

An Operating System Architecture for Embedded Systems —
Design and Implementation

Cheng, Chen-Mou

Department of Electrical Engineering

National Taiwan University

email: `doug@star.ee.ntu.edu.tw`

June 1998

Abstract

A real-time operating system architecture for embedded applications, namely, the OS/T 3.0 architecture, is proposed, and an implementation on the IBM PowerPC 403GA platform, called the OS/T v3.0-PPC, is explored in depth in this thesis. The architecture addresses the issues of splitting control of physical hardware devices from that of logical flow of program execution, as well as a compact model of efficient and flexible interrupt handling. Furthermore, implementations of the OS/T 3.0 architecture on different platforms should be source-level compatible in the sense that the same application program can run on each of the implementations after limited modification of hardware-related code and re-compilation. One of the implementation, the OS/T v3.0-PPC features a fully-preemptive multitasking real-time operating system, with support for periodic and sporadic tasks, as well as deterministic and predictable behaviors. Performance measurement of OS/T v3.0-PPC has been carried out, and the results show the effectiveness of the design.

Contents

1	Introduction	1
1.1	Overview and Design Goals	1
1.2	Related Works	3
1.3	Outlines of the Thesis	4
2	The OS/T 3.0 Architecture	5
2.1	The Process Model	5
2.2	Inter-Process Communication	6
2.2.1	The Event Model	6
2.2.2	The Message Queue Model	6
2.2.3	The Semaphore Model	7
2.3	The Time Model and Timer Facilities	8
2.3.1	Calendar Time and Date	8
2.3.2	Alarm Timer Facilities	8
2.3.3	Comparisons between Timer Facilities	9
2.4	The Exception Model	9
2.5	Process State Transitions	10
3	The OS/T v3.0-PPC Implementation	15
3.1	Overview	15
3.2	Naming Convention	15
3.3	Queue Operations	16
3.4	Memory Management	17
3.5	Process Management and Inter-Process Communication	18
3.6	Interrupt Handling and Timer Management	18
3.7	Debugging Facilities and Miscellaneous	19
3.8	Error Handling	20
3.9	Bootstrapping and System Initialization	20
3.10	Performance Measurement	21

4	Conclusions	26
A	Reference Manual	27
A.1	Time Management	27
A.2	Process Management	29
A.3	Inter-Process Communication	31
A.4	Exception Handling	34
B	STBLD	35

List of Figures

2.1	An Example Periodic Task	8
2.2	Another Example Periodic Task	9
2.3	An Example Exception Handler	11
2.4	An Example Multiple-IRQ Device Driver	12
2.5	The Process State Transition Graph	13
3.1	System Call Timing Under Light Load ($N = 10$)	22
3.2	System Call Timing Under Medium Load ($N = 100$)	23
3.3	System Call Timing Under Heavy Load ($N = 500$)	24
3.4	System Call Timing Under Different Load Conditions	25

Chapter 1

Introduction

1.1 Overview and Design Goals

The operating system is defined as a layer of software that multiplexes as well as abstracts physical resources, such as volatile memory, processor cycles, I/O devices, etc, for the application programmers. Application software, in contrast, refers to everything else that is designed for a special purpose, to achieve a specific goal, or to solve a particular problem. From the programmer's point of view, in addition to providing a set of high-level operations that conceals the details of the bare machine, an operating system enables programming based upon an extended, elegant, and platform-independent model that is probably not embedded in any hardware design.

An embedded system usually involves a microcontroller with limited computation power and simplified architecture. Despite the fact that programming model of a microcontroller is simple as compared with that of a general purpose processor, developing an operating system for embedded applications is not an easy task because requirements in such applications are usually very strict, especially in timing constraints. A real-time system is a system in which the correctness is determined not only by the result of computation but also by the time at which the result is computed. Missing timing constraints will, in a so-called *hard* real-time system, cause the result totally useless, thus regarded as a system failure even it is correctly computed. In contrast, in a *soft* real-time system, missing timing constraints occasionally is not fatal, although it might indicate a flaw in the design.

Microcontroller-based embedded systems are widely used in industry. One of the challenges in developing software for such applications is how to fastly adapt existing application software onto a new and more cost-effective hardware platform. As cheaper, faster, and more powerful embedded hardware develops, such ability can effectively shorten the time-to-market of new and competitively priced products. But application programs on embedded systems are mostly involved with direct manipulation of hardware for efficiency considerations, making porting across different hardware architecture nearly infeasible. The strategy is to separate control of physical hardware devices from flow of program execution; the operating system provides primitives to assist application programmers by controlling

the flow of program execution, leaving these programmers concentrating on writing code for hardware dependent and specific control. The goal of operating system architecture is to define common and hardware-independent abstraction of flow of program execution, making programming on different platforms similar and porting across these platforms easy.

The term ‘architecture’ means structure — the way in which something is made, built, or organized[1]. In the context of operating systems design, it refers to the assumptions on system behavior and the model based upon which a programmer develops application software. In short, the architecture of an operating system primarily determines how a programmer programs on top of it. An implementation, in contrast, is a realization of the architecture; it refers to a set of software that acts as an operating system, runs on a particular platform, and conforms to the architecture. A platform is characterized both by the hardware as well as the development software, including preprocessor, compiler, assembler, linker, etc. A set of implementations of an operating system architecture, possibly on different platforms, are said to be source-level compatible if the same application program can run without modification on each of the implementations after recompilation.

The goal is to design an operating system architecture — namely, the OS/T 3.0 architecture¹, that is primarily aimed for embedded real-time applications. OS/T 1.0 is a fast and compact real-time kernel, running on the ITRI CMS Development Card Model CMSS9306A, an MC68000-based single board system[2]. In addition to porting onto Intel real-mode 80x86-based systems, OS/T 2.x introduces the concept of interrupt handling tasks and the portability of the operating system itself[3]. Both OS/T 1.0 and OS/T 2.x feature fully-preemptive multitasking operating systems, and provide inter-process communication facilities. OS/T 2.x also addresses the issues of source-level compatibility. Based upon the experience, OS/T 3.0 formalizes our strategy in solving the portability problems of both the application software and the operating system, and provides a new model of exception handling.

Since the main purposes of applications on an embedded system are mostly to control a wide variety of hardware, they are usually deeply hardware dependent, making complete source-level compatibility a very difficult but less meaningful criterion for the operating systems to achieve. The definition of the source-level compatibility for embedded operating systems and applications is modified as follows: A set of operating systems are said to be source-level compatible if the same application program can run after modification of a limited portion of hardware dependent code and recompilation on these operating systems. In this context, the goal of the OS/T 3.0 architecture is a set of well-defined, powerful application programming interface and elegant, compact programming model, so that the implementations can easily achieve the following goals:

1. Any implementation of the architecture should have deterministic and predictable behavior;
2. Any implementation of the architecture should be fully preemptive multitasking;
3. Any implementation of the architecture should support periodic as well as sporadic tasks;

¹Or, whenever there is no danger of confusion, OS/T 3.0 for short.

4. All implementations of the architecture should be source-level compatible to one another.

1.2 Related Works

There are many commercial and academic real-time operating systems, ranging from small real-time kernels with the size of several kilo-bytes to huge packages with full support for file systems, networking systems, multiple processor, graphics, etc.

The μ C/OS from R&D Books Miller Freeman, Inc. is “a portable, ROMable, preemptive, real-time, multitasking kernel for microprocessors.” It is a small and portable system, primarily targeted at embedded controllers ranging from Motorola 6800/01/02/03, Intel 8080/85, Hitachi H8/3xx, to Zilog Z-80, just to name a few. The μ C/OS uses a simple interrupt-disabling mechanism when the kernel is updating global data, or in other words, is in the critical sections. By calculating the most lengthy code in which interrupts are disabled, the interrupt latency can be deterministically decided, which is about 700 clock cycles on Intel 80186/80188 processors[4].

The LynxOS from Lynx Real-Time Systems is a textscunix-compatible and POSIX-conforming, fully-preemptive and compact hard real-time operating system, running on most of the processors commonly seen in desktop personal computers and workstations, including Intel 80x86, Motorola 680x0 and PowerPC, Sun microSPARC/microSPARC II, and HP PA-RISC.

The OS/Open from IBM Microelectronics Corp. is specifically designed and optimized for the PowerPC processors. It is a small kernel and has hard real-time support. The typical thread switching time is about 200 cycles, which is comparable but a little bit more than that of the OS/T v3.0-PPC, indicating a slightly more complicated process model might be employed by the OS/Open than the OS/T v3.0-PPC.

The pSOS and pSOSystem from Integrated Systems, Inc. is a complete and scalable real-time operating system[5, 6]. The configurability and flexibility is achieved by highly modularized design; there are separate modules that support for single and multiple processor, file management and TCP/IP, streams communication, graphics, and JAVA.

The Real-Time Linux from New Mexico Tech. is a hard real-time extension to the Linux operating system. The idea is to use software emulated interrupts to make a real-time kernel coexist with a general purpose operating system, Linux in this case. In such an arrangement the kernel can exploit a wide variety of system service from Linux while guarantees timing requirements of the real-time tasks under its control. The results of interrupt latency measurement indicate that a great improvement over the Linux operating system is achieved; the interrupt latency of Real-Time Linux is about one third that of the Linux operating system on 33MHz Intel 80486 machines, while one eighth on 120MHz Intel Pentium machines[7].

The VxWorks from Wind River Systems, Inc. is a proprietary real-time operating and supports a wide range of industry standards including POSIX 1003.1b Real-Time Extensions, ANSI C and TCP/IP networking. A run-time C interpreter can be used for debugging purposes, making debugging

and experimentation easier and more efficient.

1.3 Outlines of the Thesis

In chapter 2 the architecture of OS/T 3.0 is described. More specifically, the programming model of process and exception handling is described in detail, with the application programming interface listed in the appendix. The application programming interface plays a central role in characterizing a set of source-level compatible operating systems; it is the collection of symbolic constants, data types, prototypes of inline functions and functions, and variables that are exported by the operating system for the purpose to provide service to application programs. In chapter 3 an implementation of the OS/T 3.0 architecture on the IBM PowerPC 403GA platform — namely, the OS/T v3.0-PPC is described. The algorithms adopted, as well as the design philosophy behind, are explored in depth. The results of performance measurement are listed and discussed. Finally conclusions are stated, summarizing the experience and insights gained in the development of the system.

Chapter 2

The OS/T 3.0 Architecture

2.1 The Process Model

OS/T 3.0 is a process-based¹ operating system, designed for real-time applications on embedded systems. A process is simply a program in execution[8, 9, 10, 11], and the system is merely a collection of concurrently and independently running processes.

A process can be fully characterized by its context. The context of a process is the collection of the content of the *private* registers, which are not shared among processes. Each process has its own copy of the contents of these registers and thus makes it independent from all one processes. The context is like a frozen image, or a snapshot, of a process; with the concept of context, the operating system can execute programs at any time in any order, interlacingly or as a batch.

The state of the system is the collection of the states of all processes, whether they are ready for execution, suspended from running, or blocked in waiting certain event to occur. There are a large number of combinations of all processes states; among these only those in a small, proper subset are said to be valid. A state transition takes the system from one valid state to another. Such a state transition is atomic, meaning that it occurs in an inseparable fashion, either completing the entire transition or not taking place at all. There is no intermediate states observable from outside of the system.

How the active processes, those which are eligible for the processor to execute, in the system obtain their processor time is called the scheduling policy of an operating system. In another perspective, the scheduling policy is a mapping from the system state to one process, which will become the *currently running process* after the policy has been enforced by the scheduler. The scheduling policy of OS/T 3.0 is fully-preemptive, meaning that at any given moment, if an active process is running, it must be with the highest *priority* among all active processes. For example, if a process with higher priority than the currently running process becomes active, it will preempt the latter and starts executing immediately. If more than one process have the same highest priority, they will run one after another in a round-robin

¹In this document, a process is sometimes called a task; these two terms are interchangeable.

fashion.

2.2 Inter-Process Communication

There are three types of inter-process communication facilities in OS/T 3.0, namely, events, message queues, and semaphores. The processes can utilize these facilities not only to communicate but to synchronize with one another. Furthermore, mutual exclusion and critical sections can also be easily constructed using the inter-process communication primitives provided by the architecture.

2.2.1 The Event Model

An event is a typed message sent synchronously from one process to another. Each process has a record of the events that have been sent to it. There are 32 different types of event in OS/T 3.0, which is encoded into a 32-bit unsigned long integer, with each bit representing the occurrence of one type of event. The application programmer can decide what information a particular type of event carries by assigning meanings to each of the event types.

A process receives events via the `ev_recv` operation. The process has to specify which kinds of event to receive, whether it will wait for the occurrence of the events, and if it wants to wait, how long it is willing to wait. If a process issues an `ev_recv` operation but there is no event of the specified types pending, it will be either informed of such a situation, blocked until the specified amount of time has elapsed, or even blocked forever, according to the specified style in the system call invocation. If any subset of the expected events occurs during that period of time, the process will be unblocked immediately and the waiting event record of that process will be cleared, showing that the process no longer waits for any event. Events other than the process is expecting will be recorded as pending events, so a subsequent `ev_recv` can correctly receive these events.

A process sends events via the `ev_send` operation. The process specifies to which process the events are sent, as well as the kinds of event being sent. If the target process has not issued an `ev_recv`, the events are recorded as pending events to the process. Any successive event of the same type will be *lost* before the target process receives such type of event. In other words, sending the same type of event to a process does not have accumulative effect; a process only knows whether a particular type of event has been sent to it, but does not know how many times it has been sent, nor how many processes have sent that type of event to it.

2.2.2 The Message Queue Model

In most operating systems processes can send messages to one another, just like the mailing system in the real world: A process puts the message into an envelope on which the address of the recipient is written, and the operating system delivers it to the mailbox of the recipient process. Under this

metaphor, groups of cooperative processes can synchronize and exchange information using such a message sending mechanism.

In OS/T 3.0, however, the metaphor is a little bit different by introducing the concept of a message queue. A message queue is like a mailbox, but this kind of mailbox is not *owned* exclusively by any process. Rather, any process which *knows* the message queue can send and receive messages from that queue. A process can still create its own message queue from which only that process can receive messages, or, alternatively, a group of cooperative processes can share a message queue to communicate with one another or with other processes outside the group. The message queue, therefore, is a more generalized model of message passing.

A process receives messages from a message queue via the `q_recv` operation. Like issuing an `ev_recv` operation, the process has to specify whether it will wait for the message to arrive and if it wants to wait, how long it is willing to wait. If two or more processes have issued `q_recv` operations over the same message queue and there is no message in that queue, they will pile up in the waiting queue of the message queue. A message arrived later on will be delivered to the piled processes in a first-in-first-out fashion, just as the name ‘waiting queue’ indicates.

There are three ways to send a message: `q_send`, `q_urgent`, and `q_broadcast`. Normally when a message arrives at a message queue where there is no process waiting, the message will be queued in a first-in-first-out fashion, just as the name ‘message queue’ indicates. However, if a message is sent via the `q_urgent` operation, it will be put at the head, rather than the end, of the message queue. A successive `q_urgent` will be regarded as having an even greater urgency that the message it delivers will be put in front of all previously arrived messages. A `q_broadcast` operation will broadcast identical messages to each process at the waiting queue of a message queue, but if there is no process waiting, the broadcast message will be lost, received by no process at all.

2.2.3 The Semaphore Model

A semaphore is an instance of an object with a nonnegative integer upon which two operations, P (wait) and V (signal), are defined[9]. A V operation increments the semaphore value whereas a P operation decrements the semaphore value if it is nonzero. Together these two operations can be used to achieve mutual exclusion.

A process acquires a semaphore via the `sem_p` operation. Like issuing an `ev_recv` or a `q_recv` operation, the process has to specify whether it will wait for the semaphore to have a nonzero value and if it wants to wait, how long it is willing to wait. If two or more processes have issued `sem_p` operations over the same semaphore and that semaphore currently has a value of zero, they will pile up in the waiting queue of the semaphore, just as if they were receiving messages in a same empty message queue. If the semaphore becomes nonzero later on, the first process that issued the `sem_p` operation acquires the semaphore.

A process releases a semaphore via the `sem_v` operation. Each semaphore also has an upper bound;

if a `sem_p` operation is about to increment the semaphore value beyond the upper bound, an error is reported to the issuing process.

2.3 The Time Model and Timer Facilities

The time model in OS/T 3.0 introduces the concept of a *tick*. A tick is the resolution, the smallest meaningful interval, of time. It is usually a few hundredths of a second, and may vary from platform to platform. Furthermore, a different configuration may redefine the tick to a different value so long as the system can operate correctly. Theoretically there is a lower bound how small a tick can be; however, it is difficult to find out the exact value.

2.3.1 Calendar Time and Date

All implementations of the architecture should maintain a real-world clock that tells calendar time and date as the UNIX operating system does. The `get_time`, `tm_gettime`, `set_time`, `tm_settime` operations, as their names indicate, tell and set the calendar time and date.

2.3.2 Alarm Timer Facilities

In addition to maintaining calendar time and date, all implementations of the architecture should provide alarm timers that are capable of waking up or sending events to a process after a designated interval or at an appointed time. The `tm_wkafter`, `tm_wkwhen` and `tm_wktime` operations wake up a process while the `tm_evafter`, `tm_evwhen` and `tm_evtime` send events.

The `tm_evperiodic` sends events periodically to a process. Such an operation can be used to support periodic task, as is illustrated in the C program in fig.2.1.

```
void periodic_task_ev(void)
{
    ULONG    tmid;

    tm_evperiodic(SOME_EVENT, MY_PERIOD, &tmid);

    for ( ; ; ) {
        ev_rcv(SOME_EVENT, I_WILL_WAIT, FOREVER);

        do_something();
    }
}
```

Figure 2.1: An Example Periodic Task

2.3.3 Comparisons between Timer Facilities

There are two distinct approaches a process can set up an alarm timer. One approach is to issue a `tm_wkx` operation in which the calling process is put to sleep immediately. A more flexible approach is to issue a `tm_evx` operation in which the calling process may elect to continue the execution. Also one single process may issue multiple `tm_evx` operations before it starts waiting on `ev_recv`. This makes the event sending timer a more powerful mechanism.

To further explain the distinction, let us examine the program in fig.2.2, which roughly support the same periodic task as shown in fig.2.1, if the computation in `do_something` is not too complicated.

```
void periodic_task_wk(void)
{
    ULONG    tmid;

    for ( ; ; ) {
        tm_wkafter(MY_PERIOD);

        do_something();
    }
}
```

Figure 2.2: Another Example Periodic Task

However, if the computation in `do_something` is so complex that it takes two or more ticks to finish, this approach may not work — at least the period has to be compensated, taking into account the computation time. The situation can be worse if the computation time can not be decided deterministically, which is usually the case in the real-world algorithms and computation. If the error of the period is required to be within, say, a few ticks, this approach does not work at all.

2.4 The Exception Model

An interrupt is the action in which the processor saves its current context and starts executing code in another predetermined location. An exception is the event that causes the processor to take an interrupt if properly enabled. An exception is described as asynchronous if it is external to the processor, while synchronous if it is internal, caused by the execution of instructions. Furthermore, if an exception is precise, the address saved when an interrupt takes place is either the address of the excepting instruction or that of the next sequential instruction. On the other hand, if an exception is imprecise, it is not required to save any addresses mentioned above, but something else may be saved. Imprecise exceptions may be unrecoverable, indicating a serious situation might have occurred.

The handling of a particular exception is platform dependent. In fact, it depends not only on the processor on which an operating system is running but on the configuration at that time. Different configuration of a same processor may have different requirements on handling the same exception.

In order to comply with such divergence, the OS/T 3.0 architecture does not enforce any exception

handling policy. Rather, the architecture suggests a programming style to handle exceptions, using the inter-process communication primitives. An exception is modeled as an event and an interrupt simply sends such an event to the corresponding process.

The architecture defines three operations regarding exception handling; they are: `isr_hook`, `isr_unhook`, and `isr_getinfo`. The `isr_hook` operation specifies an event sent to a process, after a usually small fragment of code is executed, when a particular type of exception occurs, while the `isr_unhook` operation cancels such a linkage. The `isr_getinfo` operation reports the current setting of a particular type of exception handling.

Fig.2.3 is an example showing how to create a task to handle a particular exception, or, in other words, to create an interrupt service routine. Another example in fig.2.4 shows how to write a device driver if that device uses two or more interrupt request lines, which is not rarely seen in a real-world device.

2.5 Process State Transitions

The process state transitions are depicted in fig.2.5, in which a rectangle, a circle or a diamond represents a state and an arrow represents a state transition. The system calls associated with an arrow indicates a state transition may take place if these system calls are invoked by the currently running process, which is the only process that is in the R_e state at any moment. However, this is not the sufficient condition for a state transition; for example, a `q_send` system call may wake up a process that has invoked a `q_recv` system call but hasn't received anything yet; on the contrary, such a `q_send` system call may wake up no process at all if there hasn't been any process waiting at the target message queue. A trailing asterisk in the system call name indicates that the system call is requested by the process itself. All arrows originate from the R_e state may only have such operations, and arrows with these operations only originates from that state since all system calls can only be requested by the process in the R_e state. System calls that are not associated with an arrows originating from some state may not cause any state transition from that state.

A process can have the following states:

1. R =Ready
2. R_e =Ready and Executing
3. B_e =Blocked at Events
4. B_m =Blocked at Messages
5. B_s =Blocked at Semaphores
6. S =Suspended
7. SB_e =Suspended and Blocked at Events

```

void root_task(void)
{
    ULONG    pid;

    do_some_initialization();

    t_create('ISRn', A_HIGH_ENOUGH_PRIORITY, ..., &pid);

    t_start(pid, ..., irq_n_task, ...);

    isr_hook(IRQ_N, irq_n_isr, pid, EVENT_N);

    do_other_initialization();
}

void irq_n_isr(void)
{
    do_low_level_interrupt_service();
}

void do_low_level_interrupt_service(void)
{
    clear_hardware_flags__etc();
}

void irq_n_task(void)
{
    for ( ; ; ) {
        ev_rcv(EVENT_N, I_WILL_WAIT, FOREVER);

        do_high_level_interrupt_service();
    }
}

void do_high_level_interrupt_service(void)
{
    sem_p(some_sem, I_WILL_WAIT, FOREVER);

    execute_code_mutual_exclusively();

    sem_v(some_sem);
}

```

Figure 2.3: An Example Exception Handler


```

void root_task(void)
{
    ULONG    pid;

    do_some_initialization();

    t_create('DEVX', A_HIGH_ENOUGH_PRIORITY, ..., &pid);

    t_start(pid, ..., dev_x_drv, ...);

    isr_hook(IRQ_X1, ..., pid, EVENT_X1);
    isr_hook(IRQ_X2, ..., pid, EVENT_X2);
    isr_hook(IRQ_X3, ..., pid, EVENT_X3);
    ...

    do_other_initialization();
}

void dev_x_drv(void)
{
    ULONG my_events = EVENT_X1|EVENT_X2|EVENT_X3;

    for ( ; ; ) {
        ev_rcv(my_events, I_WILL_WAIT, FOREVER);

        if ( is_irq_x1_coming() ) {
            do_something();
        }

        if ( is_irq_x2_coming() ) {
            do_another_thing();
        }

        if ( is_irq_x3_coming() ) {
            do_a_third_thing();
        }

        ...
    }
}

```

Figure 2.4: An Example Multiple-IRQ Device Driver

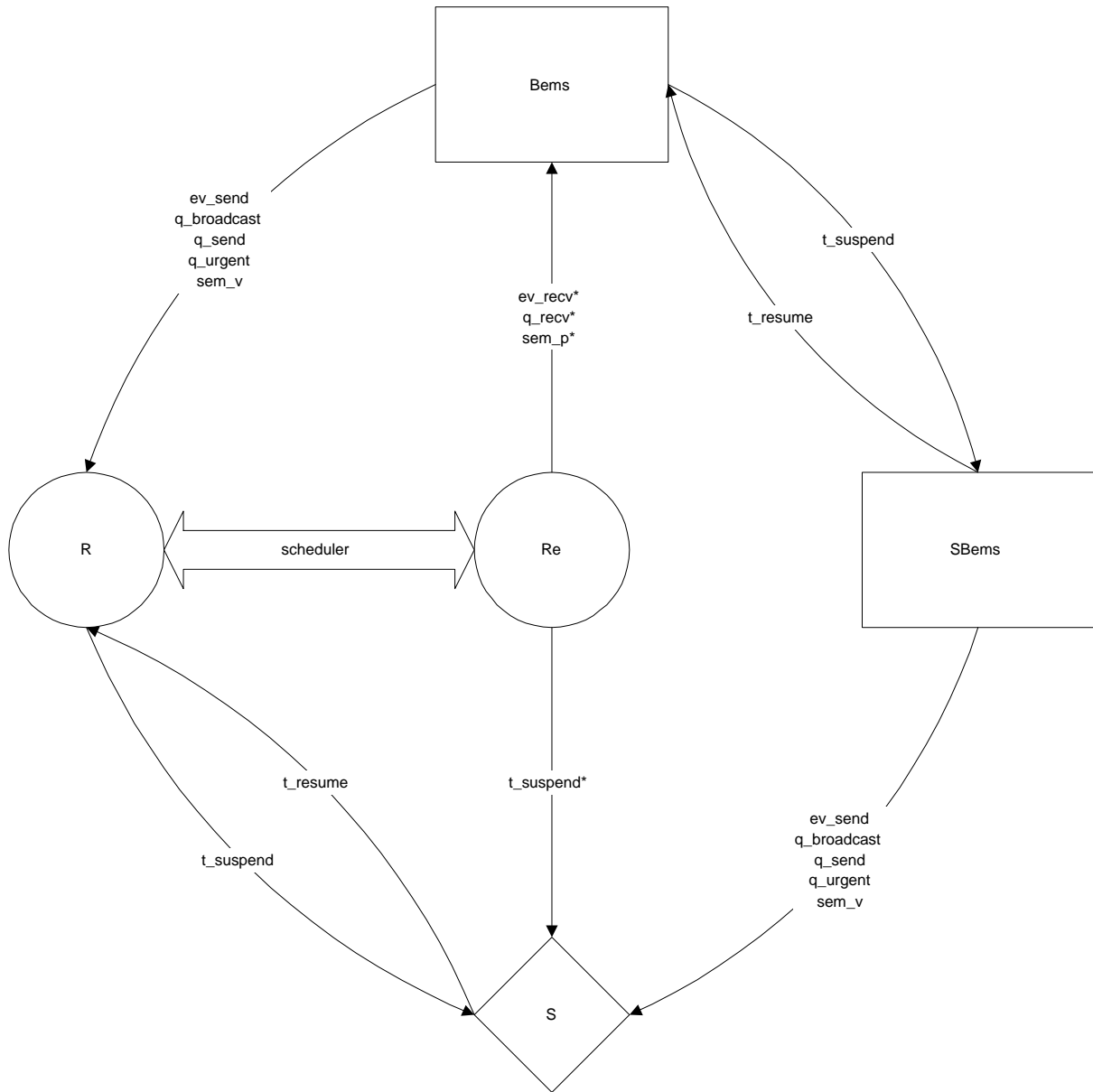


Figure 2.5: The Process State Transition Graph

8. SB_m =Suspended and Blocked at Messages

9. SB_s =Suspended and Blocked at Semaphores

B_{ems} in the graph is either B_e , B_m , or B_s , representing a state in which the process is blocked for some reason.

$R_e \rightarrow B_{ems}$

An executing process becomes blocked when:

1. It attempts to receive via `q_recv` a message from an empty message queue; or
2. It attempts to wait via `ev_recv` for an event condition that has not occurred; or
3. It attempts to acquire via `sem_p` a semaphore that is not currently available.

$B_{ems} \rightarrow R$

A blocked process P becomes ready when:

1. A message arrives at the message queue where P has been waiting and P is at the head of the waiting queue; or
2. An event for which P has been waiting is sent to P ; or
3. A semaphore for which P has been waiting is released and P is at the head of the waiting queue; or
4. P has been waiting with a timeout option for events, a message, or a semaphore and the specified timeout interval expires; or
5. P has been waiting at a message queue or a semaphore and that queue or semaphore is deleted by some other process.

Other state transitions are quite straight forward. For example, a ready process becomes suspended ($R \rightarrow S$) when another process invokes a `t_suspend` system call upon it.

Chapter 3

The OS/T v3.0-PPC Implementation

3.1 Overview

The IBM PowerPC 403GA is a 32-bit RISC embedded controller, fully compliant with specifications for 32-bit implementations of the PowerPC RISC User Instruction Set Architecture[12, 13, 14, 15], and featuring rich support for a variety of peripherals. The IBM PowerPC 403GA has four DMA channels, one serial port, separate instruction cache and write-back data cache, both two-way set-associative[16].

The OS/T v3.0-PPC is an implementation of the OS/T 3.0 architecture on the IBM PowerPC 403GA platform. In addition to complying with the architecture, the OS/T v3.0-PPC further makes easy the porting of the operating system itself by explicitly pointing out the underlying programming model, as well as carefully coding and documenting.

The implementation makes a few assumptions of the underlying platform and can be easily ported to a platform that has similar characteristics. To begin with, the OS/T v3.0-PPC assumes that:

1. The target platform has a 32-bit linear address space;
2. The target platform has two operation modes, one for user applications and the other for system operations;
3. The C compiler uses a pass-by-register calling convention.

3.2 Naming Convention

In a large-scale programming project, the naming pollution may become an undesirable phenomenon that should be avoided. When there are too many global names, it becomes very difficult to find a unique name for a new symbol, not to mention the difficulties to remember these names without confusion or misspelling. Enforcing a consistent naming convention is one of the remedies for the disease of naming

pollution. A naming convention is a systematic way of naming symbols. A symbol is the name of a constant, a typedef'ed data type, a function¹, or a variable. The logical region in which a symbol is known is called its scope. A global symbol is known to all modules, while a modular symbol should only be known to programs within the same module. OS/T v3.0-PPC is coded using a hybrid programming style, mixing code written in assembly language as well as in C language. A module may contains two separate source files written in these two different programming language. Furthermore, sometimes it is convenient to split a module into several small source files to enhance the readability and the maintainability of code. In either situations, the names of the source files in a module should indicate their belonging to that module, having a common prefix abbreviated from the module name, or located in a subdirectory named after the module.

In OS/T v3.0-PPC, all global symbol names begin with a letter 'g', while modular symbol names begin with the letter 'm'. A local symbol name may begin with an underscore character '_', or it may have nothing indicating its scope for simplicity. Immediately following the scope indicating letter, the official abbreviation of the module is appended, and then the name of the symbol describing its purpose and functionality follows after an underscore character '_'. For a symbolic constant, a name all in upper case is used; for other kinds of symbols, the name is in all lower case. The underscore character '_' works as the separating letter if the name contains two or more words. For a routine name, a verb is usually used at the beginning, followed by an object; altogether the '*verb-object*' form indicates the main operation of the routine. Finally, if the symbol is a typedef'ed data type, a suffix '_t' is appended at the end, while a '_i' is appended at the end of the name of an inline function. The length of the name, as a rule of thumb, is usually between 8–20 letters, including the prefix and the suffix[17].

3.3 Queue Operations

Queue² operations include primitives for three widely-used abstract data types in OS/T, namely, circular doubly-linked lists, priority queues, and tables. Many of the kernel data structures are derived from these three data types; in other words, these operations serve as *base classes* in object-oriented design nomenclature. Because the C programming language lacks the concept nor the mechanism of object inheritance, the invocation of these operations are a little bit awkward. Pointers to the target object have to be cast into pointers to these base data types before we pass them into associated functions.

The circular doubly-linked list is a suitable data type for most, if not all, of the kernel objects since it has constant insertion and deletion time complexity, as well as linear search time complexity, if the list does not has to be sorted. Keeping a sorted circular doubly-linked list, if required, has linear insertion time complexity. Because of its efficiency and moderate cost, the circular doubly-linked list

¹The terms functions, routines, subroutines, and procedures are used interchangeably in this document, referring to an orderly collection of statements that has an entry point and an optional return value.

²The concept introduced here is more of a list than of a queue; for historic reasons, however, the term queue is used regardlessly.

becomes the most widely used in OS/T v3.0-PPC.

The priority queue has 256 levels of priority; the number of levels is hard-wired into program code and therefore can not be changed. Furthermore, in order to accelerate search speed, three levels of masks are applied to indicate whether there is any item in a particular range of priorities[2]. Such a mechanism makes the time complexity of a priority queue $O(\sqrt[4]{N} + \sqrt{N} + \sqrt{N}) = O(\sqrt{N})$, in the expense of $\sqrt[4]{N} + \sqrt{N}$ extra memory space. Since the efficiency of the priority queue plays an important role in determining the performance of the scheduling, we feel that such a cost is much more than acceptable.

The table is another useful data structure. A table is, basically, a mapping from a nonnegative integer to a pointer that may point to an arbitrary type of object. The object, for example, can represent a process or a timer, and the nonnegative integers is the *identifier*, or simply the *id*, of that object. It has constant search and deletion time complexity, but linear insertion time in a rarely happening worst case is unavoidable.

All of the queue operations assume the existence of target objects. The operations do not concern themselves with any memory allocation nor release operations; the *derived classes*, if we put it in an object-oriented design nomenclature, should assume the responsibility for the allocation and release of the actual storage of an object.

All of the queue operations do not check the consistency of target objects for efficiency considerations. The deletion routine of the circular doubly-linked list doesn't check whether the object is really an item in such a list; it might cause unexpected errors if used carelessly. The operations on priority queues doesn't check whether the object to manipulate indeed has the priority specified either. Inconsistency may arise if incorrect priority information is passed into these routines. Insertion into a table may result in an infinite loop if inconsistency occurs inside the target table data structure. More specifically, if the removal of an entry in a full table does not fill that entry with a proper value as designed, a latter insertion into the table will loop infinitely since no empty entry can be found.

3.4 Memory Management

Memory management in OS/T v3.0-PPC provides primitives for memory allocation and release operations. It lies in the bottom of the functionality hierarchy, and is in charge of the management of the core³. The core is divide into blocks with variable sizes, each having a fixed length of header containing size and checksum information, as well as links to other blocks. Although the size of a memory block may vary, the block boundary is always word-aligned, or 4-byte-aligned. If the size, in the unit of bytes, of a memory block being allocated is not a multiple of 4, it is rounded up to a multiple of 4, and a memory block of that size is actually allocated without notifying the requester.

All memory blocks are linked together in three circular doubly-linked lists, one for free blocks, one for blocks allocated for heaps and the third for stacks. At any given instant, a memory block is in

³Also called the primary storage or the main memory in this document, following a tradition in the UNIX operating system.

exactly one of the three lists, indicating its current usage. In such arrangement, the operating system is capable of managing physically separate memory space, say, in a system with both SRAM and DRAM whose addresses are not contiguous.

The first-fit algorithm is applied when allocating new memory blocks[18]. When a memory block is to be released, it is inserted into the free memory block list according to the address of the starting byte in the block, and is merged, if possible, with its *neighbour blocks*⁴. Thus while allocating memory blocks may result in an increase in the total number of memory blocks, releasing blocks may, on the contrary, decrease the total number of blocks.

3.5 Process Management and Inter-Process Communication

At any given instant of time, there must exist one and only one process that is running and occupying the processor time, whose task control block, or TCB for short, is recorded in a global variable. It is the only process that can have a out-of-date context recorded in the task control block since it is running.

A process can run until an interrupt takes place; then its context is saved and the stack is switched to the system stack on which support routines do the computation for a possible system state transition. The kernel is merely the collection of such routines, running in the system stacks.

The atomicity of system state transitions is implemented using the simple mechanism of disabling asynchronous exceptions from interrupting the processor when a transition takes place. The disabling of interrupts is a primary source of interrupt latency, and thus is put into practice sparingly.

The possible states of a process are encoded into an unsigned long word, called the process state word, with each state being represented by setting some or all bits in the process state word to one — except for the ready state. The ready state of a process is actually represented by clearing all bits in the process state word to zero. The ready state can, in some sense, be regarded as a ‘ground state’ while all other states as ‘excited states.’ There are routines designed to activate and deactivate ‘states’ of a process by turning on and off, respectively, the corresponding bits in the process state word.

Inter-process communication in OS/T v3.0-PPC is designed to fully comply with the specification in OS/T 3.0 architecture; see chapter 2.2.

3.6 Interrupt Handling and Timer Management

After an interrupt takes place, the entire context of currently running process should be saved immediately. A brief prologue follows, which usually includes code that handles hardware related events and wakes certain process if the interrupt processing should take a long time. Finally the scheduler is called, making sure that the highest priority process will run next.

⁴Two neighbour blocks occupy consecutive bytes in the core where there is no gap between the two blocks; in other words, the beginning of one block is immediately after the end of the other block.

In IBM PowerPC 403GA, the current context of the processor is saved in two special purpose registers, one for machine status register, the MSR and the other for program counter, the PC[16]. There are two kinds of interrupts, namely, critical and non-critical. For a non-critical interrupt, the processor saves PC in SRR0 (save/restore register 0) and MSR in SRR1, while for a critical interrupt, the PC is saved in SRR2 and MSR in SRR3. Then the processor looks up in the vector table, pointed by a special register, the exception vector prefix register, or EVPR for short. Next the address of the corresponding interrupt service routine is calculated and branched to, starting the execution of the interrupt service routine. The MSR is modified to disable other exceptions which save the context in the same place to prevent nested interrupts from corrupting the context. From here the interrupt handling routines takes over, saving the entire context of currently running process. The stack has been switched to the system stack in the context save routine, so kernel routines that do state-transition computations can be performed immediately. Finally the scheduler is called, finishing an interrupt processing.

The exception handling lies at the heart of the design of a multiprogramming time-sharing environment since the real-time clock interrupts the processor periodically to generate system ‘ticks,’ based upon which the kernel program has to properly switch back and forth from one process to another. Besides real-time clock interrupts, interrupt handling can constitute a significant portion of the main task of the kernel program in an embedded system. An efficient and robust interrupt handling mechanism, therefore, is the key to a successful real-time operating system.

OS/T v3.0-PPC uses a flexible timer abstraction, which enables taking an action periodically or at a specified moment in the future. Each tick a clock interrupts the operating system, and a system routine named `gclk_tick` is invoked to check whether there is any timer expiration, and if so, the time-out action routine of that timer is invoked.

3.7 Debugging Facilities and Miscellaneous

Debugging capability of an embedded operation system is difficult to design since such a system usually has vulnerable output channels. When a serious bug is run into, the output channel can hardly survive, telling the programmer what was going on before the big crash; the system just hangs there without leaving a clue. Hardware debuggers are of much help; they usually have the ability not only to peek at the content of the processor registers and memory locations, but to do single-step execution.

One of the primary debugging facilities in OS/T v3.0-PPC is a system log daemon. When the system log routine is invoked, it logs the value of the two time base register, `tbhi` and `tblo`, along with the process identifier (pid) of the currently running process and the caller that requests such a logging action. Latter on the log can be dumped via the `dsys_dump_log` routine. An off-line log analyzer can use the log to analyze the timing of the system routines, as well as the profile and the statistics of each active process.

There is a example debugger that is implemented as a privileged process. In addition to dump

system log, the debugger can be used to peek at most of the kernel data structures, as well as memory locations. Timing of system calls can also be measured by issuing a `sc` command in the debugger.

OS/T 3.0-PPC is designed to gracefully halt the execution of the system when a serious error has been detected. When the system encounters internal errors that are not recoverable, the `panic` routine will be invoked to inform the system programmer of such a situation. Depending on whether a RISC WATCH debugger is attached, `panic` will dump appropriate debugging information before halting.

The `gppc_m[ft]reg`, `gppc_l[whb]z` and `gppc_st[whb]` functions are the C interface for accessing registers as well as main memory. They are restricted for privileged usage.

There are I/O routines for the built-in serial port of IBM PowerPC 403GA; these routines are primarily designed for kernel to send and receive control information. User applications should develop their own serial port drivers; there is an example driver in `spdrv.c`.

3.8 Error Handling

The error handling concept in OS/T v3.0-PPC resembles that in the C++[19] and Java programming language, “try, catch, and through exceptions⁵.” Because the C programming language lacks such a feature, the error handling module is designed, naturally, for remediation. A callee function merely reports success or failure; when an error has occurred, say, in function `foo`, the `gerr_throw` routine is called to throw an exception by recording the detailed reason in a global variable, `gerr_errno`, and push `foo` unto the call stack. Receiving the failure report of `foo`, the caller examines the content of `gerr_errno` and decides whether it is capable of handling such an error; if not, it just reports failure to its own caller, further propagating the error up in the call stack without doing actual error handling. In such a mechanism an error can be propagated through the call stack until control reaches a function that knows how to handle the error. If no function at all actually knows how to handle an error, which is usually the case when a fatal error or inconsistency in system data structures has occurred, the `panic` routine is invoked, gracefully halting the execution of the system.

3.9 Bootstrapping and System Initialization

Since OS/T v3.0-PPC is designed for embedded applications, the bootstrapping is quite different from operating systems for general purpose operating systems on desktop computers. The entry point of application programs has to be a function named ‘`root_task`,’ which will be created as a process when OS/T v3.0-PPC bootstraps.

Application programs are compiled and linked with the kernel. Through the help of IBM OpenBIOS ROM Monitor, the binary image of OS/T v3.0-PPC and application programs is downloaded via the BOOTP protocol, which uses the TFTP (Trivial File Transfer Protocol) upon UDP to communicate and is common in bootstrapping diskless workstations.

⁵Actually, the nomenclature is borrowed from there.

There is a 64-Kbytes symbol table in OS/T v3.0-PPC, which is constructed by the STBLD⁶ program using the information in the ELF (Executable and Linkable Format) file[20] generated by the High C compiler and linker[21, 22, 23]. The STBLD program reads and parses the map file and writes what is needed by OS/T v3.0-PPC into the ELF file. In addition, the STBLD program calculates the beginning and the ending of the OS/T v3.0-PPC image in the memory, according to which the operating system can bypass the occupied memory, collect unused memory blocks, and link these blocks in the free memory block list as described in 3.4 when it initializes the main memory. The STBLD program is appended in appendix B.

3.10 Performance Measurement

Fig. 3.1, 3.2, 3.3, and 3.4 summarize the results of performance measurement of the OS/T v3.0-PPC; the execution time of system calls are measured under different load conditions. The measurement is conducted with N sets of randomly created background load; in each set there is an active process doing trivial but CPU-intensive computation, accompanied by an alarm timer, a message queue, and a semaphore. A lightly-loaded system is simulated with $N = 10$, while a medium-loaded with $N = 100$. An extremely heavily-loaded situation is when $N = 500$, which is very unlikely to be observed in a real-world embedded system where so many processes are competing for such a large amount of resources.

For many system calls analyses in algorithms indicate that the time complexity should be linear in the worst case, while the results of performance measurement show only a few percents of increase in the execution time of these system calls. As load increases, however, the standard deviation of system call timing measurement increases accordingly. This is not unexpected since the constant in time complexity may exceed the coefficient of the linear term, i.e. $O(C_0 + C_1N)$ where $C_0 \gg C_1$. Also, the figures show that under medium load, some of system calls execute even faster in average than they do under lightly-loaded situations. This can be explained by the effect caused by cache misses since a cache miss can introduce a fairly large performance penalty. The reason why `t_delete` system call requires such a long time to finish is due to the fact that when deleting a process, all of its resources have to be reclaimed, involving a large amount search that requires time proportional to the maximum number of objects allowed to create in a particular type of system configuration; thus the execution time of deleting a process is independent of the load condition and has relatively small variance.

⁶Acronym for Symbol Table Builder.

system call name	minimum	mean	std dev.	maximum
gtsk_save_context_i	2.43 μ s	2.46 μ s	0.24 μ s	9.75 μ s
null_sc	23.55 μ s	24.08 μ s	0.41 μ s	27.18 μ s
tm_gettime	20.97 μ s	21.67 μ s	0.30 μ s	22.41 μ s
get_time	85.47 μ s	86.19 μ s	0.52 μ s	87.72 μ s
tm_settime	17.94 μ s	18.77 μ s	0.34 μ s	19.68 μ s
set_time	86.82 μ s	89.60 μ s	0.53 μ s	91.11 μ s
tm_evafter	61.23 μ s	62.43 μ s	0.58 μ s	69.33 μ s
tm_evperiodic	50.13 μ s	51.72 μ s	0.84 μ s	59.19 μ s
tm_delete	55.02 μ s	56.90 μ s	0.84 μ s	59.49 μ s
tm_getload	28.50 μ s	31.46 μ s	0.58 μ s	33.42 μ s
t_create	69.72 μ s	71.41 μ s	0.93 μ s	83.01 μ s
t_start	44.19 μ s	45.46 μ s	0.65 μ s	47.55 μ s
t_suspend	37.50 μ s	39.89 μ s	0.82 μ s	42.90 μ s
t_resume	38.40 μ s	40.21 μ s	0.69 μ s	41.88 μ s
t_delete	1601.82 μ s	1603.26 μ s	0.83 μ s	1606.92 μ s
t_getid	20.85 μ s	22.12 μ s	0.48 μ s	23.67 μ s
t_comptime	30.57 μ s	32.24 μ s	0.61 μ s	33.75 μ s
ev_send	27.78 μ s	28.29 μ s	0.58 μ s	30.33 μ s
ev_recv	23.97 μ s	24.42 μ s	0.35 μ s	25.53 μ s
q_create	50.85 μ s	53.35 μ s	0.97 μ s	59.67 μ s
q_send	52.62 μ s	55.01 μ s	0.79 μ s	57.84 μ s
q_broadcast	37.44 μ s	39.86 μ s	0.80 μ s	42.51 μ s
q_urgent	56.13 μ s	57.71 μ s	0.61 μ s	59.79 μ s
q_recv	49.35 μ s	50.74 μ s	0.76 μ s	53.55 μ s
q_delete	55.44 μ s	57.30 μ s	0.69 μ s	59.10 μ s
sem_create	47.49 μ s	48.81 μ s	0.75 μ s	57.36 μ s
sem_p	32.85 μ s	33.82 μ s	0.49 μ s	35.25 μ s
sem_v	32.10 μ s	33.83 μ s	0.73 μ s	36.33 μ s
sem_delete	47.01 μ s	48.83 μ s	0.67 μ s	51.24 μ s

Figure 3.1: System Call Timing Under Light Load ($N = 10$)

system call name	minimum	mean	std dev.	maximum
gtsk_save_context_i	2.43 μ s	2.46 μ s	0.24 μ s	9.75 μ s
null_sc	23.52 μ s	24.02 μ s	0.42 μ s	27.27 μ s
tm_gettime	20.79 μ s	21.67 μ s	0.29 μ s	22.32 μ s
get_time	84.96 μ s	86.20 μ s	0.52 μ s	87.75 μ s
tm_settime	17.94 μ s	18.77 μ s	0.34 μ s	19.62 μ s
set_time	87.42 μ s	89.60 μ s	0.53 μ s	91.14 μ s
tm_evafter	58.47 μ s	59.64 μ s	2.19 μ s	126.27 μ s
tm_evperiodic	48.48 μ s	49.91 μ s	2.27 μ s	118.32 μ s
tm_delete	51.69 μ s	53.21 μ s	0.78 μ s	56.16 μ s
tm_getload	28.50 μ s	31.42 μ s	0.58 μ s	33.21 μ s
t_create	66.60 μ s	68.23 μ s	2.31 μ s	137.55 μ s
t_start	45.06 μ s	46.21 μ s	0.70 μ s	48.24 μ s
t_suspend	37.53 μ s	40.05 μ s	0.80 μ s	42.87 μ s
t_resume	38.10 μ s	40.22 μ s	0.77 μ s	42.42 μ s
t_delete	1600.29 μ s	1601.89 μ s	0.82 μ s	1605.18 μ s
t_getid	20.31 μ s	22.08 μ s	0.49 μ s	23.79 μ s
t_comptime	30.72 μ s	32.25 μ s	0.60 μ s	33.78 μ s
ev_send	27.75 μ s	28.29 μ s	0.58 μ s	30.33 μ s
ev_recv	23.97 μ s	24.40 μ s	0.36 μ s	25.59 μ s
q_create	49.08 μ s	51.11 μ s	2.28 μ s	117.33 μ s
q_send	54.39 μ s	55.86 μ s	0.78 μ s	58.26 μ s
q_broadcast	37.92 μ s	40.14 μ s	0.77 μ s	43.14 μ s
q_urgent	56.07 μ s	57.35 μ s	0.66 μ s	59.34 μ s
q_recv	49.11 μ s	50.66 μ s	0.86 μ s	53.40 μ s
q_delete	53.91 μ s	54.93 μ s	0.56 μ s	56.73 μ s
sem_create	48.30 μ s	50.39 μ s	2.22 μ s	117.30 μ s
sem_p	33.27 μ s	34.09 μ s	0.47 μ s	35.55 μ s
sem_v	31.41 μ s	33.57 μ s	0.74 μ s	36.03 μ s
sem_delete	46.98 μ s	49.83 μ s	0.68 μ s	52.20 μ s

Figure 3.2: System Call Timing Under Medium Load ($N = 100$)

system call name	minimum	mean	std dev.	maximum
gtsk_save_context_i	2.43 μ s	2.46 μ s	0.24 μ s	9.75 μ s
null_sc	23.52 μ s	24.00 μ s	0.44 μ s	27.15 μ s
tm_gettime	21.06 μ s	21.67 μ s	0.29 μ s	22.32 μ s
get_time	32.55 μ s	86.14 μ s	1.77 μ s	87.72 μ s
tm_settime	17.40 μ s	18.77 μ s	0.34 μ s	19.62 μ s
set_time	87.75 μ s	89.61 μ s	0.53 μ s	91.14 μ s
tm_evafter	65.10 μ s	66.94 μ s	10.27 μ s	390.72 μ s
tm_evperiodic	58.74 μ s	60.41 μ s	14.52 μ s	384.90 μ s
tm_delete	60.03 μ s	61.61 μ s	0.89 μ s	64.53 μ s
tm_getload	29.49 μ s	32.25 μ s	0.55 μ s	33.93 μ s
t_create	73.11 μ s	74.83 μ s	10.31 μ s	399.51 μ s
t_start	47.04 μ s	48.23 μ s	0.66 μ s	50.13 μ s
t_suspend	39.33 μ s	41.65 μ s	0.79 μ s	44.25 μ s
t_resume	39.99 μ s	42.40 μ s	0.78 μ s	44.73 μ s
t_delete	1600.05 μ s	1601.69 μ s	0.78 μ s	1604.31 μ s
t_getid	20.61 μ s	22.13 μ s	0.50 μ s	23.67 μ s
t_comptime	30.54 μ s	32.25 μ s	0.61 μ s	33.78 μ s
ev_send	27.78 μ s	28.28 μ s	0.58 μ s	30.33 μ s
ev_recv	23.97 μ s	24.43 μ s	0.36 μ s	25.53 μ s
q_create	55.14 μ s	56.98 μ s	10.22 μ s	378.84 μ s
q_send	56.49 μ s	59.06 μ s	0.79 μ s	61.26 μ s
q_broadcast	39.60 μ s	42.65 μ s	0.62 μ s	44.82 μ s
q_urgent	60.39 μ s	62.21 μ s	0.55 μ s	63.90 μ s
q_recv	55.29 μ s	57.15 μ s	0.88 μ s	59.79 μ s
q_delete	63.84 μ s	65.92 μ s	0.76 μ s	68.34 μ s
sem_create	51.30 μ s	52.77 μ s	10.36 μ s	379.41 μ s
sem_p	32.85 μ s	33.80 μ s	0.50 μ s	35.43 μ s
sem_v	32.31 μ s	33.54 μ s	0.76 μ s	36.36 μ s
sem_delete	51.78 μ s	53.69 μ s	0.72 μ s	56.19 μ s

Figure 3.3: System Call Timing Under Heavy Load ($N = 500$)

system call name	lightly loaded	medium loaded	heavily loaded
gtsk_save_context_i	2.46 μ s	2.46 μ s	2.46 μ s
null_sc	24.08 μ s	24.02 μ s	24.00 μ s
tm_gettime	21.67 μ s	21.67 μ s	21.67 μ s
get_time	86.19 μ s	86.20 μ s	86.14 μ s
tm_settime	18.77 μ s	18.77 μ s	18.77 μ s
set_time	89.60 μ s	89.60 μ s	89.61 μ s
tm_evafter	62.43 μ s	59.64 μ s	66.94 μ s
tm_evperiodic	51.72 μ s	49.91 μ s	60.41 μ s
tm_delete	56.90 μ s	53.21 μ s	61.61 μ s
tm_getload	31.46 μ s	31.42 μ s	32.25 μ s
t_create	71.41 μ s	68.23 μ s	74.83 μ s
t_start	45.46 μ s	46.21 μ s	48.23 μ s
t_suspend	39.89 μ s	40.05 μ s	41.65 μ s
t_resume	40.21 μ s	40.22 μ s	42.40 μ s
t_delete	1603.26 μ s	1601.89 μ s	1601.69 μ s
t_getid	22.12 μ s	22.08 μ s	22.13 μ s
t_comptime	32.24 μ s	32.25 μ s	32.25 μ s
ev_send	28.29 μ s	28.29 μ s	28.28 μ s
ev_recv	24.42 μ s	24.40 μ s	24.43 μ s
q_create	53.35 μ s	51.11 μ s	56.98 μ s
q_send	55.01 μ s	55.86 μ s	59.06 μ s
q_broadcast	39.86 μ s	40.14 μ s	42.65 μ s
q_urgent	57.71 μ s	57.35 μ s	62.21 μ s
q_recv	50.74 μ s	50.66 μ s	57.15 μ s
q_delete	57.30 μ s	54.93 μ s	65.92 μ s
sem_create	48.81 μ s	50.39 μ s	52.77 μ s
sem_p	33.82 μ s	34.09 μ s	33.80 μ s
sem_v	33.83 μ s	33.57 μ s	33.54 μ s
sem_delete	48.83 μ s	49.83 μ s	53.69 μ s

Figure 3.4: System Call Timing Under Different Load Conditions

Chapter 4

Conclusions

The thesis has proposed an operating system kernel architecture, called OS/T 3.0, for embedded systems. The OS/T 3.0 architecture defines abstraction of, as well as primitives for, concurrent programming on embedded real-time systems, enabling programming under an elegant and compact model. The architecture defines multitasking with processes, and primitives for inter-process communication. It also provides primitives for exception handling and suggests an interrupt driven programming style by encapsulating and modeling exceptions as events.

In micro kernel concepts, the inter-process communication facilities are the only service a kernel should provide, and it is sufficient to build an operating system of full functionality using only inter-process communication primitives. The OS/T 3.0 has, in some sense, a micro-kernel flavor in that OS/T 3.0 only provides a set of powerful inter-process communication primitives as well as CPU house-keeping services. However, OS/T 3.0 is not a micro-kernelled operating system since the cost of such flexibility is overkill to embedded systems.

OS/T v3.0-PPC fully comply with the specification of OS/T 3.0; it has deterministic and predictable behavior, is fully preemptive multitasking, and supports both periodic and sporadic tasks. Furthermore, by explicitly pointing out the assumptions about the underlying hardware platform, it becomes, at least, easier to tell whether it is possible to port OS/T v3.0-PPC to another platform, and if it is, which part of code should be modified. OS/T v3.0-PPC is, in this context, a portable operating system itself.

We hope that we can port OS/T v3.0-PPC to similar platforms in the future to confirm that the design goal of the OS/T 3.0 architecture is not only feasible but profitable in that the porting is easy as compared with writing a new operating system.

Appendix A

Reference Manual

The application programming interface, or API for short, of OS/T 2.x is adopted after minor modifications to provide backward compatibility and make easy the design process. Among all, the system call interface may be the most important aspect of the application programming interface. The SYNOPSIS of real-time kernel system call is described as a reference manual.

A.1 Time Management

NAME

get_time, tm_gettime — obtain system's current time

SYNOPSIS

```
typedef struct time_t {
    ULONG tick;
    ULONG second;
    ULONG minute;
    ULONG hour;
    ULONG day;
    ULONG month;
    ULONG year;
} TIME;

void get_time(time)
TIME *time;

void tm_gettime(second)
ULONG *second; /* from 1970/1/1 0:0:0 */
ULONG *tick;
```

NAME

set_time, tm_settime — set system's current time¹

¹The tick field in TIME is ignored since the tick offset within a second is maintained automatically.

SYNOPSIS

```
ULONG set_time(time)
TIME *time;

void tm_settime(seconds)
ULONG seconds; /* from 1970/1/1 0:0:0 */
```

NAME

tm_wkafter — suspend the calling task for the specified interval

SYNOPSIS

```
ULONG tm_wkafter(afterticks)
ULONG afterticks;
```

NAME

tm_wkwhen, tm_wktime — suspend the calling task until the appointed system's time

SYNOPSIS

```
ULONG tm_wkwhen(time)
TIME *time;

ULONG tm_wktime(seconds, ticks)
ULONG seconds, ticks;
```

NAME

tm_evafter — send events to the calling task after the specified interval

SYNOPSIS

```
ULONG tm_evafter(afterticks, events, tmid)
ULONG afterticks, events, *tmid;
```

NAME

tm_evwhen, tm_evtime — send events to the calling task at the appointed system's time

SYNOPSIS

```
ULONG tm_evwhen(time, events, tmid)
TIME *time;
ULONG events, *tmid;

ULONG tm_evtime(seconds, ticks, events, tmid)
ULONG seconds, ticks, events, *tmid;
```

NAME

tm_evperiodic — send events to the calling task periodically

SYNOPSIS

```
ULONG tm_evperiodic(period, events, tmid)
ULONG period, events, *tmid;
```

NAME

tm_delete — delete a timer watch dog

SYNOPSIS

```
ULONG tm_delete(tm_id)
ULONG tm_id;
```

NAME

tm_getload — return the value as system loading: $\frac{10000 \times \text{working time}}{\text{total time}}$

SYNOPSIS

```
ULONG tm_getload()
```

A.2 Process Management

NAME

t_create — create a new task

SYNOPSIS

```
ULONG t_create(name, priority, stack_size, slice, pid)
ULONG name;      /* task name */
ULONG priority; /* 0 - 255, 0: highest */
ULONG stack_size;
ULONG slice;     /* 0: default time slice */
ULONG *pid;      /* obtain new task id */
```

NAME

t_start — start the execution of a newly created task

SYNOPSIS

```
ULONG t_start(pid, mode, entry, arg1, arg2, arg3, arg4)
ULONG pid;
ULONG mode;
void (*entry)();
ULONG arg1, arg2, arg3, arg4;
```

NAME

t_delete — delete a task

SYNOPSIS

```
ULONG t_delete(pid)
ULONG pid;
```

NAME

t_exit — task exit

SYNOPSIS

```
void t_exit() /* never return */
```

NAME

t_suspend — suspend a task

SYNOPSIS

```
ULONG t_suspend(pid)
ULONG pid;
```

NAME

t_resume — resume a suspended task

SYNOPSIS

```
ULONG t_resume(pid)
ULONG pid;
```

NAME

t_ident — obtain the task identifier of the named task

SYNOPSIS

```
ULONG t_ident(name, pid)
ULONG name;
ULONG *pid;
```

NAME

t_getid — obtain the task identifier of the calling task

SYNOPSIS

```
ULONG t_getid()
```

NAME

t_getpri — obtain the task priority

SYNOPSIS

```
ULONG t_getpri(pid, priority)
ULONG pid;
ULONG *priority;
```

NAME

t_setpri — set the task priority

SYNOPSIS

```
ULONG t_setpri(pid, priority)
ULONG pid;
ULONG priority; /* 0 - 255 */
```

NAME

`t_getslice` — obtain the task time slice

SYNOPSIS

```
ULONG t_getslice(pid, slice)
ULONG pid;
ULONG *slice;
```

NAME

`t_setslice` — set the task time slice

SYNOPSIS

```
ULONG t_setslice(pid, slice)
ULONG pid;
ULONG slice; /* 0: default time slice */
```

NAME

`t_comptime` — obtain the task computing time

SYNOPSIS

```
ULONG t_comptime(pid, seconds, ticks)
ULONG pid;
ULONG *seconds;
ULONG *ticks;
```

A.3 Inter-Process Communication

NAME

`ev_send` — send events to a task

SYNOPSIS

```
ULONG ev_send(pid, events)
ULONG pid;
ULONG events;
```

NAME

`ev_recv` — receive events

SYNOPSIS

```
ULONG ev_recv(events, wait_flag, wait_ticks)
ULONG events;
ULONG wait_flag; /* 0: no wait, nonzero: wait */
ULONG wait_ticks; /* ticks to wait, 0: forever */
```

NAME

`q_create` — create a message queue

SYNOPSIS

```
ULONG q_create(name, len_limit, qid)
ULONG name;      /* message queue name */
ULONG len_limit; /* max no. of messages queued */
ULONG *qid;      /* obtain the queue id */
```

NAME

q_delete — delete a message queue

SYNOPSIS

```
ULONG q_delete(qid)
ULONG qid;
```

NAME

q_send — post a message to a message queue

SYNOPSIS

```
ULONG q_send(qid, message)
ULONG qid;
ULONG *message; /* up to four long words */
```

NAME

q_broadcast — broadcast a message to all tasks waiting in a message queue

SYNOPSIS

```
ULONG q_broadcast(qid, message, num_tasks)
ULONG qid;
ULONG *message; /* up to four long words */
ULONG *num_tasks; /* obtain no. of tasks that receives */
```

NAME

q_urgent — post an urgent message to the head of a message queue

SYNOPSIS

```
ULONG q_urgent(qid, message)
ULONG qid;
ULONG *message; /* up to four long words */
```

NAME

q_recv — receive a message from a message queue

SYNOPSIS

```
ULONG q_recv(qid, buf, wait_flag, wait_ticks)
ULONG qid;
ULONG *buf; /* up to four long words */
ULONG wait_flag; /* 0: no wait, nonzero: wait */
ULONG wait_ticks; /* ticks to wait, 0: forever */
```

NAME

q_ident — obtain the queue identifier of the named message queue

SYNOPSIS

```
ULONG q_ident(name, qid)
ULONG name;
ULONG *qid; /* obtain the queue id */
```

NAME

sem_create — create a semaphore

SYNOPSIS

```
ULONG sem_create(name, init_value, max_value, sid)
ULONG name;
ULONG init_value; /* initial value */
ULONG max_value; /* max. value */
ULONG *sid; /* obtain semaphore id */
```

NAME

sem_delete — delete a semaphore

SYNOPSIS

```
ULONG sem_delete(sid)
ULONG sid;
```

NAME

sem_p — acquire a semaphore

SYNOPSIS

```
ULONG sem_p(sid, wait_flag, wait_ticks)
ULONG sid;
ULONG wait_flag; /* 0: no wait, nonzero: wait */
ULONG wait_ticks; /* ticks to wait, 0: forever */
```

NAME

sem_v — release a semaphore

SYNOPSIS

```
ULONG sem_v(sid)
ULONG sid;
```

NAME

sem_ident — obtain the semaphore identifier of the named semaphore

SYNOPSIS

```
ULONG sem_ident(name, sid)
ULONG name;
ULONG *sid; /* obtain semaphore id */
```

A.4 Exception Handling

NAME

isr_hook — specify actions when an interrupt takes place

SYNOPSIS

```
ULONG isr_hook(vector, pid, events)
ULONG vector; /* exception vector */
void (*isr)(); /* isr entry point */
ULONG pid;
ULONG events;
```

NAME

isr_unhook — cancel actions when an interrupt takes place

SYNOPSIS

```
ULONG isr_unhook(vector)
ULONG vector; /* exception vector no. */
```

NAME

isr_getinfo — obtain information about handling an interrupt

SYNOPSIS

```
ULONG isr_(vector)
ULONG vector; /* exception vector */
ULONG *name; /* obtain name of vector */
void (**isr)(); /* obtain isr entry point */
ULONG *pid; /* pid of ISR */
ULONG *events; /* events sent */
```

Appendix B

STBLD

```
#include <stdio.h>

#define chend(x) ((x)=((x)&0x000000ff)<<24| ((x)&0x0000ff00)<<8|\
                ((x)&0x00ff0000)>>8 | ((x)&0xff000000)>>24)

typedef unsigned long ULONG;

/* 57328 is chosen to make sizeof(struct symbol_table_t) = 65536 bytes */
static struct symbol_table_t
{
    ULONG img_start;
    ULONG img_size;
    ULONG img_end;
    ULONG max;
    ULONG addr[1024];
    ULONG name[1024];
    char pool[57328];
}
symtab;

static struct symbol_entry_t
{
    ULONG number, value, size;
    char bind[256], type[256], sect[256], name[256];
}
syment;

static char symtabstr[] = "gerr_symbol_table_storage";

int
main (int argc, char *argv[])
{
    char linebuf[4096], tokbuf[64], *elffile;
    ULONG flag, strptr, offset, d_addr, d_off, valtmp, strtmp, *buf;
    int i, bytes;
    FILE *fp;

    symtab.img_start = 0x10000;
    symtab.img_size = 0;
    symtab.img_end = 0x10000;
    symtab.max = 0;
```



```

    strptr = flag = offset = d_addr = d_off = 0;

    while (gets (linebuf) != NULL)
    {
        if (flag)
        {
            sscanf (linebuf, "%d 0x%x %d %s %s %s %s",
                    &syment.number, &syment.value, &syment.size,
                    syment.bind, syment.type, syment.sect, syment.name);

            if (syment.size != 0)
            {
                if (strcmp (syment.name, symtabstr) == 0)
                {
                    offset = syment.value;
                }

                symtab.addr[symtab.max] = syment.value;
                symtab.name[symtab.max] = strptr;
                symtab.max++;

                strcpy (&symtab.pool[strptr], syment.name);
                strptr += strlen (syment.name) + 1;
            }
        }
        else if (sscanf (linebuf, "    Section #%*d: %s %*s addr=0x%X, off=0x%X",
                        tokbuf, &valtmp, &strtmp) > 0)
        {
            tokbuf[63] = (char) 0;

            if (strcmp (tokbuf, ".data,") == 0)
            {
                d_addr = valtmp;
                d_off = strtmp;
            }

            if (valtmp > symtab.img_end)
            {
                gets (linebuf);
                if (sscanf (linebuf, "    size=%d", &strtmp))
                {
                    if (strtmp != 0)
                    {
                        symtab.img_end = valtmp + strtmp;
                    }
                }
            }
        }
        else if (strncmp (linebuf, "    -----", 10) == 0)
        {
            ++flag;
        }
    }

    symtab.img_size = symtab.img_end - symtab.img_start;

```

```

fprintf (stdout, "%d bytes used for %d symbols.\n", strptr, symtab.max);

fprintf (stdout, "binary image: addr=0x%X, size=%d(0x%X)\n",
        symtab.img_start, symtab.img_size, symtab.img_size);

do
{
    /* bubble sort */
    flag = 0;

    for (i = 1; i < symtab.max; ++i)
    {
        if (symtab.addr[i - 1] > symtab.addr[i])
        {
            valtmp = symtab.addr[i - 1];
            strttmp = symtab.name[i - 1];

            symtab.addr[i - 1] = symtab.addr[i];
            symtab.name[i - 1] = symtab.name[i];

            symtab.addr[i] = valtmp;
            symtab.name[i] = strttmp;

            ++flag;
        }
    }
} while (flag);

for (i = 0; i < symtab.max; ++i)
{
    chend (symtab.addr[i]);
    chend (symtab.name[i]);
}

chend (symtab.max);
chend (symtab.img_end);
chend (symtab.img_size);
chend (symtab.img_start);

if (offset == 0)
{
    fprintf (stderr, "%s: can't find %s in elf file.\n", argv[0], symtabstr);
    exit (1);
}
else if (d_addr == 0 || d_off == 0)
{
    fprintf (stderr, "%s: can't find data section in elf file.\n", argv[0]);
    exit (1);
}

if (argc < 2)
{
    elffile = "OST";
}
else

```

```

    {
        elffile = argv[1];
    }

    if ((fp = fopen (elffile, "r+b")) == (FILE *) 0)
    {
        fprintf (stderr, "%s: can't open elf file %s.\n", argv[0], elffile);
        exit (1);
    }

    offset = offset - d_addr + d_off;

    fseek (fp, offset, SEEK_SET);

    bytes = fwrite ((void *) &symtab, sizeof (char), sizeof (symtab), fp);

    fclose (fp);

    fprintf (stdout, "%d bytes written to %s, offset = 0x%08X.\n",
            bytes, elffile, offset);

    exit (0);
}

```

Bibliography

- [1] John Sinclair, editor. *Collins Cobuild English Dictionary*. HarperCollins, 1995.
- [2] Ching-He Lin. A real-time kernel: Design and implementation. Master's thesis, National Taiwan University, Jun 1994.
- [3] I-Chun Chuang. A portable real-time operating system kernel: Design and implementation. Master's thesis, National Taiwan University, Jun 1996.
- [4] Jean J. Labrosse. *μ C/OS: The Real-Time Kernel*. R & D, 1992.
- [5] Integrated Systems, Inc. *pSOSystem System Concepts*, Nov 1996.
- [6] Integrated Systems, Inc. *pSOS+/68K User's Manual*, Sep 1992.
- [7] Michael Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Jun 1997.
- [8] Douglas Comer. *Operating System Design, the Xinu Approach*. Prentice-Hall, 1984.
- [9] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, fourth edition, 1995.
- [10] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, second edition, 1997.
- [11] James R. Pinkert and Larry L. Wear. *Operating Systems*. Prentice-Hall, 1989.
- [12] Jerry Young. *Insider's Guide to PowerPC Computing*. Que, 199?
- [13] Gary Kacmarcik. *Optimizing PowerPC Code*. Addison Wesley, Apr 1995.
- [14] Tom Shanley. *PowerPC System Architecture*. Addison Wesley, Feb 1995.
- [15] Kip McClanahan. *PowerPC Programming for Intel Programmers*. IDG, Jun 1995.
- [16] IBM Corporation. *IBM PowerPC 403GA User's Manual*, Mar 1995.
- [17] Steve C. McConnell. *Code Complete: a Practical Handbook of Software Construction*. Microsoft Press, 1993.

- [18] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968.
- [19] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, June 1994.
- [20] Tool interface standard (tis) portable formats specification, Oct 1993.
- [21] IBM Corporation. *High C/C++TM Programmer's Guide*, first edition, Aug 1995.
- [22] IBM Corporation. *ELF Linker User's Guide*, second edition, Nov 1995.
- [23] IBM Corporation. *ELF Assembler User's Guide*, first edition, Aug 1995.
- [24] Leslie Lamport. *L^AT_EX: A Document Preparation System*. Addison-Wesley, second edition, 1994.
- [25] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, 1994.
- [26] Helmut Kopka and Patrick W. Daly. *A Guide to L^AT_EX: Document Preparation for Beginners and Advanced Users*. Addison-Wesley, 1992.
- [27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.
- [28] Leland L. Beck. *System Software: An Introduction to Systems Programming*. Addison-Wesley, second edition, Mar 1990.
- [29] Richard A. Burgess. *Developing Your Own 32-Bit Operating System*. Sams, 1995.
- [30] IBM Corporation. *RISCWatch 400 Debugger User's Guide*, fifth edition, Oct 1995.