheme, schedule problems arise because a task holding a resource may not be scheduled in time for other higher priority tasks to gain access to this resource; thus, preventing the tasks from meeting their real-time requirement. In essence, a lower priority task blocks a higher priority one, entirely contradicting the priority-based scheme. This is often referred to as priority inversion.

It is often important to know whether the kernel supports preemptive scheduling. Preemptive scheduling allow tasks to be interrupted to re-evaluate which task ultimately deserves control of the system. Implementing a preemptive RTOS often results in system overhead but its response time will often offset the overhead. A preemptive RTOS may be the only real choice for many systems and applications with hard real-time requirements. These applications and embedded microprocessors have found their way into electronics devices, appliances, automobiles, industrial control systems and computer network equipment using 8-bit processors and DSP chips.[5]

2.2 RTOS Spectrum

The requirements of the target architecture determine what CPU-board hardware to implement, what software OS vendor to use, and what the architecture of the real-time OS should be for the design.[6] An RTOS interfaces the hardware system to the application to provide a real-time environment. An RTOS can only optimize the use of the hardware components and interface it to the application but its maximum performance limitations lie with the hardware. Many times, the hardware constraints should be considered before looking at the RTOS features because certain processor features such as context switching and interrupt handling are first limited by the hardware

chosen. If the hardware suffices then an analysis of the RTOS kernel becomes the next logical step.

Features supported by most microkernels include fast multitasking, built-in interrupt handling support, and preemptive, non- preemptive or round robin scheduling. It should also provide the application with timing services to aid in meeting the real-time requirements. Memory management is often a good feature to have in an RTOS. Often a quality feature of an RTOS is its ability to dynamically allocate and de-allocate memory.

The microkernel design should minimize system overhead and enable fast responses to external events. It should also provide efficient task communication mechanisms, permitting tasks to coordinate within the real-time system and to external systems. This can be accomplished via mailboxes, message queues and network interfaces. Control of critical system resources is handled via various types of semaphores.

Other RTOS features are related to its implementation. Scalability often becomes important for a range of embedded systems. The scalability of an RTOS allows an application to incorporate only the functionality it requires. Where memory and system resources are limited, some of the RTOS elements may have to be scaled back or eliminated. Other larger and more powerful systems may have the resources to make use of all the available features. Certain applications and systems may require particular kernel services such as a network interface or priority inheritance.

Good development environments help the RTOS to be a success, reducing the time to market. The necessary RTOS development tools include compilers, debuggers, simulators, emulators and other development tools. Compilers and the language used for

the RTOS applications must be accepted by the industry and be in wide spread use by developers. Most developers are familiar with developing with a known processor that can easily be tested and controlled. In embedded systems, this type of environment should be preserved where possible to aid in the development process. These tools should also be processor independent or be able to conform to other processors. The embedded systems industry utilized numerous processors that are required to meet stringent specifications; thus, not only should the tools be processor independent but so should the RTOS. Part of the RTOS must be written for a particular processor but it should be kept to a minimum.

Testing and debugging a real-time system requires special simulators or emulators because the actual system may not be available or does not lend itself to development. The test environment should match the system closely and allow the developer to take control. The same type of environment should allow the developer access to source code and aid in debugging the application or system.

To continue the development cycle of design, test, debug, and optimize, a tool should be available to optimize the performance of the RTOS and its applications. It should be customizable and intuitive for the developer, who may have a system that requires certain optimizations or run in a non-standard fashion. One such RTOS, DR DOS has an open kernel that allows developers to fine-tune the application as needed.[7] This can be accomplished by keeping to standards such as POSIX 1003.1b real-time extensions (ANSI/IEEE) and by maintaining an open environment for the developer. By conforming to a standard such as POSIX, the developer is free to migrate to various hardware architectures as needed, allowing him/her the option of using off-the-shelf

solutions.[8] Without these environments and tools, the system will be more of a "black box" with limited information of how the system is working. This is unacceptable in a market that demands the "faster-cheaper-better" systems with seemingly impossible deadlines.

These systems may take on different forms, having various specification and expectations. Some systems may have strict speed and heavy resource requirements. Many of the simpler designs have very minimal requirements, needing only the basic services an RTOS has to offer. The type of processor chosen for a real-time system usually reflects what kinds of features are required from the RTOS. 32-bit processors usually reflect a system that need all the rich features an RTOS has to offer while a 16-bit processor may require only a few and a 8-bit processor the bare minimum.

Architectural differences between 8-bit,16-bit and 32-bit processors are quite pronounced. The instruction set varies the most among these processors. As the bit width is extended, more instructions can be added that may take advantage of hardware. Also, greater address ranges can be accomplished with greater register width. Computer engineers incorporate all the latest architectural advances in the newer 32-bit processors, leaving 16-bit processors with some of the recent advancements and 8-bit usually with older technology. These advancements affect a processor's performance drastically and new advancements open the door to new embedded applications that may not have been possible a few years ago. For example, many new advanced devices may take advantage of new technology such as speech recognition or image processing that makes use of these 32-bit RTOSs. The following sections discuss other distinctions that may be more relevant to implementing an RTOS.

2.2.1 8-bit Processors

8-bit processors can often help an RTOS be configurable, efficient and stay within a tight budget.[9] On the other hand, these processors can also be limited on several fronts. For these processors, CPU speed remains relatively low. This limits the speed requirements of the application, making features such as multithreading unavailable. For a task to meet its requirements, a task's scheduled uninterrupted time slot must not be too short i.e. tasks may be time sliced in milliseconds. On higher speed processors, the time slot can be shorter because the task can achieve more work in a shorter interval i.e. tasks may be time sliced in microseconds. However, for certain processors and applications, 8-bit processors schedule the tasks adequately and other hardware choices need to be considered.

Often, processor speed does not dictate the choice of a hardware system. For example, 8-bit processors often do not have the same number of internal registers. This makes it difficult for tasks to accomplish much of their work in their scheduled time slot. A task would be forced to use memory more often that is considerable slower than working out of internal registers. Others have improved upon the 8-bit processor, making large internal registers available. As an example the Intel 80386SX and the Motorola (West Austin, TX) 68008 both have an 8-bit external data bus that feeds 32-bit wide internal registers.[10]

Often 8-bit processors do not have the same interrupt handling ability as other more advanced processors. These processors may not have the same number of interrupts that restricts the number of external devices in the system. Also, CPU

response time to interrupts may be a limiting factor for many real-time applications. Many times an 8-bit CPU's interrupt system suffices and only the RTOS interrupt services need to be evaluated.

Most 8-bit processors have ample timing functions, a feature that the RTOS demands from the hardware. The limitation lies in the resolution of the timers i.e. they may be measured only in microseconds. For some embedded systems, the timer resolution may be adequate and system I/O must be considered.

The nature of embedded systems is often one of heavy I/O. The speed of the I/O device(s) often becomes a driving force in determining the hardware requirements. This device will also determine the width of the data path between it and the processor. Many times, 8-bits processors may not meet this requirement or they may have to rely on other external devices to interface to these devices. Present 8-bit processors may have a need to be maintained because a system has a small board area and low power requirements. Otherwise, other higher performance processors may be more effective.

Memory requirements often dominate the choice of hardware systems. One of the major limitations of the 8-bit processors is their addressing ability. However, special hardware can expand the addressing range of these processors, as with the MCS51 that expands the addressing limit to 16 Megabyte.[9] Some other processors may be limited to as little as 256 bytes of internal memory. Often the memory consists of not only RAM but also ROM where the RTOS resides. If adding the expanded memory can eliminate this limiting factor, then the RTOS can provide more of its microkernel services. Toshiba, that did implement an expanded memory solution in their 807 family allows for its

processor to reap the benefits of an RTOS's ability to context-switching among tasks or handle ISRs(Interrupt Service Routine) within an RTOS.[11]

8-bit processors have other compelling advantages, at least presently, that make them useful. First of all, they are cheap. This alone is a very compelling reason to use an 8-bit processor. Mass production of embedded systems such as small appliances demand that component costs remain low. In particular, portable devices not only require the parts to be low cost but power efficient. Intelligent, low-powered devices are becoming common place as seen in many handheld computers, mobile phones, pagers, etc. These devices are what driver processors and the hardware they mange to shrink in size and consume less power while maintaining their intelligence.[9]

Finally, most 8-bit processors have proven themselves in the industry. This also implies that many current processors may be upgraded with an RTOS and optimized or improved upon. This already eliminates the need to chose or design hardware, allowing the focus to be on matching the current processor with the appropriate RTOS.

Some engineers desire a low cost 8-bit processor but need better performance. One vendor has a solution that is a combined 8/16 bit microcontroller, developed by SGS-Thomson Microelectronics (STM) from their ST9+ core, that provides both the low cost of the 8-bit MCU's and improved performance with certain functions from their 16-bit technology.[12]

2.2.1.1 Example RTOS

μC/OS is the focus of this thesis and it is a kernel that can be used with 8-bit processors because it is implemented in a small, efficient way without sacrificing many of the critical real-time services. The following is a list of its features.

- Supports the following processors but can be ported to almost any processor:
  - AMD's 29K; Hitachi's H8/300H; Intel: ix86 SMALL model, ix86 LARGE model, 80486 Protected Mode with context switching of floating-point regs; MCS-251; Motorola: MC68HC11,68HC16,680x0 and CPU32; Philips: XA MEDIUM model, XA LARGE model; VLSI Technology: ARM; Zilog: Z-80/Z-180
- Event driven operation
- Multitasking preemptive scheduling
- Static and dynamic kernel objects
  - Tasks
  - Semaphores
  - Queues
  - Mailboxes
  - Timers
  - Interrupt Handlers
- Fixed or dynamic task priorities
- Inter-task communication and synchronization
  - Semaphores
  - Mailboxes
  - Queues
- Timeout options for many services
- Efficient interrupt servicing
- Small RAM and ROM requirements
- Standard programmer interface in C on all processors

2.2.2 16-bit Processors

16-bit processors have limits in some of the same areas as the 8-bit processors but not to the same degree. Most 16-bit processors form a much more powerful system than their 8-bit predecessors. Usually they contain higher speed processors that may accommodate more powerful applications but may be inadequate for certain high performance embedded systems. When a task is scheduled, it can complete a reasonable amount of work within its allocated time slot. Higher speed processors may be able to accomplish a heavier workload in this time but 16-bit processors may be sufficient for many applications.

As with 8-bit processors, speed is not the only consideration in choosing a hardware system. 16-bit processors often have more internal hardware available that can aid in running the RTOS and its applications. More internal general registers are available as well as more timers with higher resolutions. This can accommodate applications that having strict timing requirements. Often other issues are more of a factor. 16-bit processors usually handle interrupts in a sufficient manner. They usually have a sufficient number of interrupts or its response time meets the specifications. As with some 8-bit processors, this is not an issue but it must be known how the RTOS handles the interrupts with respect to microkernel services.

16-bit processors can usually accommodate faster devices with wider data paths. This eliminates any need for external interfaces for these devices. Limited file systems or other forms of storage may be an option. Still, many of the newer devices have data paths and speed requirements that surpass those available with a 16-bit processor.

16-bit processors have several other advantages. Advancement in chip technology has brought the price of these processors down drastically, making them a viable option to an 8-bit processor. Small, embedded system designs can incorporate 16-bit processors at a marginal cost. They also get more out of the RTOS kernel services and see greater improvement in system performance. The power specifications of 16-bit processors may also meet the requirements of many portable applications. In the past few years, 16-bit processors have been the choice for many applications and can be considered a proven technology. Because many of these processors exist, the only real cost of implementing an RTOS or real-time system environment is software and time intensive. As with its 8-bit predecessor, 16-bit processors may lack the performance of 32-bit processors but may still meet the needs of many real-time applications. Motorola has recognized this and has built the M.Core that reduces power consumption and improves the performance by using the 32-bit architecture and a 16-bit instruction set.[13]

2.2.2.1 Example RTOS

Motorola's RTEK Kernel , used in developing real-time applications, is classified as a deterministic, multitasking real-time kernel.[14] Its API (Application Program Interface) supports C and assembly language; thus, kernel services are accessed in a familiar way. The following is a list of its features.

- Supports AMCU microcontrollers products
- Event driven operation
- Multitasking with selectable scheduling methods
  - Preemptive
  - Round Robin
  - Time-Sliced
- Static and dynamic kernel objects
  - Tasks
  - Semaphores
  - Queues
  - Mailboxes and Messages
  - Memory Partitions
  - Exclusive Access Semaphores (Mutexes)
  - Timers
  - Interrupt Handlers
- Fixed or dynamic task priorities
- Inter-task communication and synchronization
  - Semaphores
  - Mailboxes and Messages
  - Queues
- Timer management
- Timeout options for many services
- Partitioned memory management
- Efficient interrupt servicing
- Fast context switch
- Small RAM and ROM requirements
- Standard programmer interface in C on all processors
- Highly flexible configuration to custom fit the application

RTEKgen is the development tools that handle the host development and downloading of code to the target system. It has many default values for the RTOS such as number of

tasks or creation of semaphores and communication structures. It provides a comfortable GUI to interface the RTOS to the user. Figure 2 gives a snap shot of the RTEK interface.

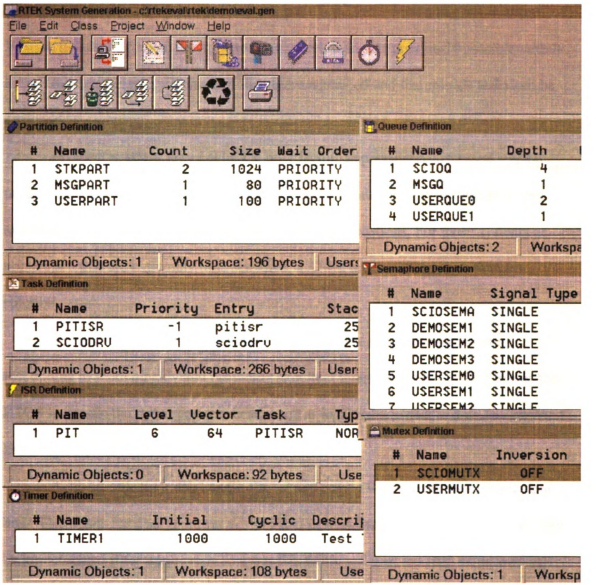


Figure 2. RTEK System Generation Tool.

2.2.3 32-bit Processors

Most of the advanced embedded real-time systems make use of 32-bit processing technology. Often, these same systems run most of today's desktop computers. They out

perform any of the other available 8-bit or 16 bit processors for embedded processing and for hosting an RTOS. 32-bit processors incorporate such features as multithreading and advanced memory management. They can satisfy the most demanding applications. When a task is scheduled, it completes more work than was previously possible with 8-bit or 16-bit processors. Because of this, more advanced and feature rich tasks can be designed.

Processing speed may not be the only factor, as previously discussed. 32-bit processors have the most internal hardware. They have the highest resolution timers available that can aid in running the RTOS and its applications. For those systems or application that are extremely time sensitive, a 32-bit processor provides the best solution.

These processors usually handle interrupts in the most optimal manner, responding to them quickly and providing excellent interrupt handling. This leaves the RTOS to handle the remaining overhead, such as task management and nested interrupt handling.

A 32-bit processor can handle almost any device with a wide address and data bus. This opens the door to newer devices and new applications. Full file systems may be implemented if the application requires this feature. Other I/O such as networking interfaces may also be part of the design.

The memory constraints of the other systems are not a concern for this 32-bit processor. Almost any kernel service can be implemented with a 32-bit processor, the only impediment may be either financial costs or lack of available development time.

Financial cost is often the major obstacle to using a 32-bit processor, but even this changes with time and many systems with 32-bit processing can be used at a reasonable cost. Some other obstacles include large power requirements and size restrictions. Larger embedded system designs are now incorporating 32-bit processing and maximize the use of RTOS kernel services. 32-bit processors are becoming the *de facto* processor for many embedded applications and most RTOSs support the majority of available 32-bit processors. These 32-bit processors do exist in some embedded systems without an RTOS and integrating one would prove to be beneficial.

2.2.3.1 Example RTOS

VxWorks™ of WindRiver Systems leads the RTOS market. One of its more notable applications is NASA's data and control application for its Mars Pathfinder. The following gives a list of the powerful features of VxWorks™.

- Efficient task management
  - multitasking, unlimited number of tasks
  - preemptive and round-robin scheduling
  - fast, deterministic context switching
  - 256 priority levels
- Fast, flexible inter-task communications
  - binary, counting and mutual exclusion semaphores with priority inheritance
  - message queues
  - POSIX pipes, counting semaphores, message queues, signals and scheduling control sockets
  - shared memory
- Fast, efficient interrupt and exception handling
- Optimized floating-point support
- Dynamic memory management
- System clock and timing facilities

Network Support

- Complete TCP/IP networking, including Zbuf (No-Copy TCP)
- Communications across various media (Ethernet, serial, backplane, custom)
- Sockets
- Remote login (rlogin, telnet)
- Remote Procedure Calls (RPC)
- Network File System (NFS) client & server
- File Transfer Protocol (ftp/tftp client & server)
- Remote command execution (rsh)
- BOOTP booting protocol
- SNMP (MIB-II support)
- STREAMS

Fast, Flexible I/O and Local File System

- Complete portable I/O system
- POSIX asynchronous I/O and directory handling
- SCSI support
- Extended MS-DOS and RT-11 file systems
- Raw disk file system

Target Development Features

- Full ANSI C compliance and C++ support
- Extensive POSIX 1003.1, .1b compatibility
- Interactive, C-interpreter target shell
- Symbolic debugging and disassembly
- Powerful performance monitoring
- Extensive kernel, task, and system information utilities
- Dynamic linking loader
- Libraries of over 1100 utility routines
- Flexible booting from ROM, local disk or over the network
- Highly scalable design allows for wide range of applications

Another advantage of VxWorks™ is its modularity and scalability. This allows the RTOS to be tailored for a system with unique specifications or constraints. It does so by keeping the design of the kernel layered, adding the support modules as needed.

It also has a rich set of development tools designed around a host/target environment, using the Object Oriented C++ language for development. Host-based tools included with VxWorks™ are a cross-compiler along with a powerful remote source-level debugger VxGDB™. Target tools include an interactive C-interpretive shell and linking loader for prototyping, as well as libraries of over 1100 utility routines. Optional VxWorks™ accessory products include VxVMI™ for virtual memory interface, VxMP™ for multi-processing and VX-Windows for graphics support. Optional WindPower™ host-based tools include the WindView™ real-time dynamic system visualizer, the StethoScope real-time data monitor, WindC++ ™ (including the popular iostreams class library), WindC++ Gateway for ObjectCenter, and the VxSim™ simulator.

2.3 Summary

A powerful RTOS is the key to a successful real-time system. It makes use of proven hardware with a well-designed and re-useable real-time kernel. This eliminates the need for development time and money toward building a custom operating system. Often these commercial operating systems meet very high standards due to the immense competition and stringent requirement of real-time systems. They far surpass the quality of many desktop operating systems.

Yet many engineers that don't develop an operating system for the desktop feel they must develop their own RTOS. In the past there has been no alternative and applications were not as sophisticated. However, widely available and high quality RTOSs can now be incorporated into these complex embedded designs. The trend is shifting to designers using an RTOS to develop more complex applications.[15] The task

may be a daunting one for those that have to port over an entire system and it may not be practical to use an RTOS. For those that build a new system or re-design is possible, an RTOS is a great way to improve and optimize their real-time system. Most vendors attempt to aid in this process by developing user-friendly development tools and provide turnkey design solutions for these complex embedded applications.[16]

A RTOS aids in designing and building new systems that have real-time requirements or optimizing and enhancing existing real-time systems. Implementing these systems only requires that application specifications fit well with the hardware and appropriate RTOS kernel services.

Chapter 3 Design & Implementation of μC/OS

3.1 Introduction

We present the implementation details of μC/OS and how it was ported to the Handy Board. We explain some of the reasoning behind both our hardware and software choices related to the implementation. A more in depth explanation of what the Handy Board consists of and what the MC68HC11 processor has to offer is also discussed. The development environment is presented as well. It was a key in porting μC/OS and developing its applications.

3.2 Target Architecture

The MC68HC11 serves as the target architecture for this thesis and the Handy Board's system incorporates this processor into a more useful system. This processor is widely used in MSU engineering labs and in products throughout the embedded industry; thus, implementing μC/OS for this processor has merit and usefulness. Implementing μC/OS for the Handy Board, which incorporates the MC68HC11, will help to bring a real-time development environment to the MSU engineering labs.

3.2.1 MC68HC11

The MC68HC11's 8-bit embedded architecture has proven to be a solid design in the embedded industry; thus, using μC/OS for the MC68HC11 is industry motivated. Many robotic and control applications could make use of the real-time kernel services. Some of the features that make the MC68HC11 a well-suited embedded controller are its timer functions, prioritized interrupts, digital ports and on-board A/Ds.

The MC68HC11's real-time interrupt capabilities provide μC/OS with a periodic interrupt that μC/OS uses to schedule tasks. The RTI (Real-Time Interrupt) is based off the MC68HC11's crystal of 8.0 MHz (E = 2.0 MHz) and can be configured for various interrupt periods (see Table 1 for listing of rates). It ranges from 4.10 to 32.77 millisecond interrupt rates and is selected by two bits in a control register for the RTI.

A fast rate may cause too much system overhead and not allow a task to complete a reasonable amount of work. On the other hand, not interrupting often enough may affect the timeliness of the application. A reasonable rate of 8.19 milliseconds allows the task to execute approximately 1700 lines of C code(assumed average of 3 assembly instructions per line of C code and each instruction takes 3 cycles) , a good amount of work for most tasks.

Table 1. Real-time interrupt rate for MC68HC11

RTI Control Bits RTR1	RTR0	Rate is E divided by	Interrupt Period (in milliseconds for 8.0 MHz crystal)
0	0	2^{13}	4.10
0	1	2^{14}	8.19
1	0	2^{15}	16.38
1	1	2^{16}	32.77

The MC68HC11 saves the context of the processor during an interrupt by storing the register to where the SP (Stack Pointer) points. Upon return from the interrupt the registers are restored from the SP location. The µC/OS scheduler is aware of this process and uses it to save and restore the tasks' context.

The MC68HC11 provides a built-in serial interface. To begin using, it only needs to be enabled and set to the proper baud rate via a serial configuration register. The serial interface can be used to debug and developing code through *printf()* statements. It may also be used to communicate with other devices that are part of the real-time design.

All register and ports within the MC68HC11 are memory mapped. External devices may also be memory mapped if they have access to the processor's data bus, as with the Handy Board. This allows for easy access to external devices and configuration of the MC68HC11.

3.2.2 Handy Board

The MC68HC11 by itself is a useful and prominent component in the embedded systems industry but it not a complete system. The Handy Board incorporates the processor into its design giving the MC68HC11 added value. The following is a list of Handy Board features.

- socketed 52-pin 6811
- 32K battery-backed static RAM that is rechargeable
- digital input latch
- digital output latch driving two L293 chips
- 14-pin LCD interface
- two user pushbuttons
- powered/polarized individual sensor connectors
- 40 kHz IR output drive
- one servo motor output
- External Serial Card for easy access to serial communications. It communicates with the Handy boards on board serial interface through a standard telephone wire.
- LCD Display

Table 2 gives the memory map that allows the Handy Board to communicate to the outside world. First, the data bus is made available via memory mapped address ranging from $4000-6fff. Second, other interfaces exist to facilitate expanded memory and motor control. Also, the MC68HC11's registers are memory mapped. Finally, the MC68HC11 interrupt vector bank is located at $bfc0 to $bfff for the Handy Board rather than the normal range of $ffc0 to $ffff.

Table 2. Handy Board memory map[17]

Device	Location	Notes
6811 internal RAM	$0000-$00ff (A1chip) $0000-$01ff (E1chip)	Built-in
6811 control registers	$1000-$103f	Built-in
Expansion I/O Bank 0	$4000-$4fff	Memory reads in this range enable the Y1 latch selector, present on the HB Expansion Bus. Memory writes in this range enables the Y0 selector on the Expansion Bus. No devices are present on a stock Handy Board.
Expansion I/O Bank 1	$5000-$5fff	Reads enable Y3; writes enable Y2. See explanation above.
Expansion I/O Bank 2	$6000-$6fff	Reads enable Y5; writes enable Y4. See explanation above.
Digital inputs	$7000-$7fff	The digital inputs consist of the two switches and sensor ports 10 through 15. A memory read from anywhere in this range returns the value of the digital input byte from the data bus.
Motor outputs	$7000-$7fff	A memory write to anywhere in this range controls the motor outputs. The low four bits are motor direction, and the high four bits are motor enable (1=on).
External RAM	$8000-$ffff	The 32K of battery-backed memory is mapped to the upper 32K block of the 6811 address space.

The Handy Board pin out is represented in Figure 3 and lays out the MC68HC11 interface. The Y0-Y1 pins control external latching of the data bus, D0-D7. The latch is controlled as depicted in Table 3. For example a read from address $4000 will bring

The Y0-Y1 pins control external latching of the data bus, D0-D7. The latch is controlled as depicted in Table 3. For example a read from address $4000 will bring R/W' low and will set Y0. Setting Y0 could trigger an external latch to latch in data from an external device.

Table 3. Latch control for data bus.

R/W'	Address Range	I/O	Latch selector
0	$4000-$4FFF	O	Y0
1	$4000-$4FFF	I	Y1
0	$5000-$5FFF	O	Y2
1	$5000-$5FFF	I	Y3
0	$6000-$6FFF	O	Y4
1	$6000-$6FFF	I	Y5
0	$7000-$7FFF	O	Y6
1	$7000-$7FFF	I	Y7

3.3 Development Environment

The design environment consists of the Whitesmiths tools, CodeWright and Telix. These tools provide reading, modifying and processing of application source code. They provide an interface to debug the application and for communication via a serial port.

3.3.1 Whitesmiths Tools

The DOS-based Whitesmiths tools cross compile, link and load into the processor the µC/OS kernel and its applications. These tools handle the entire process of taking the source code in standard ANSI C to the necessary Motorola S-record format (the ASCII format downloaded to the processor). These tools also provide a debugger to aid in the development process.

3.3.1.1 Compiler and Linker

The compiler is the key in converting the RTOS source code into object form for conversion into machine language. In this case, it represents a key aspect in how the RTOS functions. The compiler places pointers to ISR functions into the vector table. This is often done via *extern* definitions of the functions that the programmer defines in another source file, in this case the µC/OS source code.

The compiler defines how functions are called and how arguments are passed down to them. This is important when porting the RTOS because when context switching or scheduling a task, the RTOS must emulate how the compiler would normally operate.

The RTOS must be concerned about stack usage because many compilers use a stack to store local variable and pass arguments. The stack represents the context of the processor or task and the stack must be preserved during scheduling and context switching. The compiler also provides an assembly language interface for the RTOS to controller the processor where high level code is inefficient or inadequate.

3.3.1.2 Debugger

The Whitesmiths' debugger provides much insight in porting μC/OS and developing its application. When the compiler produces code, it may also produce extra information for debugging. CXDB, the Whitesmiths' debugger, uses this information to create a simulated environment that allows the developer to dissect and analyze the RTOS and application code. It allows the programmer to step through routines, check that the status of tasks (by reading the Task Control Block structure variable in μC/OS), and simulate I/O or interrupts via memory reads and writes. This can be done at the high level, C source code, or the low level, assemble code. This gives the developer insight into the application and allows him/her full control of it.

3.3.2 Support Tools

A simple COM port program, Telix, provided an interface to the Handy Boards serial interface. Telix captures all data from the serial port and engineers can make use of *printf()* functions to develop and debug an application.

CodeWright provides a customizable editor to modify the source code for the applications and µC/OS. This may not seem important, but a good editor can save time and make a programmer feel more comfortable reading and modifying source code.

3.4 The RTOS, µC/OS

This kernel had originally been designed specifically for the 8-bit processor but is capable of running on other higher performance processors. Certain portions of the µC/OS are done in Motorola assembly but it is kept to a minimum to maintain the RTOS's portability. µC/OS only requires that the processor have a stack and its registers be available. It also provides many of the essential kernel services that help maintain a real-time environment. It implements the core features of many RTOSs. We present µC/OS in this section beginning with its handling of system resources. We then discuss how µC/OS manages its tasks and the communication among them. Interrupt handling is discussed and we finish with the performance of the kernel services.

3.4.1 Resources

Real-time systems contain many resources that various applications require in order to complete their work. This can range from I/O devices to abstract concepts such as shared arrays. Shared resources are available to all tasks or a subset of tasks and must be managed to avoid corruption or other problems that may include dead lock. Deadlock will occur when two tasks hold a resource the other needs in order to continue and each will wait indefinitely for the needed resource.[18]

Semaphores provide a method by which tasks can synchronize or gain access to a resource. Once the semaphore is obtained, the task can continue executing. Two types of semaphores can be used. Binary semaphores follow the key/lock concept. This allows a task to gain access to resources. Once a semaphore for a given resource is obtained by a task its state transfers to the locked state. Any other task attempting to obtain the semaphore for that resource will see that it is locked and wait for it to be unlocked. Thus, the semaphore has a binary value or state of lock or unlocked.

Counting semaphores goes beyond the idea of binary semaphores and provides multiple valued semaphores. Its size is determined by its bit width i.e. an 8-bit semaphore would have 256 possible values. Tasks that attempt to use the semaphore when its value is zero are put on a priority waiting list under µC/OS. On the other hand, if the semaphore is positive then the resource is made available. When the semaphore becomes available, µC/OS schedules the highest priority task and decrements the semaphore. When the task releases the semaphore the scheduler increments the semaphore's value.

3.4.2 Tasks

μC/OS defines a task to be similar to a thread but the task is an independent process. A thread often resides within a process, sharing control of allocated memory and resources. A task, like a thread, must be concerned about resources that may be shared among tasks. It can also synchronize with other tasks, much like threads. Tasks are different in that they do not automatically inherit any resource as threads do. Threads are often time sliced where tasks, in μC/OS, are priority-based. It is up to the designer to assign these priorities based on the needs of the application.

3.4.2.1 Task Priority

The priorities within μC/OS are unique, no two tasks can have the same priority. The highest priority matches the numerically lowest priority number; thus, the highest possible priority is "0" and the lowest "63". This gives the application 63 possible tasks to accomplish its goals, the last priority is usually reserved for the "idle" task. The "idle" task runs when all other tasks are inactive for whatever reason and μC/OS needs a task to fill the time slice until another task is ready to run. The context of each task entails the current CPU registers, its own stack and a priority. A task can be static or dynamic in nature. It can be statically assigned a priority on creation and can not change. μC/OS allows the priority to change during execution making it a dynamic priority. Figure 4 illustrates the task context within μC/OS.

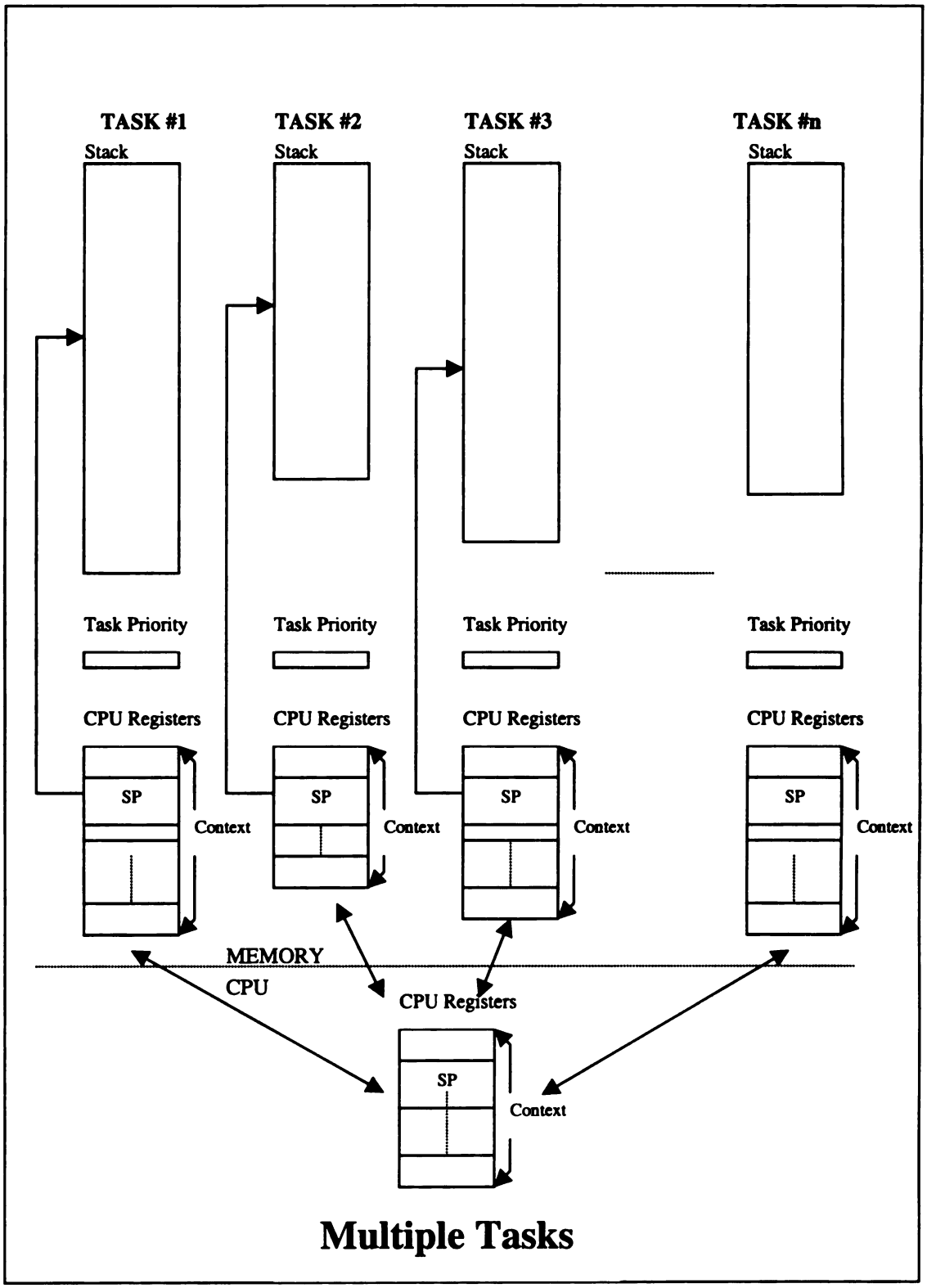


Figure 3. Task context.

Figure courtesy of Jean J. Labrosse, © R & D Publications, 1992. All rights reserved

3.4.2.2 States

To implement these tasks in a multitasking environment requires that each task be assigned a particular state. The possible states of a task are Dormant, Ready, Running, Delayed, Waiting For An Event or Interrupted. If a task's state is Dormant, it is not ready to take control of the CPU and the kernel has not made the task available to run. A task in the Ready state wants to take control of the CPU. The kernel will allow it as soon as the current tasks with higher priorities are not scheduled to execute. The Delayed state translates to a task that has suspended itself for a period of time. If a task is in the Waiting For An Event state it will not execute or attempt to execute until a particular event has occurred such as a semaphore becoming available or a message resides in a mailbox, etc. Lastly, the Interrupted state occurs when a task looses control because of an interrupt that must be serviced. Figure 5 shows the flow among task states.

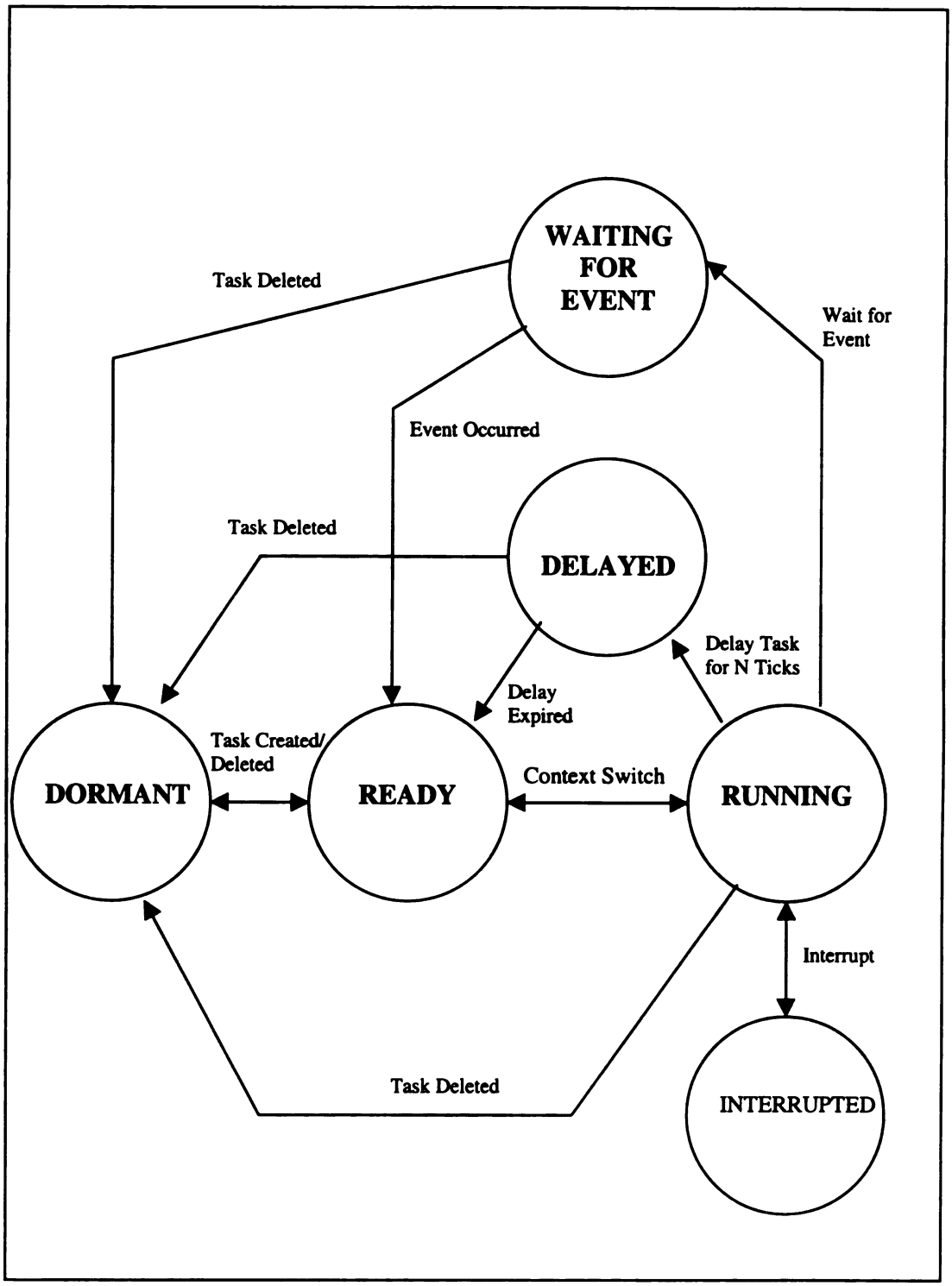


Figure 4. Task states for μC/OS

Figure courtesy of Jean J. Labrosse, © R & D Publications, 1992. All rights reserved

3.4.2.3 Task Control Blocks

Because μC/OS incorporates a preemptive scheduling algorithm it must keep track of the task control information. This is accomplished via a structure called the Task Control Block (TCB). This contains all the necessary information to suspend or interrupt a task and to bring it back into execution. The TCB is a table with entries that keep track of the state of the task, priority and stack information and TCB control management fields such as previous and next entries. This aids in the scheduling process of the tasks. The "idle" task created by μC/OS is also placed in the table. The kernel keeps track of available entries for new tasks and manages the deletion of tasks from the table.

3.4.2.4 Task Scheduling

A preemptive, priority-based scheduling algorithm is implemented by μC/OS. The kernel determines the highest ready-to-run task and executes it. All other tasks are scheduled based on their given priorities. This type of scheduling better matches real-time applications because system response is higher. The current task will be preempted to allow a higher priority task immediate access to the processor. Thus, the higher priority tasks have a better chance of meeting deadlines specified for the real-time application. This creates a more deterministic system.

Extra care must be taken with this design because shared functions, data and resources can be corrupted when a task is preempted and returns to execution. For example, a task could be operating on a global variable when it is preempted. It will assume the value has not changed when it begins execution. However, while it is preempted, another higher priority task may manipulate the very same global variable causing data corruption. One can maintain the integrity of shared resources via semaphores.

3.4.3 Communication

Frequently, tasks need to communicate among themselves. The kernel services should provide the vehicle to communicate. µC/OS provide this service via mailboxes and queues. Both are consider events that a task triggers via a POST or waits for the event via a PEND. When a task PENDs, µC/OS puts the task on a priority waiting list similar to the semaphore list. When the event occurs, the highest priority task in the list takes control of the processor and retrieves the message from a mailbox or queue.

3.4.3.1 Event Control Block

µC/OS uses a structure similar to the TCB called the Event Control Block to control mailboxes, queues and semaphores. It is used in scheduling tasks for the various events such as scheduling the next task for an available semaphore.

3.4.3.2 Mailboxes and Queues

Mailboxes contain messages POSTed by other tasks. The application defines what type of message resides in the mailbox and other tasks that use the message must be aware of the type. For example, if a character based message is stored by one task all others must be aware that the message is character based when they retrieve it. µC/OS provides the kernel services to create a mailbox, PEND to the mailbox and POST to it. ISRs must never make calls to these services because an ISR does not have the required task information i.e. they cannot be put on a priority waiting list. A timeout can be specified when waiting for the event, but an application can specify that it will wait indefinitely. The mailbox can only contain one message at any given time and an error will occur if this is violated.

µC/OS implements queues to extend the mailbox concept. A queue can contain multiple messages and operates as a standard queue with a FIFO (First-In First-Out). Tasks manipulate queues in a similar fashion as mailboxes with a creation function and POST/PEND functions. µC/OS manages the tasks as it did for mailboxes but allows multiple POSTs to the queue. Figures 6 and 7 illustrate the flow of mailboxes and queues.

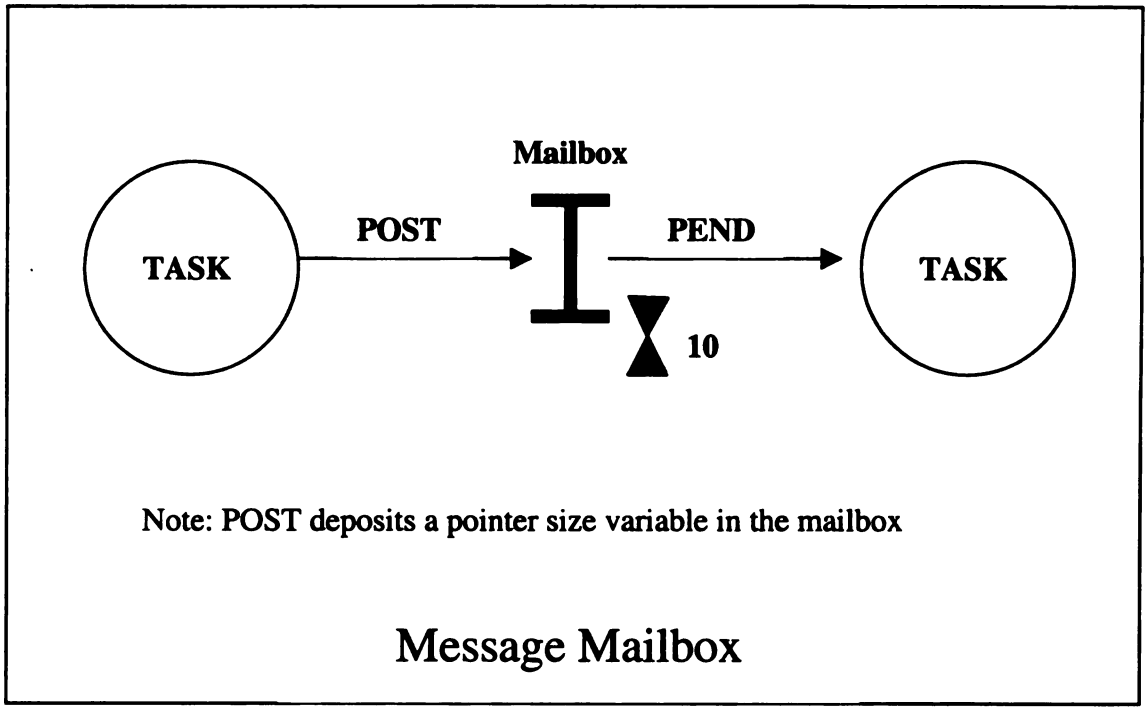


Figure 5. μC/OS mailbox

Figure courtesy of Jean J. Labrosse, © R & D Publications, 1992. All rights reserved

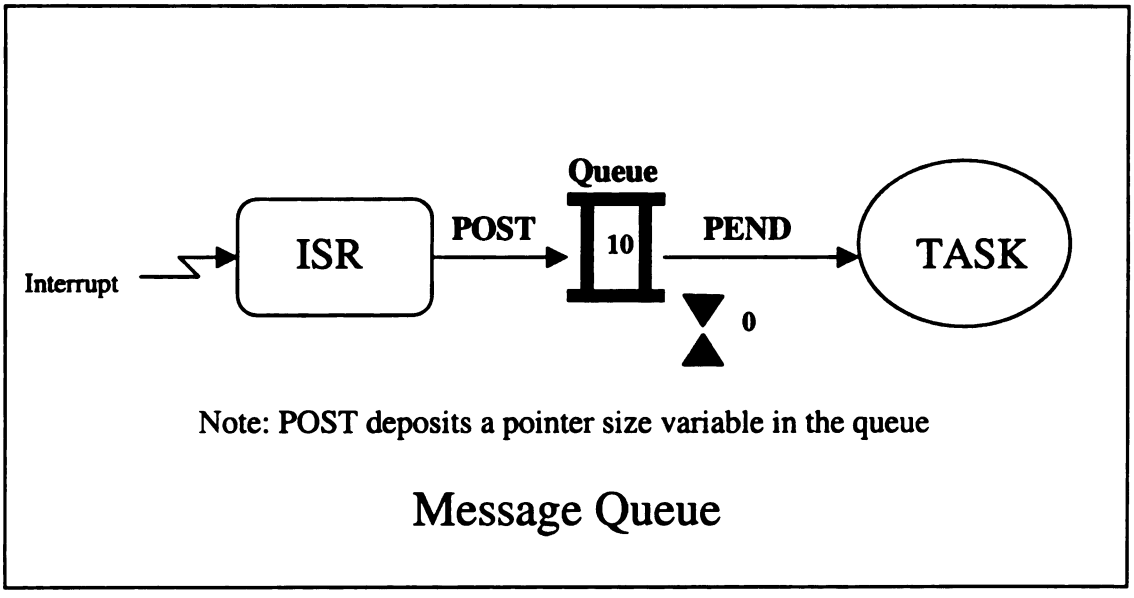


Figure 6. μC/OS message queue

Figure courtesy of Jean J. Labrosse, © R & D Publications, 1992. All rights reserved

3.4.4 Interrupts

Interrupt handling often requires an assembly language interface and is handled by an ISR. Applications often require fast interrupt response and handling, but this depends on the specific architecture of the processor. Writing the ISR that interface to the RTOS in the assembly language of the target processor can greatly increase interrupt response time. When an interrupt occurs, the current task context must be saved and other interrupt handling overhead will be incurred such as re-scheduling.

The scheduler makes use of one of the processor's periodic interrupts for a system clock. This allows for periodic re-scheduling of the tasks i.e. multitasking. The system designer can optimize the frequency of this periodic interrupt so that the tasks complete their work while meeting their deadlines.

An 8-millisecond interrupt was implemented to act as µC/OS's periodic interrupt for scheduling the tasks. µC/OS does not handle this interrupt as it does other interrupts. It performs a software context switch as opposed to a typical interrupt context switch. µC/OS expects this interrupt and knows the state of the system when this interrupt occurs. Other interrupts can occurs at more random or unexpected times and the interrupt handling mechanism must reflect it. µC/OS uses this periodic interrupt to provide timing services such as its delay and clock functions. Figure 8 reflects how µC/OS handles interrupts.

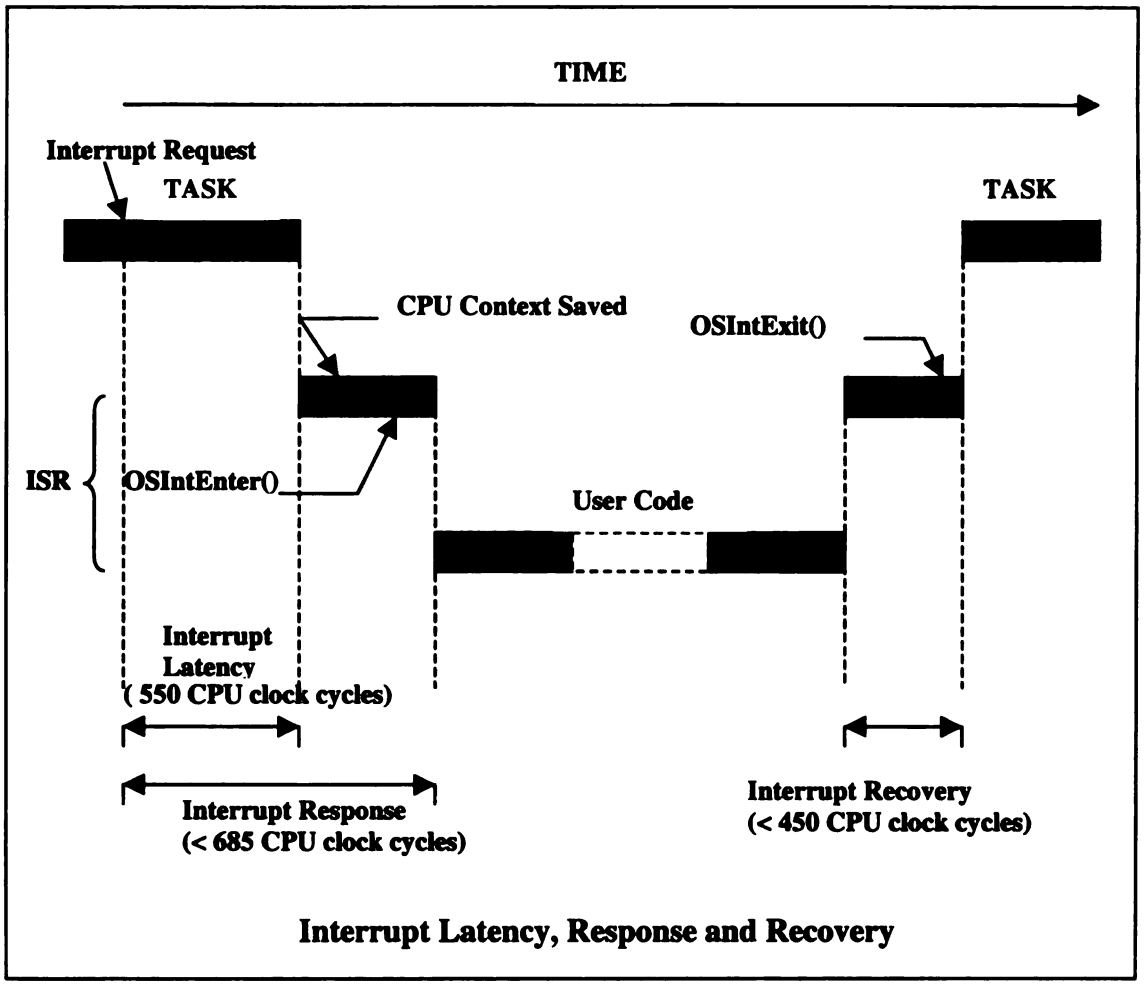


Figure 7. μC/OS interrupt handling

Figure courtesy of Jean J. Labrosse, © R & D Publications, 1992. All rights reserved

3.5 Development Challenges and Solutions

The following sections discuss some of the challenges that arose from porting μC/OS to the MC68HC11. The following sections have been included to facilitate better use of μC/OS and to avoid possible pitfalls or problems when re-porting μC/OS to another processor.

3.5.1 Task Argument Passing

Tasks are initialized in μC/OS upon creation by parameters that are passed to it as arguments. The argument to the task function had been invalidated due to μC/OS's assumptions about how the compiler passed arguments for the MC68HC11. This problem was observed by attempting to print the parameter passed into the task. The argument is passed a void pointer parameter in order to accommodate all data types; thus, by default it is passed by reference. This adheres to the ANSI C language standard but is handled by a processor dependent process. This process determines how the pointer is stored in memory and passed to functions.

The *main()* function calls a *TaskCreate()* function for each task. This sets up the individual tasks stack, priorities and all other TCB information including the task's initial parameters. Argument passing is processor dependent in the sense that the compiler uses different architectural features to pass parameters, handle variables and other language details; thus, setting up μC/OS and its tasks involves understanding how the compiler operates.

For the MC68HC11, the Whitesmiths tools compiler passed arguments by reference in the 'D' register. This register consists of the 'A' and 'B' accumulators. The 'A' accumulator is in the high order byte and 'B' in the low order. Each task has an associated stack that holds the argument and context of the task. Part of the context of the task is the value of the accumulators and PC (Program Counter) that includes the 'A' and 'B' accumulators. These two accumulators are 8-bits wide and together they form a word, while the other accumulators and the PC are 16 bits wide; thus, when the stack is being set up with its data (arguments), it is done so in word fashion as shown in Table 4. The following code segment assigns the stack pointer to the argument.

***stk = pdata;** where **stk** currently points to the beginning address of 'A' and pdata is the argument.

Table 4. Stack snapshot, SP begin at 0x8900.

Stack location	Memory	Value
	0x8908	PCL (lower byte of Program Counter)
	0x8907	PCH
	0x8906	IYL
	0x8905	IYH
	0x8904	IXL
	0x8903	IXH
SP ->	0x8902	A
	0x8901	B
	0x8900	CCR

Function calls use the 'D' accumulator, concatenation of 'A' accumulator (MSB) and 'B' accumulator (LSB), and the stack for handling the void pointer argument. The method used to call a task (which is a function) is to pull from the task's stack the PC with its associated accumulators and begin executing. The PC does not point directly to

the task but to some function entry code inserted by the compiler that pulls the arguments from the 'D' accumulator and does other function entry work such as variable initialize.

The compiler can use the stack for local variables but the option of using local RAM as storage was taken; thus, leaving the stack with only the return address and function arguments. The scheduler must setup the 'D' accumulator and the PC to call the function entry code. It does so by using the 'Pull' instruction to pull the accumulators and PC from the task's stack into the processors accumulators and PC. The 'Pull' instruction starts at the top of the stack. It then works its way down the stack. A 'Pull' on the 'D' accumulator occurs for parameter passing. In this case, the 'A' accumulator is put in the lower byte and 'B' in the upper. This causes the pointer address to be inverted. Thus, the pdata may be 0x8020 (actual argument address during debugging) gets turned around when it's pulled off the stack and it looks for the argument in 0x2080.

Stepping, via the debugger, through the task initialization and discovering the proper pointer address order for the task's arguments solved this challenge. After this initialization, µC/OS executes its assembly level scheduling process that actually performs the task level context switch. This bring the highest, ready-to-run task into the execution state i.e. runs the task. In order to accomplish this context switch, the scheduler pulls the task's stack from the OSTCBPrioTbl[] (which holds the pointer to the task's stack) into the processors stack and begins executing the appropriate task. When the processor's stack is replaced with the task's stack, it resumes executing where the task forfeited execution.

In stepping through this process, it became apparent that the argument pointer inverts when pulled from the stack for task execution. Initially, a print statement that

attempted to show the value of the argument alluded to the problem. The value should have been an ASCII '1' but printed as some unreadable character. The next step entailed involving a debugger to discover the true root of the problem.

The debugger brought out the problem and the solution later came at the assembly level but a first attempt came in the *TaskCreate()* function. In order to resolve the problem, pointer manipulation must be used to switch the two bytes of the word-sized pointer. The crux of this problem lied with the compiler that fails to allow any useful manipulations of the argument pointer. The assignment could not be made any other way (word wise) due to the fact that the stack had to be built from top down as implemented in the MC68HC11.

The solution normally would be to fill in the stack a byte at a time but as fore mentioned the compiler would not allow byte access or manipulation of the pointer argument. If this feature existed in the compiler, then two byte wide variables could be assigned to the high and low order bytes of the address respectively. The stack could then be built with these values. Because of this roadblock, the solution lied with the assembly level context switching process.

The first step in the assembly solution came from the fact that the processor stack was replaced with the highest priority task's stack; thus, the task's SP (Stack Pointer) was available at the assembly level. This made manipulation of the argument pointer simpler and faster than creating a new process to manipulate the OSTCBPrioTbl[] directly. Before the 'RTI' (Return from Interrupt) instruction that actually performs the processor context switch, assembly code switched the bytes appropriately e.g. accumulators 'A' and 'B' where place in their proper order to be inserted into the 'D' register. This guarantees

that even if the argument had been a value and not a pointer, it would still be applied in the proper sequence.

3.5.2 Print Services

Print services became available after configuring the serial interface on the MC68HC11. It required that the configuration register for the SPI (serial peripheral interface) be set up for a given baud rate and the transmit and receive be enabled. The compiler comes with built in functionality to handle this interface to the SPI such as *printf()* and related routines. Because the actual hardware interface does not exist in the debugger, these status bits must be set manually to simulate working hardware. It can be done while stepping through the program by writing the expected status bits to the memory mapped SPI register.

3.5.3 The Debugger Interface

The debugger aided immensely in pointing out flaws in porting µC/OS to the Handy Board. It also gave insight into how the RTOS operated, how it handles task management, resources, etc. and how it handled the Handy Board/MC68HC11 interfaces such as the SPI. In order to use the debugger, the compiler flags for producing debug information were required. One of its uses came in simulating the status bits of the SPI. Simulating the I/O became as simple as setting a memory location to a given value.

For example, the SPI contains a transmit done bit in the status register. Normally the print functions would transmit a byte to the SPI buffer and wait until the transmit done bit was set. It would then continue to feed the SPI buffer. In the debugger this does not happen because of lack of actual hardware. Because the debugger allowed memory

writes, the status bit could be set manually. As a result, the code could continue to execute as if the hardware truly existed and had responded.

One difficulty arose in attempting a context switching that is performed by a 'SWI' instruction or by a standard interrupt. The debugger assumes that the vector table resides at 0xffd6-0xffff, but the Handy Board re-locates this to 0xbfd6-0bfff. The linker assumes the code is destined for the Handy Board and places the vector table in 0xbfd6 via the linker file. Thus, when the debugger runs the code, the vector table must be copied or inserted into the proper place for interrupt processing to be simulated correctly. The following code segment gives the necessary instructions to modify the proper vector table entries with the correct addresses of the interrupt handlers. The first instruction is actually the simulation of the transmitter done bit being set. The second and third move the vector addresses to the proper place.

```
>mb 0x102e 0x80

>mw 0xfff0 0xf274
>mw 0xfff6 0xf230
```

3.5.4 Simulating Interrupts

The debugger has the ability to simulate interrupt processing. It accomplishes this by checking an internal debugger variable that represents the state of all of its allowed interrupts. Once the bit is set, it maps to a particular interrupt and the debugger performs a lookup into the vector table. Once the instruction pointer to the interrupt handler is found, the debugger begins executing instructions/code at that location. This mimics the MC68HC11 well enough to simulate interrupts manually i.e. they cannot be done

periodically. The '.irq' variable maps the interrupts to each of its 16 bits, refer to Table 5 for the bit mapping.

Table 5. Interrupt bit allocation for CXBD - MC68HC11[19]

Bit	Allocation
0	COP clock fail
1	COP COP fail
2	XIRQ
3	IRQ
4	Real Time Interrupt
5	Timer input 1
6	Timer input 2
7	Timer input 3
8	Timer output 1
9	Timer output 2
10	Timer output 3
11	Timer output 4
12	Timer output 5
13	Timer overflow
14	Pulse acc overflow
15	Pulse input edge
16	SPI
17	SCI

The RTI interrupt periodically interrupts the MC68HC11 based on the setup values in its configuration registers. It's based on the microprocessor's crystal, dividing it out to the desired interrupt cycle. When the interrupt occurs, µC/OS schedules the next ready-to-run, highest priority task and handles the timing service work. An 'SWI' interrupts the processor at the instruction level, not the device level, and does not appear in the debugger's interrupt table. When executing a 'SWI' instruction, the debugger indexes into the vector table and begins executing instructions/code from the interrupt vector location. Other MC68HC11 interrupts are handled similarly, such as an illegal instruction interrupt.

In order to actually simulate the interrupt, a switch to the debugger's low-level assembly interpretation mode requires the '\' command to be executed. To switch back to the high level of C language interpretation, simply execute the switch command again. The following code segment shows how to simulate the RTI interrupt, the "heart-beat" of μC/OS.

> \.Irq = 16 (16 = 0x10 which is bit 4, matching the RTI bit.)

By simulating this interrupt, the scheduling and task context switching could be dissected and monitored for insight and debugging. The failure of the RTI interrupt as it relates to task management blocked the initial port of μC/OS to the Handy Board platform. Task switching corrupted the current task's stack that created inaccurate and fatal code execution upon return to the task. Corrupting the stack essentially modifies the PC, taking the processor to a random location in memory to continue executing code.

The difficulty did not present itself initially due to the fact that a task ran once and passed off execution to another task. This task would continue correctly but the problem would occur upon return to the original task. In following the execution sequence, the proper method of giving up execution became apparent and a correction was made to how the scheduling algorithm worked. This came about after much scrutiny of the state of the individual task's stack before and after the scheduling process.

The corruption occurred because μC/OS had two context switching functions available for task management. The one interrupt handler dealt with a task being interrupted by some external device and the other a task level switch by the 'SWI' instruction. Either by oversight or lack of clarity from the author of μC/OS, the interrupt context switch had been implemented. The confusion may have stemmed from the fact that an interrupt occurred to switch the task but the interrupt context switching facility was not used because it was an expected interrupt. The interrupt handler manipulates the stack before it returns execution to the interrupted task. The handler removes superfluous interrupt handling functions that reside on the stack before returning to the task. In the case of simply performing a task level switch the interrupt handler corrupted the stack because it attempted to remove interrupt handling functions from the stack that were not present.

Implementing the task level context switching function relieved the problem and the task management operated correctly. This leaves the task's stack intact and processor execution continued correctly. Tables 6 and 7 illustrate the state of the stack in each scenario. Keep in mind that when μC/OS schedules a new task the current processor stack is stored as the state of the current task. If an external interrupt occurred , then in order to restore Task1, the stack must be cleaned of the interrupt handler functions. The real state of the stack represents that of a non-interrupted context; thus, the task level context switch should be executed. When the stack unwinds, it will resume execution somewhere within Task1 and not at some random location as it did with the interrupt context switching.

Table 6. Task level context switching example.

State of Processor	State of System Stack
Task1 release execution	Some point within Task1
TimeDelay() ends	TimeDelay() return statement
OS schedule() ends	OS schedule() return statement
SWI – processor actually switches	Task2

Table 7. Interrupt context switching example.

State of Processor	State of System Stack
Task1 interrupt	Some point within Task1
Interrupt handler() ends	Interrupt handler() return statement
Timer Services () ends	Timer Services () return statement
SWI – processor actually switches	Task2

3.6 Summary

μC/OS gives the 8-bit processors a taste of what RTOSs are capable of bringing to the embedded systems world. It does so efficiently, without sacrificing system performance or resources such as memory. It has remained fairly portable with only a small part actually requiring an assembly interface. It only had to make use of the MC68HC11's stack, its accumulators (registers) and a small portion of memory to function. All of this led to a small, efficient RTOS that helps applications written in ANSI C for the MC68HC11 meet their hard real-time requirements.

Inspiration for implementing an RTOS stems from new embedded applications that make use of new technologies in RTOS development for such work as space exploration. The Mars Pathfinder application runs the Pathfinder's entire system on one of the latest RTOSs, VxWorks™. This RTOS had not only been designed to meet the real-time constrains of such high standard as NASA's but it had a build in trace logging and a open design interface that allowed for "on-the-fly" modifications to its kernel services.

Since the Pathfinder's mission began on July 4, 1997, the system and VxWorks™ had run flawlessly. However, a few days later the system began to periodically reset with no obvious cause. The JPL engineers where able to use the trace log to determine that a shared resource, an "information bus", was causing a problem known as priority inversion.

This rarely occurred due to the nature of priority inversion. It only became evident after analyzing the trace log that showed a high priority task blocked on a resource held by a low priority task. This low priority task could not be re-scheduled due to a mid-level priority task blocking the low-level task from being scheduled and releasing the resource. Once priority inheritance was turned on by the JPL engineers via a user interface defined by VxWorks™ , the problem was alleviated. The low priority task inherited the high priority task until it released the resource and unblocked the high priority task. This inspires the development of similar RTOSs or at least to begin using some of the technology they offer.

Chapter 4 Test Application for μC/OS

4.1 Introduction

We present a train control application that deals with many classic real-time issues. The application requires its tasks to communicate and coordinate with each other as well as perform their work by a given deadline. It makes use of the kernel's semaphores to coordinate the tasks to complete the work. The application relies on the task priority scheme and preemptive scheduling to ensure that the most important tasks are completed on time and function correctly. The specification of this application classify it as a hard real-time application because the timeliness of the tasks is key to making the application work.

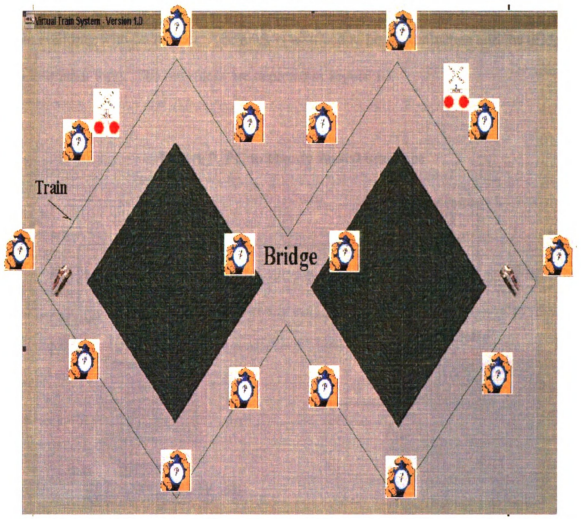
4.2 General Specifications

Automated train control can now be accomplished due to technologies such as radio-based train control that may make automated train control cost effective and an industry standard.[20] Because such complex real-time application will exist in the embedded industry, it "makes sense" to implement a similar application for μC/OS that interfaces to a simulated train environment.

The environment is as represented in Figure 9, which is an annotated screen shot of the simulation program written by Xiao Huang and Seth Mosier. Two trains travel around a given circular route of known length on separate tracks. The two tracks will have a bridge that both trains must cross and the bridge has one single track; thus, only one train can cross at any one time. The bridge will have a controller that can stop or slow the trains and allows them to pass on the bridge. Slowing the trains down takes a

certain amount of time. Because of this, the controller must slow down the train(s) early enough to avoid collision. Each of the train's length is fixed, and their position along the route is known only at certain points. At these points, each train's passage is time stamped and made available to the bridge controller. From this information the controller is able to control access to the bridge.

Figure 8. Train track representation.



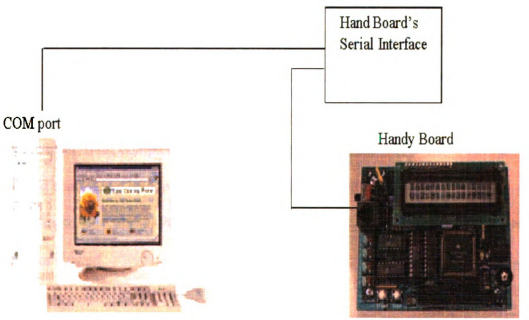
This represents the position marker with time stamps of the train.

This represents where the controller signals the train to slow or stop.

4.3 Environment and Design

The environment will consist of the Handy Board running a bridge control algorithm that uses a serial interface to communicate with a PC running a simulation of a train system. The PC will be running on Windows NT™ or Windows 95™ and gives a graphic display of the trains traveling around the track. The bridge control algorithm, written using µC/OS and running on the Handy Board, controls the trains to keep them from colliding with each other. The track is assumed to be 80 kilometers and the bridge is a kilometer long. The interfaces are depicted in Figure 10.

Figure 9. PC to Handy Board interface



4.3.1 PC Components

The PC components consist of a Visual C++ train simulation application using one of the available COM ports. The train application runs the GUI (Graphical User Interface) and simulates the trains running on the track and crossing the bridge. It controls the trains based on information coming from the COM port. The port brings in control information and the train application sends position information via the port. The COM port interfaces to the Handy Board to acquire the control information and update the Handy Board of the trains' positions on the track. This information is passed to the Handy Board via its serial interface.

4.3.2 Bridge Algorithm

When new position information is available, a time stamp is recorded for that train to determine its speed and destination time to the bridge. This information is used to determine when to signal a train to stop or slow in order to avoid a collision on the bridge. These timestamps are maintained and recorded until the train enters the bridge, then they are reset; thus, complete knowledge of the train's speed and position can be estimated and control can be applied as needed. The control is applied by communicating to the external PC simulator, via a serial interface, that the train will need to be halted or slowed.

4.3.3 Tasks

Three tasks run on the MC68HC11 that run the control algorithm and interface to the PC. A *main()* function creates these tasks with priorities suited for a bridge control algorithm. The gathering of position information by the Marker task is made the highest priority because missing this information could lead directly to a crash. Actually controlling the trains, done by the Controller task, is a lower priority because the control algorithm takes into account the possibility of getting the train position information late. It monitors their position and controls them before it really needs to, leaving room for mistakes or missed information. The Logistics task has a intermediate level priority and processes the information passed in from the Marker task and exports it to the Controller task. The following gives a description of each task and pseudo code that illustrates its function.

Marker task: This task handles the actual gather of the position data and time stamps of the trains as they pass the markers on the track. It passes the information onto the Logistics Task.

```
WHILE(1)
        Wait for the PC to have data ready;
        SWITCH( Input from PC)
                CASE  (Train 1)
                        Timestamp Train 1;
                        Signal Logistics task;
                CASE (Train 2)
                        Timestamp Train 2;
                        Signal Logistics task;
        ENDSWITCH
        Delay, Allow other task to proceed;
ENDWHILE
```

Logistics task: Computes position and destination time of each train based on information passed in from the Marker task. It communicates its results to the Controller task. The following pseudo code illustrates the function of this task:

```
WHILE(1)
        Wait for Signal from Marker task;
        Compute arrival time at bridge for each train;
        Signal Controller task that new train information is available;
ENDWHILE
```

Controller task: This signals the external simulator when a train needs to be halted or re-started based on information received from the Logistics task.

```
WHILE(1)
        Wait for Signal from Logistic task;
        IF (Train 1 AND Train 2 Arrival Time < Control Threshold)
                Control Train 1 via PC interface;
        ENDIF
ENDWHILE
```

4.4 Summary

We've developed a real-time application that is time critical and requires solid software development. It's a type of application that is exemplary of the applications that many engineering are implementing for real-time systems. This train application may serve as a base line application that many students can learn from, and expand on, for lab assignments. Also, it may be studied as a lead-in to further research in the area of RTOS development or co-design.

Chapter 5 Results

5.1 Introduction

µC/OS's power may not lie with the richness of its features but with the quality of kernel services. It provides the core features of a solid RTOS such as preemptive scheduling and task communication. µC/OS provides these services without much demand for large system requirements such as large quantities of memory or 32-bit processing capability. Its kernel has been written efficiently and provides the services in a deterministic manner.

These quality features are attractive to engineers using 8-bit processors such as the MC68HC11 because most of its application use assembly or small C (a subset of the full ANSI C) and µC/OS could provide them with a real-time kernel for the first time. We present some of the other applications that were written to test µC/OS's features and deterministic capabilities. One application tested some of the task communication features and timing functions and another implement priority inheritance using µC/OS's ability to dynamically change a task's priority.

5.2 Performance Metrics

Much of the metrics presented are not backed by extensive monitoring or testing but give a general idea of how well this RTOS may perform. Metrics exist that may relate to an RTOS but are not a measure of the kernel such as context switching; thus, they are not included. Many of these types of metrics are important to a designer when building a real-time system but should be kept separate from the evaluation of the RTOS itself.

Deterministic kernel service is an important performance metric and all services provided by μC/OS are deterministic[21] Its services execute in a deterministic time frame and each kernel service such as POST, has a known execution time given in terms of clock cycles.

Interrupt response can be crucial to a real-time application and because μC/OS has the ability to preempt tasks the response is quite fast. A real-time kernel must provide deterministic services to meet many of the real-time specifications.

μC/OS keeps memory usage to a minimum using the following formula to compute a particular application's memory needs:

RAM (in bytes)= 200
+ ((1+ OS_MAX_TASKS)*16)
+ (OS_MAX_EVENTS*13
+ SUM(Storage requirements for each message queue)
+ SUM(Storage requirements for each task stack)
+ (OS_IDLE_TASK_STK_SIZE)

Scalability for μC/OS is limited to the number of task that can be run or the addition of communications mechanisms such as message queues. If an application needs more tasks and needs them to communicate, then μC/OS can provide up to 64 tasks with mailboxes and message queue as a means to communicate. Modifying the kernel code to meet a specific need is the only way to scale it further. The code has been well documented and commented, making modifications to the kernel an easier task.

Interrupt latency and the response time to an interrupt event are important performance metrics. µC/OS disables interrupts to handle system services and gives the worst case interrupt latency and response by the following formula that is processor dependant:

Interrupt latency = Maximum interrupt disable time + time to vector to the ISR

Interrupt response = interrupt latency + time to save CPU registers + time to execute OSIntEnter().

5.3 Case Studies

These next two sections discuss some case studies that test some of the features of µC/OS. The first illustrates how tasks communicate and make use of the timing services of the kernel. The second implements a priority inheritance solution to the serious problem of priority inversion found in many real-time applications or systems.

5.3.1 Task Communication And Timing Function Application

Five tasks were created to illustrate the task communication: Task1, Task2,Task3, DispTask and KeyTask. These tasks swap messages and signal each other on certain events to complete their work. KeyTask would either POST a message to a mailbox or POST multiple messages to a queue depending upon user input. The following gives the pseudo code illustrating KeyTask's functions:

```
WHILE(1)
    Wait for user input;
    SWITCH(user input)
        CASE '1'
            POST message to a mailbox;
        CASE '2'
            POST two message to a queue;
    ENDSWITCH
ENDWHILE
```

Task1 would PEND on and then print the message coming from the KeyTask. It would then signal others that it received the message by POSTing to a semaphore. The following gives the pseudo code illustrating Task1's functions:

```
WHILE(1)
        Wait for KeyTask to put message in mailbox;
        Print message;
        Wait for access to global variable;
        Increment global variable;
        Release access to global variable;
ENDWHILE
```

Task2 operated similar to Task1 except it PENDed on the queue. Two other tasks were created to illustrate the timing functionality, Task3 and DispTask. Task3 delays for a second and updates a variable to keep track of the number of seconds. It then checks if a minute has passed and updates another variable to keep track of minutes. These were global variables that Task3 would signal to the DispTask, via a POST to a semaphore, when they were updated. The following gives the pseudo code illustrating Task3's functions:

```
WHILE(1)
        Wait for access to the global SECONDS variable;
        Increment SECONDS variable;
        IF (seconds variable = 60)
                Increment global MINUTES variable
        ENDIF
        Release access to SECONDS variable;
ENDWHILE
```

The DispTask would PEND on the semaphore and print the time in minutes then seconds. The following gives the pseudo code illustrating DispTask's functions:

```
WHILE(1)
        Wait for access to the timing and other global variables;
        Make local copy of global variables;
        Release access to global variables;
        Print or Display global variables to user;
ENDWHILE
```

This application shows some of the determinism of the kernel by properly tracking the time while providing inter-task communication services.

5.3.2 Priority Inheritance Application

Priority inversion creates a situation where a low priority task blocks a high priority task. This occurs if the lower priority task has taken control of a shared resource that the higher priority task requires in order to continue execution. This only becomes a problem if the lower priority task is kept from releasing the resource by other higher priority tasks. They would prevent the low priority task from being scheduled because they do not require the resource to continue. This situation is referred to as priority inversion because a low priority task blocks a high priority task, not the intent of a priority-scheduling algorithm.

The solution is to implement priority inherence and allow the low priority task to run temporarily by inheriting the priority of high priority task being blocked. It would then release the resource and unblock the higher priority task. The application written under μC/OS uses Task1 as the high priority task, Task2 as a medium priority task and Task 3 as the low priority task. A Monitor task was created, at a higher priority than all others, to implement the priority inheritance. Since two tasks can not have the same priority in μC/OS, Task3 was temporarily changed to a higher priority than Task1. A semaphore was used to represent the resource that Task 3 held.

A complete implementation of priority inheritance would require modification to the kernel code and may change some of the deterministic behavior; thus, great care must be taken if this is to be implemented into the kernel. The following lists the pseudo code that illustrates the function of each task in the priority inheritance scheme.

Task1:
```
WHILE(1)
        Delay for a second;
        Wait for access to the global resource;
        Release resource

ENDWHILE
```

Task2:
```
Initial Delay
WHILE(1)
        Perform other work not related to global resource;
ENDWHILE
```

Task3:
```
WHILE(1)
        Signal Monitor task;
        Gain access to global resource;
        Delay;
        Release global resource;
ENDWHILE
```

```
Monitor:
WHILE(1)
        Wait for a Task to obtain a resource
        IF (Priority Inversion Exists)
                Change priority of theTask holding the resource to highest priority;
                Delay, the Task is allowed to run and release the resource;
                Change priority of Task to its original priority;
        ENDIF
ENDWHILE
```

Task1 was temporary blocked when it attempted to gain access to the resource because it waited long enough for Task3 to lock it. Task2 had blocked Task3 because it did not require the resource but then Task3 was temporary raised to a higher priority. This was done by the Monitor task that periodically checked the resource to see if the task holding the resource had a lower priority than any other task waiting for it. Once the Monitor task realized that Task3 blocked Task1, it triggered the priority inheritance algorithm to alleviate the problem.

Chapter 6 Summary and Conclusion

Performance and validity in real-time systems depend upon the timeliness and logical operation of the application. Real-time operating systems aim to support such applications and make the process of working with real-time systems easier for the developer. It does this by removing many of the system details that engineers often deal with and allows them to focus more on the applications.

The RTOS must be written very efficiently, making the most out of the available hardware. It must give the developer control of the system via kernel services. The kernel also handles all of the timing services and interrupts. This keeps the application at a higher level, breaking the application into more logical tasks that are simpler to test and debug. Keeping tasks modular can help to expand or scale the system and most of RTOSs remain modular to adhere to systems' specific needs.

These abstract, priority-based tasks can communicate with each other and gain access to system resource. All the work that an application would normally do without an RTOS is still accomplished but can be done via kernel services. This allows the application to be more portable and easier to maintain and develop. Engineers should not consider writing their own RTOS as a real design option, they should only consider which RTOS best fits the system's specifications. Older systems with pre-existing customized operating systems may not be as easily ported but it may be worth the time and effort. More engineers use an RTOS and it has proven to be a useful and necessary tool in the development of real-time embedded systems.

Chapter 7 Future Work

This thesis provides a basis for real-time development with μC/OS as the RTOS. Future work with μC/OS may include adding a user interface, expanding its functionality with instructional modules, and analyzing its performance in greater depth. μC/OS may be used for other research or educational purposes including Dr. Rover's POLIS project with co-design, a capstone course in embedded systems or an operating system class for real-time application development.

BIBLIOGRAPHY

1. Labrosse, Jean J. *μC/OS The Real-Time Kernel*, R & D Publications,c1992, p.5

2. [Online] UC Berkeley, POLIS Project, http:// ptolemy.eecs.berkeley.edu/~pino/Ptolemy/thirdparty.html#ucberkeleyPOLIS

2. Ganssle, Jack G. "A Question of Balance", *Embedded Systems Programming*, Buyer's Guide 1996, p.15

3. Schwan, Karsten; Zhou, Hongyi. "Dynamic scheduling of hard real-time tasks and real-time threads. (Technical)" *IEEE Transactions on Software Engineering*, August 1992, v18, n8, p.736(13)

4. Margolin, Bob. "Smarter stuff. (embedded processors increasingly accessible and controllable over networks) (includes related article on Internet appliances) (Technology Information)" *Byte* June 1997, v22, n6, p.85(5)

5. Klann, David; Schultheis, Steven; Smith, David. "Match real-time OS to boards for smooth porting; an efficient port to new single-board computers hinges on harmonizing the workings of a real-time OS and target CPU. (real-time operating system/single-board computer matching) (Tutorial)" *Electronic Design*, Feb 7 1994, v42, n3, p.75(9)

6. Petreley, Nicholas. "Reasons you may be using DOS again soon and actually (gasp!) liking it. (Caldera to offer turneky Linux/NetWare server, leverage acquisition of DR DOS) (Product Information)" *InfoWorld*, Jan 19 1998, v20, n3, p.100(1)

7. Rajpal, Gurjot S. "Posix: conformance versus compliance. (includes relatedarticle on Posix definitions)" *Electronic Design*, Feb 21 1994, v42, n4, p.S21(3)

8. Barrett, Tom. "RTOS maximizes productivity in portable applications. (real-time operating system)(Engineering Software)" *Electronic Design*, May 28 1996, v44, n11, p.129(2)

9. Grehan, Rick. "8-bit microcontrollers grow up ... and down", *Computer Design*, May 1997, p.73

10. Grehan, Rick. "8-bit microcontrollers grow up ... and down", *Computer Design*, May 1997, p.75

11. "SGS sees life in 8-bit micro market. (SGS-Thomson Microelectronics' ST9+ microcontroller platform)(ProductAnnouncement)" *Electronic News* (1991) Nov 3 1997, v43, n2192, p.8(2)

12. Santoni, Andy. "Motorola expands offerings in microprocessor cores. (M.Core CPU with memory and I/O)(Product Announcement)" InfoWorld Sept 22 1997, v19, n38, p.29(2)

13. Motorola RTEK product evaluation literature, on-line help.

14. Bassak, Gil. Embedded RTOSs keep pace with processing power. (real-time operating systems)(includes directory of firms offering embedded operating systems) *Electronic Design*, Oct 14 1996,4,n21,p.80(5)

15. Bassak,Gil. "Embedded RTOSs keep pace with processing power (real-time operating systems)"", *Electronic Design*, Oct 14 1996, v44, n21, p.85

16. [Online] MIT (Massachusetts Institue of Technology). The Handy Board, http://lcs.www.media.mit.edu/groups/el/Projects/handy-board/faq/index.html

17. Stallings, William. *Operating Systems*, New York : Macmillan ; Toronto : Maxwell Macmillan Canada ; New York : Maxwell Macmillan International, c1992, p.113

18. Whitesmiths Ltd., *CXDB Source Level Debugger; User's Manual*, 1993,p.3-6

19. Pollack, Matthew. "Train control: Automating the world's railways for safety", *IEEE Potentials*, Feb/Mar 1998, p.8

20. Labrosse, Jean J. *µC/OS, The Real-Time Kernel*, R & D Publications,c1992, p.35

MICHIGAN STATE UNIV. LIBRARIES
31293017749551