Software/Hardware RTOS and System Overhead in a Hard Real-Time Environment

by

Khaled Rajeh, BSc

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

May 2010

Your file Votre référence
ISBN: 978-0-494-68642-3
Our file Notre référence
ISBN: 978-0-494-68642-3

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis

Software/Hardware RTOS and System Overhead
in a Hard Real-Time Environment

submitted by

Khaled Rajeh, BSc

in partial fulfillment of the requirements for

the degree of Master of Applied Science in Electrical and Computer Engineering

---

Chair, Prof. Howard Schwartz, Department of Systems and Computer Engineering

---

Thesis Supervisor, Prof. Trevor Pearce

Carleton University

May 2010

Abstract

Using a Real-Time Operating Systems (RTOS) in an embedded system is a good solution for reducing design time, but an RTOS can affect runtime behaviours. Developers of hard real-time systems should be very careful, as these RTOS effects may result in missing a critical deadline. An RTOS lacks determinism because of associated jitter and system overhead. Therefore, it is important to have a tool that can be used to decide whether or not a system using an RTOS can meet hard real-time constraints.

This thesis extends the standard task response time model to account for system overhead and to improve the prediction of task response times. Furthermore, runtime determinism can be improved by partitioning the RTOS into hardware and software components. A Field Programmable Gate Array (FPGA) is used to build a prototype to explore a partitioning of the μC/OS-II RTOS. A case study illustrates the improved determinism of the partitioned RTOS system, and provides performance data needed to compare the standard response time model to the improved model. The results show that applying the improved model to the partitioned RTOS can improve the response time prediction accuracy by several orders of magnitude.

Acknowledgments

I would like to thank my supervisor, Professor Trevor Pearce for his supervision, suggestion, and ideas during the thesis process, especially for his great effort during the communication with Xilinx support engineer and the thesis writing phase.

I must also thank CMC for their donation of the target system, and thanks are also for their support engineer Susan for her help and fast feedback. The XILINX University Program support team was also very helpful in dealing with a suspected problem in the target board.

My great thanks are also to my mother, my family, my wife and my children for their full support and their patience during this thesis.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
BSP	Board Support Package
CPU	Central Processing Unit
ECB	Event Control Block
EDF	Earliest Deadline First
ELLF	Enhanced Least Laxity First
FPGA	Field Programmable Gate Array
GBI	Generic Bus Interface
GUI	Graphical User Interface
HRT	Hard Real-Time
HW	Hardware
IPC	Inter-Process Communication
ISR	Interrupt Service Routine
µC/OS-II	Micro-Controller Operating System, version 2
OPB	On-chip Peripheral Bus
OS	Operating System
PIT	Programmable-Interval Timer
PLB	Processor Local Bus
PPC	PowerPC

RISC	Reduced Instruction Set Computer
RM	Rate Monotonic
RTM	Real-time Task Manager
RTOS	Real-Time Operating System
RTU	Real-Time Unit
SOC	System-on-Chip
SW	Software
TCB	Task Control Block
TDBI	Technology Dependant Bus Interface
VME	Versa Modular Eurocard
XPS 9.1i	Xilinx Platform Studio, version 9.1i

1 Introduction

Developers have been faced with growing complexity in both the hardware and software used in embedded systems. A Real-Time Operating System (RTOS) has become a very good solution since it provides developers with easy access to the hardware and reduces design time. Unfortunately, an RTOS may introduce overheads and may add to the non-determinism of the system. This is especially problematic for Hard Real-Time (HRT) systems, where developers are required to show that the systems meet timing constraints.

Non-deterministic behaviour is one of the downsides of using an RTOS. An RTOS implemented in software often requires a periodic hardware interrupt which works as the system's heartbeat and allows the RTOS to do periodic processing. The periodic interrupt is known as the periodic tick. An RTOS timer known as the tick-timer is based on the periodic tick, and as will be seen later, the tick-timer can introduce variations in the response time of the system. This variation is called jitter and it reduces the determinism of the system's temporal behaviour.

System overhead is also undesirable behaviour of an RTOS. Using the RTOS causes system overhead since it requires more memory and must often share a processor with application tasks. Switching between tasks and managing the tick-timer all contribute to system overhead.

Furthermore, an RTOS often allows tasks to be delayed for a number of ticks. Some RTOSs use the delay service to implement periodic application tasks. For a preemptive RTOS, using the delay service in this way can cause timing problems which are discussed in detail later.

Some researchers [1, 2, 3, 4, 5] have tackled these software problems by off-loading some RTOS functionalities to an external hardware Field Programmable Gate Array (FPGA). Unfortunately, this solution requires a considerable increase in performance in order to compensate for the chip to chip communication overhead between the processor and the FPGA. The rapid innovations in System-on-Chip (SOC) technology now allows the CPU core and the FPGA fabric to be included in one chip, and research to partition the RTOS between the software (SW) and hardware (HW) has become increasingly valuable.

The major contribution of this thesis is to improve the standard task response time prediction model by addressing the overhead, jitter, timing and non-determinism problems associated with a pure software RTOS. The thesis goes further by off-loading some of the time consuming functionalities from SW to HW, and analyzing response times in the partitioned SW/HW RTOS system. The significant improvement in determinism of the SW/HW RTOS is obtained by implementing the periodic tick-timer in hardware. This eliminates the jitter in task execution time and the system overhead due to managing the tick-timer. In addition, the SW/HW RTOS supports the management of periodic tasks in hardware to solve the timing problem associated with using the delay

service for this purpose. A verification technique is also provided to verify that HRT tasks meet their timing deadlines. The technique profiles timing information using hardware timers as well as resource activity and task information for each task. The implementation work has uncovered a suspected hardware problem in the AP1000 target board.

The development of real-time systems is central to the thesis, and chapter 2 provides background information about scheduling, priority assignment strategies, schedulability testing and the development process for embedded systems. An analysis of the state of the art in research related to the use of hardware support for RTOS functionality is given in chapter 3. The analysis shows that many of the researchers concentrate on improving the performance of the real-time systems, but there is insufficient consideration of system overheads and the impact on a performance prediction model. A concise statement of the thesis, along with the motivation and scope of work are presented in chapter 4. The major contribution of the thesis is a theoretical accounting for system overheads that improves performance predictions.

The theory is developed in chapter 5 for a system that uses a pure SW RTOS. µC/OS-II is used as the SW RTOS. The system design and implementation are discussed along with a description of the time drift and jitter problems in the system. Attention then turns to a partitioned SW/HW RTOS. The design and implementation of the partitioned system is discussed in chapter 6, including details about the modified µC/OS-II software component, the major hardware blocks and the logical states of the HWRTOS. The

chapter ends with a discussion of the major differences between the pure SW RTOS and the partitioned SW/HW RTOS.

A theory that addresses system overheads relies on accurate measurements of characteristic behaviours. Chapter 7 develops a system overhead equation for the tick-timer based on practical measurements for both the pure software and partitioned RTOSs. The measurements account for different numbers of tasks and different tick-timer frequencies. An improved model for response time analysis is then developed and discussed in chapter 8. The improved model includes system overhead which makes it more accurate. A case study is conducted for both the pure SW and partitioned RTOS systems and the comparison of the predicted response times to the measured response times provides an error estimate for the prediction model. The case study results show that for system using the pure SW RTOS with a tick frequency of 4000 tick/sec, the maximum error percentage using the improved model is about 1.8% while it reaches about 65.8% using the standard model. The partitioned RTOS results are even more impressive, as the error dips below 0.001%.

The contributions of the research are drawn out further through critical analysis. A performance comparison between both systems is discussed in chapter 9. An example shows how the performance improvements due to eliminating the tick-timer overhead could allow a system based on the SW/HW RTOS to meet timing deadlines in a case where the pure SW RTOS overheads caused the system to fail. The partitioned RTOS also has some implementation limitations that might be reduced through further research.

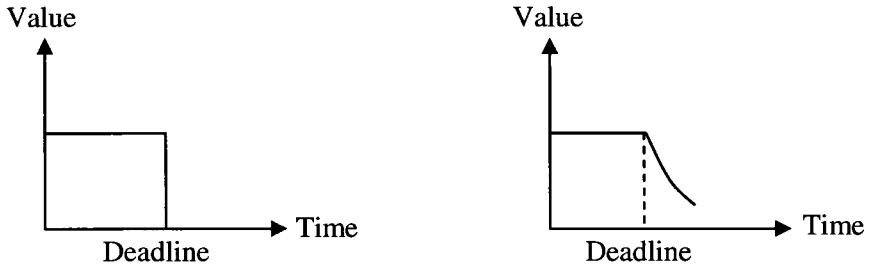
The collected results show that the required communication between software and the hardware components can degrade the performance of some of the service calls, and software overhead associated with context switching is still present. Chapter 10 presents a general discussion of issues, including a suspected problem with the target board and some practical limitations of the approach taken in this research. Finally, chapter 11 provides conclusions and some recommendation for future work.

2 Real-Time Systems

A real-time system is a system that responds to unpredictable external stimuli with timely and predictable outputs [6]. The two important words in this definition are timely and predictable. The developer should be able to predict the output of the system in response to given inputs, and the response should be within timing deadlines. This chapter presents background information that is relevant to predicting timely responses. The real-time operating system and its associated scheduling algorithm are critical factors. The terminology used here is taken from Labrosse [7], unless noted otherwise.

There are two major types of real-time system, namely hard and soft. A **Hard Real-Time System** requires a validation technique to show that the system response is always within specified deadlines [8]. Figure 2-1.a depicts hard real-time system behaviour where output value goes down sharp at the specified deadline; examples of this type of system are a heart pacemaker, an air-bag system, and a car engine control system. A **Soft Real-Time System** recovers correctly if a deadline is occasionally missed. Missing a deadline reduces the quality of the system [6]. Figure 2-1.b depicts soft real-time system behaviour where output value goes down gradually after the specified deadline; examples of this type of system are a vending machine, a live audio-video system, and a telephone dial tone.

In low complexity systems such as toys or microwave ovens, designers use a technique called **Foreground/Background** (also referred to as a cyclic executive or



a. Hard Real-Time System b. Soft Real-Time System

Figure 2-1: Hard/Soft Real-Time Systems

super loop). The application in these systems consists of two levels, namely a task level (background) and an interrupt level (foreground). The background is an infinite loop which accomplishes the desired job, while the foreground consists of Interrupt Service Routines (ISR) that take care of external events as shown in Figure 2-2. Due to the existence of unpredictable external events and checking of ISR inputs at a fixed point in the loop, asynchronous ISRs and subsequent processing can inject jitter which can lead to nondeterministic responses to the external events.

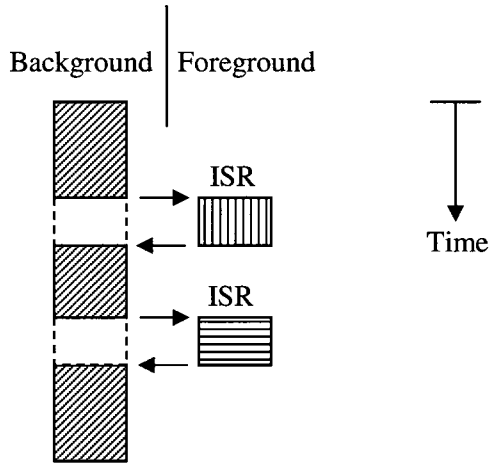


Figure 2-2: Foreground/Background Systems [7]

Developers often use an RTOS to help them cope with rapid innovations in technology, the increasing complexity of the real-time systems, and the need to control multiple applications. An RTOS provides runtime support for many functions needed by the applications. An RTOS organizes and manages an application as a set of tasks. A task is a simple program (typically an infinite loop) that supports some system function. Each task typically has a deadline, and predicting whether tasks meet their deadlines can be an onerous development challenge.

One of the most important functions of the RTOS is the scheduling process of finding the next task that should take control of the CPU and run. The RTOS component responsible for this operation is called the scheduler. As shown in Figure 2-3, the scheduler uses a ready queue to manage tasks. The scheduler uses its scheduling algorithm to order tasks in the queue such that the next task to run can be dispatched from the front of the queue. Preempting a running task returns the task to the ready queue. The time at which the scheduler decides which task to run next is called a scheduling point. A set of tasks is schedulable when the scheduling algorithm ensures that the tasks always meet their deadlines. The process of changing from the preempted task to the dispatched task is called a context switch. The process involves saving the current contents of the CPU registers for the preempted task and loading a stored copy of the CPU registers for the dispatched task. An RTOS will often include services that allow tasks to communicate and synchronize.

As in Figure 2-4, the RTOS kernel resides between the application and the hardware, so it provides the application with easy access to the hardware.

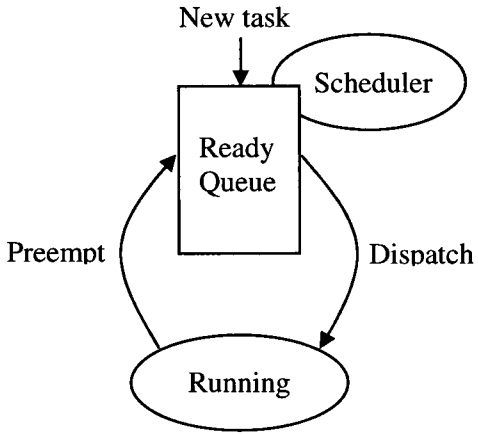


Figure 2-3: Scheduler

The three major task types in real-time systems [8] are periodic, aperiodic and sporadic tasks. A **periodic task** is also known as time-driven and it is released (becomes available to run) at a fixed frequency or regular time interval. This task can be characterized by its execution time, period (time between successive releases) and deadline. An **aperiodic task** is also known as event-driven and it is non periodic since the arrival time is unknown at design time. It often has a soft time constraint and can be characterized by its execution time and arrival time. A **sporadic task** is an aperiodic task

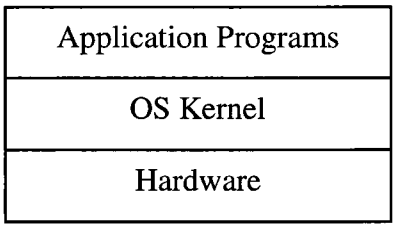
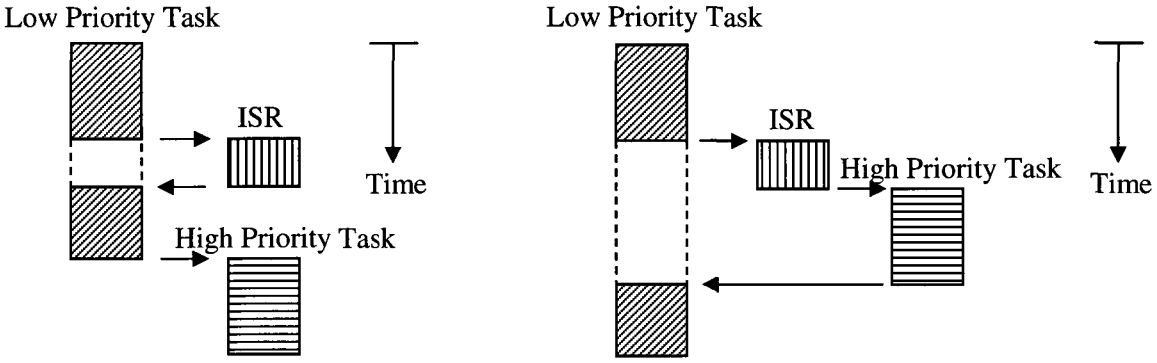


Figure 2-4: OS-based System

with a defined minimum inter-arrival time. It often has a hard time constraint and can be characterized by its execution time, minimum inter-arrival time and deadline.

The ability of an application to respond to inputs often depends on the task preemption policy inherent to the RTOS. **Non-Preemptive Kernels** supports cooperative multitasking because each task has to relinquish the CPU to maintain concurrency. External events are serviced by ISRs, but even if an ISR makes a higher priority task ready to run, the ISR will return control to the interrupted task which will relinquish the CPU eventually as depicted in Figure 2-5.a. Since the high priority task must wait for the low priority task to relinquish the CPU, the system's response time to the external event is nondeterministic. With **Priority-Preemptive Kernels**, each task has a priority and the highest priority task always gets immediate control of the CPU. Figure 2-5.b shows the case where an ISR makes a higher priority task ready to run. At the end of the ISR, the higher priority task will preempt the lower priority task to complete the system's response to the external event. This scheme still suffers from execution jitter



a. Non-Preemptive Kernels b. Priority-Preemptive Kernels

Figure 2-5: Non-Preemptive and Priority-Preemptive Systems [7]

due to preemption by higher priority tasks, but the response time of higher priority task is more deterministic, and that's why most commercial real-time operating systems are priority-preemptive.

Task priorities can be assigned using different policies. In **Static Priority Assignment**, known also as fixed priority assignment, the system designer gives each task a priority and this priority stays fixed over the life time of the task. Static priority assignment could depend on task parameters such as the task period, as in Rate Monotonic (RM) scheduling algorithm where the task with the shortest period gets the highest priority. In **Dynamic Priority Assignment**, the scheduler assigns the priority of each task dynamically at run time, as in Earliest Deadline First (EDF) scheduling algorithm where the task with the nearest deadline gets the highest priority.

The **Rate Monotonic Scheduling Algorithm** is used to schedule periodic tasks where each task is assigned a priority based on its period. The task with the shortest period is assigned the highest priority to run first as shown in Figure 2-6 for the next example. In this case, task priority is static, with T2 having higher priority than T1. This is dictated by the fact that T2 has a shorter period than T1. Whenever T2 is ready at any scheduling point, then it will gain control of the CPU over T1 as can be seen at times 0, 3, 6, 9 and 12. Time 0 is called critical instant where all tasks are released and are ready to run simultaneously. A critical instant represents the worst-case loading of a system by periodic tasks. The period from time 0 to 15 is called hyperperiod which is the least

common multiple of all task periods. If a task set is schedulable in the first hyperperiod, then it is schedulable after that.

Example:

	Computation Time	Period	Deadline (D)
Task1 (T1)	3	5	5
Task2 (T2)	1	3	3

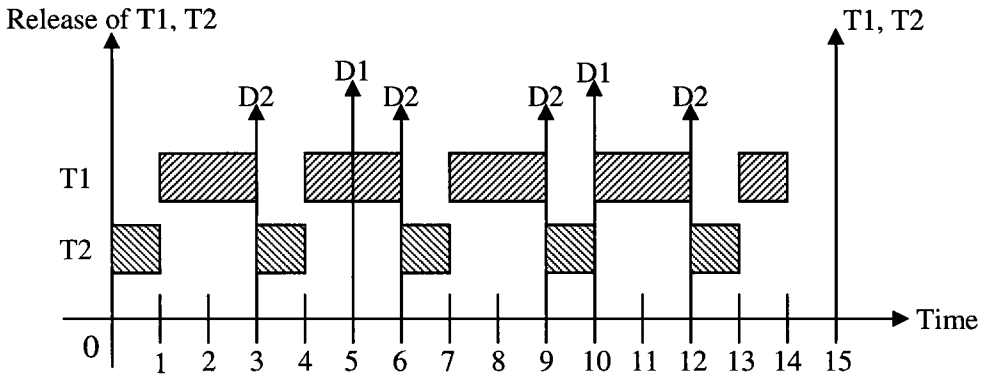


Figure 2-6: Rate Monotonic Scheduling Algorithm

The **Earliest Deadline First Scheduling Algorithm** is a dynamic scheduling algorithm where at each scheduling point the scheduler will assign the highest priority to the task that has the nearest deadline so it runs first and meets its deadline. This algorithm is optimal on a preemptive uni-processor system. This means that if a set of tasks can be scheduled by any algorithm such that they complete by their deadlines, then EDF can schedule them to meet their deadlines. EDF can achieve 100% CPU utilization so tasks can fully occupy the CPU and make it 100% busy. Figure 2-7 shows the EDF algorithm for the same tasks in the previous example. In this case, the task priority is dynamic. At time 0, T2 is higher priority than T1 since it has shorter deadline (3 versus 5), while at

time 3, T1 is higher priority than T2 as the time left until T1's deadline is 2 versus 3 for T2.

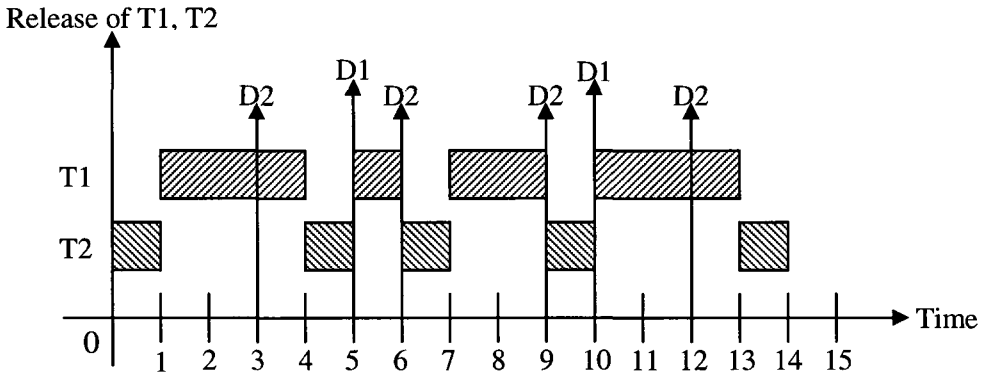


Figure 2-7: Earliest Deadline First Scheduling Algorithm

2.1 Schedulability Test

Development of hard real-time system requires a method to determine whether a set of tasks are schedulable or not. Many schedulability tests have been proposed, including the utilization test, the time demand test and the response time test.

The **Utilization Test** presented by Liu and Layland in 1973 is based on rate monotonic priority assignment [9]. There are four main assumptions for this test namely: tasks are periodic, tasks are independent (i.e. running a task does not depend on interaction with or completion of other tasks), the test is for a uni-processor system and system overhead is zero. Research has addressed all of the assumptions, however the overhead assumption has been the most difficult to overcome.

The processor utilization (U) by a task is the proportion of time the processor spends executing the task. U can be calculated from the following equation:

$$U = \frac{e}{p} \qquad \text{(Equation 2-1)}$$

where *e* is the execution time of the task and *p* is the period of the task.

The total processor utilization by all tasks can be calculated as follows:

$$U_{total} = \sum_{i=1}^{n} \frac{e_i}{p_i} \qquad \text{(Equation 2-2)}$$

where *n* is the number of tasks in the system, and e_i and p_i are the execution time and period of $task_i$.

Liu and Layland show that the least upper bound to the processor utilization for n tasks with rate monotonic priority assignment is:

$$U_{RM}(n) = n(2^{1/n} - 1) \qquad \text{(Equation 2-3)}$$

The simple utilization test for schedulability is based on the least upper bound for processor utilization:

- If $U_{total} > 1$, then the task set is not schedulable
- If $U_{total} \leq U_{RM}(n)$, then the task set is schedulable
- If $1 \geq U_{total} > U_{RM}(n)$, then a more detailed analysis should be conducted

The utilization test is sufficient but not necessary. This means that the validity of the second condition above is sufficient to conclude that the task set is schedulable. If the

third condition is valid, then the task set may still be schedulable, but this can only be determined by a more detailed analysis.

The **Time-Demand Test** is a more detailed analysis which could be done when the utilization test is not sufficient ($1 \geq U_{total} > U_{RM}(n)$). The test involves finding the total task demand for processor execution time (i.e. the demand due to a task and all higher priority tasks) at scheduling points. If the task demand at any scheduling point is less than the delivered processing up to that point, then the task is schedulable, otherwise the task is not schedulable. The test must be repeated for all tasks that have hard deadlines. The time-demand analysis is both sufficient and necessary.

The **Response-Time Test** is another detailed analysis that can be used when the utilization test result is not definitive. This test involves calculating the response time for each task from a critical instant including the delay caused by all higher priority tasks. If the response time of $task_i$ is less than its deadline D_i, then it is schedulable. The test is characterized by the recurrence relation:

$$R_i = e_i + \sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil e_j \qquad \text{(Equation 2-4)}$$

where: R_i is the task response time for $task_i$, e_i is the task execution time of $task_i$

$\left\lceil \frac{R_i}{p_j} \right\rceil$ is the ceiling function of the number of times that a $task_j$ is activated, and

$\sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil e_j$ is the total delay caused by all higher priority tasks

The recurrence relation can be solved by starting with an estimate of $R_i^0 = e_i$ with no delay caused by higher priority tasks. The recurrence calculates refined estimates, until it converges to the response time $R_i^{n+1} = R_i^n$. If $R_i^n \leq D_i$ then $task_i$ meets its deadline and is schedulable. Response-time analysis is also both sufficient and necessary.

2.2 Embedded System Development

Development for embedded systems usually involves both hardware and software. A growing trend in hardware development is the use of a programmable SOC, which includes one or more processors embedded as hard or soft cores in the chip. The software typically includes the application code, the RTOS and the Board Support Package (BSP) as shown in Figure 2-8. The RTOS and the BSP are located in the middle layers between the hardware and the user application code. The BSP is needed to access peripheral devices outside the processor and is specific for each board. The RTOS includes the kernel and the processor port. The kernel contains the processor-independent code and provides the RTOS services. The processor port is the processor-dependant code and data type definitions, and should be modified when a new processor architecture is used. The development tool is often an integrated development environment for creating a complete embedded system.

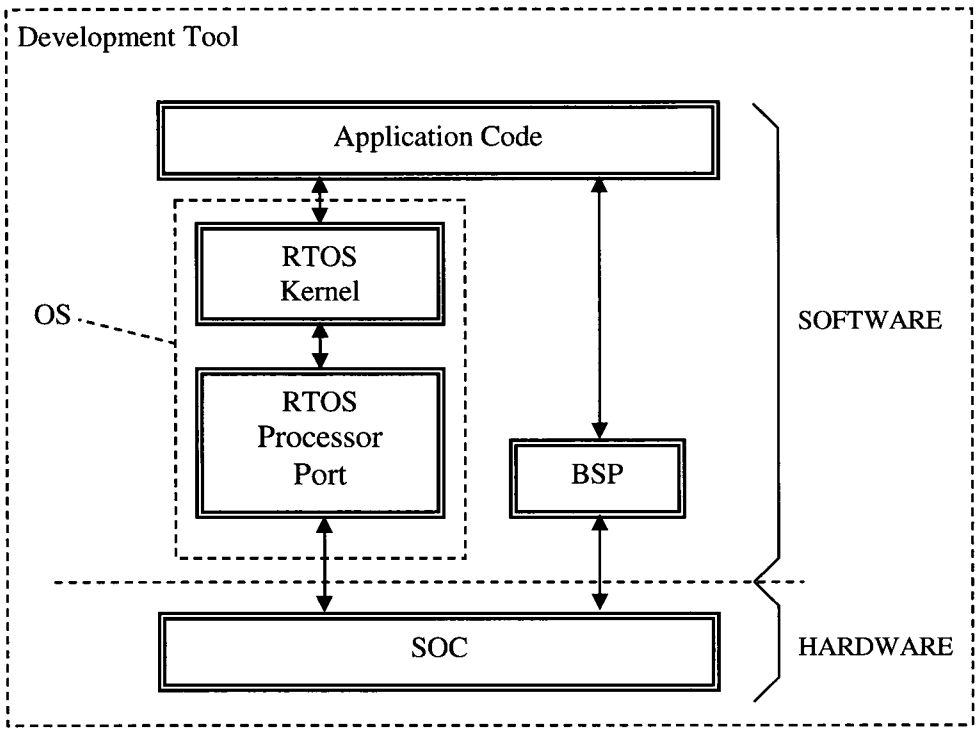


Figure 2-8: Embedded System Model (modified from [10])

This research uses the μC/OS-II RTOS [7], the Xilinx XPS 9.1i development tool [11], and AP1000 hardware board [12]. The programmable SOC includes a hard core PowerPC (PPC) as shown in Figure 2-9. A detailed description of each part follows.

The Micro-Controller Operating System, version 2 (μC/OS-II) is a preemptive real-time multitasking OS. The μC/OS-II kernel is written entirely in ANSI C which makes it highly portable. The μC/OS-II PowerPC 405 port is written in assembly code for easy adaptability to different processor architectures.

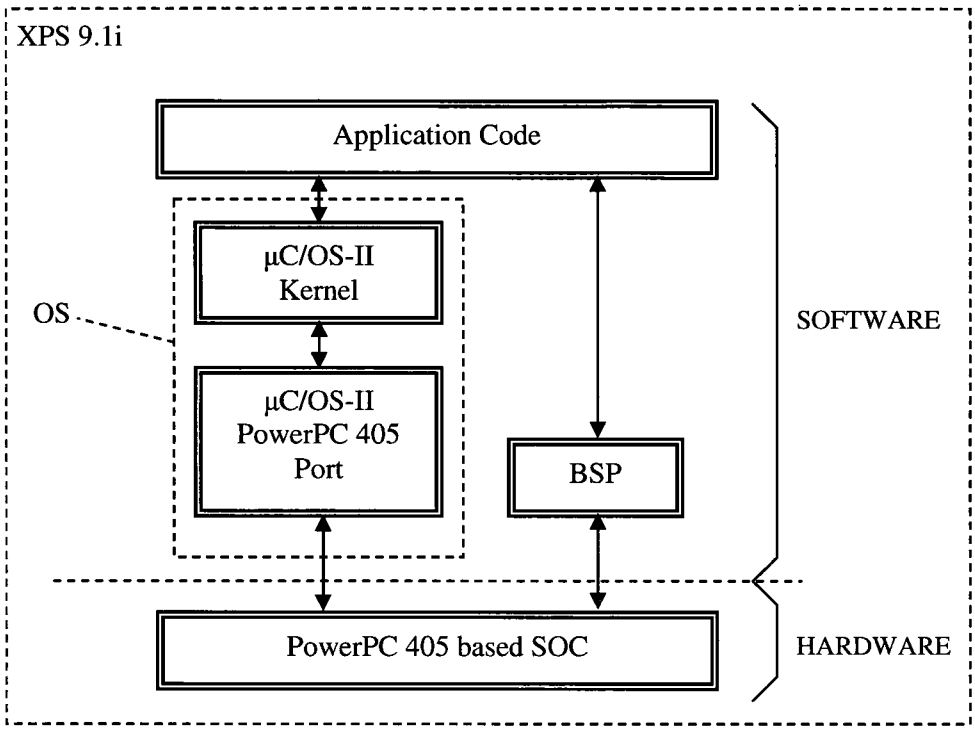


Figure 2-9: µC/OS-II Based Embedded System (modified from [10])

The µC/OS-II operating system manages tasks using logical states. A task in the µC/OS-II can be in one of five states: dormant, ready, waiting, running, and ISR. As shown in Figure 2-10, the starting state is the dormant state. Creating a task puts the task in the ready state. The task stays in the ready state until an appropriate scheduling point, where the scheduler moves the task to the running state. At any time, if a higher priority task is scheduled to run then it preempts the current task and sends that task back to the ready state. While a task is running, an external interrupt moves the task to the ISR running state until the end of the ISR where the task goes back to the running state. If the task requests a resource which is not available then it goes to the waiting state, and when

the resource is granted the task then returns to the ready state. The deletion of the task causes it to go to the starting (dormant) state.

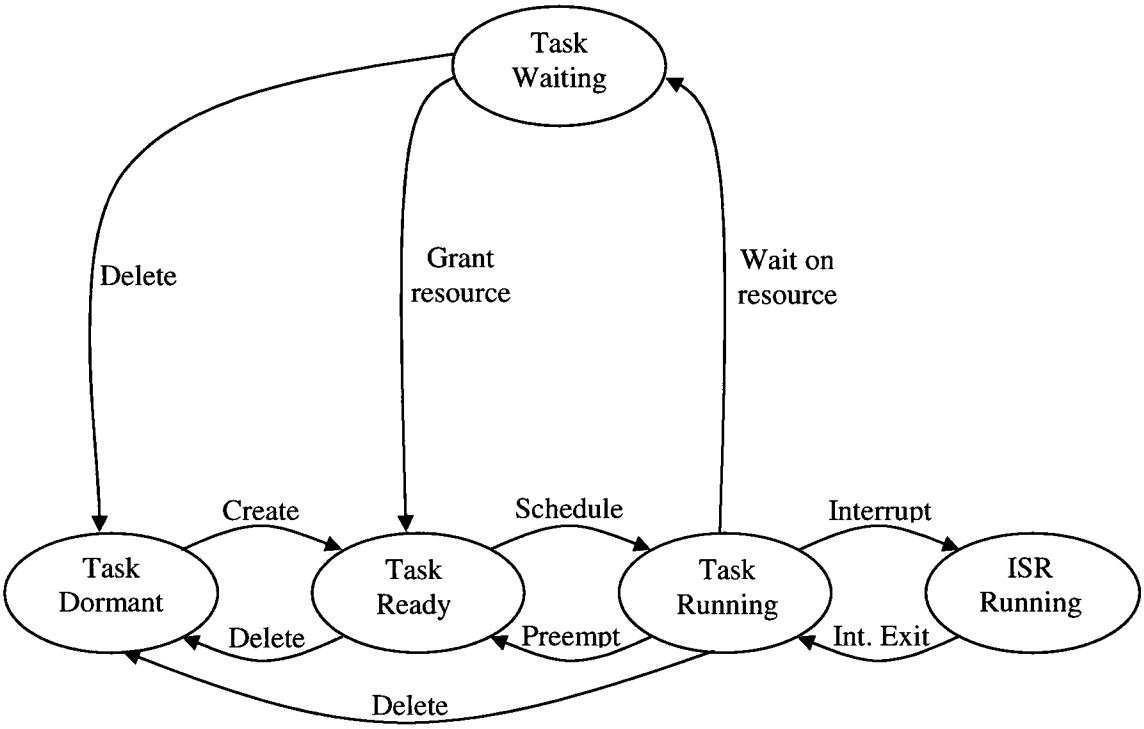


Figure 2-10: µC/OS-II Task State Transitions [7]

It is important to understand how µC/OS-II manipulates the ready list because this functionality is implemented in the hardware RTOS as described later. There are three major operations performed by the scheduler: add a task to the ready list, remove a task from the ready list and find the highest priority task in the ready list. Each task in the system is assigned a unique priority number where the lower the number means the higher the priority. µC/OS-II can manage up to 64 tasks. As can be seen in Figure 2-11, the ready list consists of two variables, OSRdyTbl and OSRdyGrp. OSRdyTbl is an array

of eight bytes, each bit represents a task priority from 0 (highest priority) to 63 (lowest priority = idle task). OSRdyGrp is one byte with each bit acts as an index to a row in the OSRdyTbl.

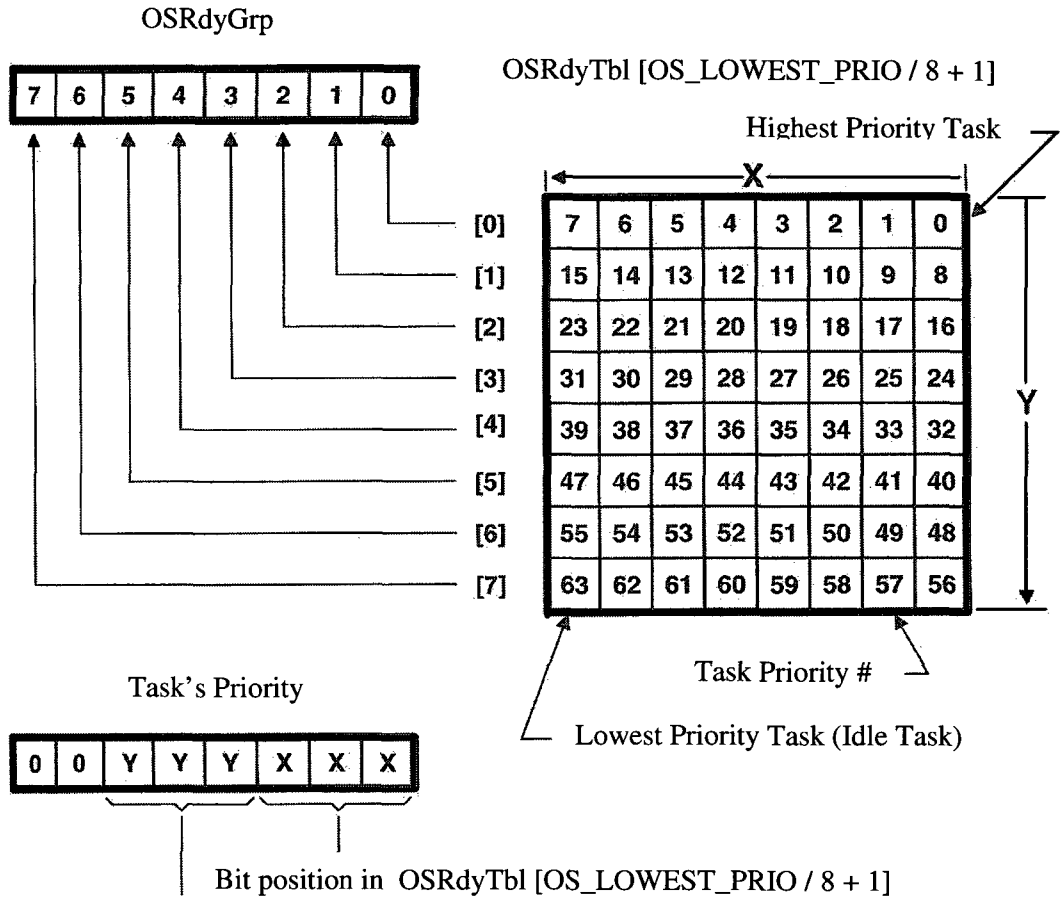


Figure 2-11: Ready List in µC/OS-II [7]

The task priority byte is organized as two 3-bit fields, denoted X and Y as shown in Figure 2-11. X and Y collectively specify the position of a bit in OSRdyTbl that represents the task with that priority. X represents the bit position in a row of OSRdyTbl. While Y represents the row of OSRdyTbl and also identifies the bit position in OSRdyGrp associated with that row. OS_LOWEST_PRIO is a defined constant for the lowest priority in the system and it is equal to 63.

Adding a task to the ready list is done by setting the task's corresponding bits in the OSRdyTbl and OSRdyGrp. Removing a task from the ready list is done by clearing the task's corresponding bits in the OSRdyTbl and OSRdyGrp. Finding the highest priority task in the ready list is a very important function, and it has to be done fast to reduce system overhead. Instead of going through the whole ready list to find the highest priority task to run, μC/OS-II uses the following simple algorithm:

```
    y  =  OSUnMapTbl[OSRdyGrp];
    x  =  OSUnMapTbl[OSRdyTbl[y]];
Priority  =  (y << 3) + x;
```

where: << means shit left

OSUnMapTbl returns the position of the first '1' starting from the least significant bit of the input byte.

Each has a Task Control Block (TCB) to store task parameters such as the task's stack pointer, priority, delay count and other parameters. Some of the services that are provided by μC/OS-II to manage tasks are create a task, delete a task, suspend and

resume a task. Also, to protect the system from priority inversion (discussed on next page); the operating system can change task priorities at run time.

μC/OS-II uses the periodic tick-timer interrupt to manage timed services such as the delay service. The interrupt must go through all tasks requiring timed services and decrement the delay field of each task. When the field reaches zero, the task is put in the ready list and the scheduler is invoked. The frequency of this tick-timer is very critical. Setting the frequency too high will introduce a higher system overhead, while setting it too low will reduce timing resolution and increase response jitter in the system. Furthermore, this service is non-deterministic as the time it consumes depends on the number of tasks in the system. Overcoming these problems associated with the tick-timer service is a central component of the SW/HW RTOS introduced later.

μC/OS-II provides some services for time management such as delaying a task for a number of ticks, resuming a delayed task, and getting and setting the current time. To maintain the multitasking feature of the system, a task should be implemented as a loop that includes a call to the delay service to sleep for a number of ticks. This looping approach to using timed services can cause a time drift problem, which is discussed in detail later.

In typical real-time system, tasks communicate information by either sharing access to variables or by sending messages. Sharing access to a variable can introduce a critical section which must be enforced as atomic code that allows only one task at a time to

access the variable. μC/OS-II provides synchronization services to protect critical sections, such as a mutex (mutual exclusion or binary semaphore) with functions to create a mutex, delete a mutex, pend (wait) on a mutex and post (release) a mutex. A counting semaphore is also provided to support task synchronization, with functions to create a semaphore, delete a semaphore, pend on a semaphore and post a semaphore. Figure 2-12 represents a mutex or semaphore by a key. Each task requiring access to the shared resource must first acquire the key using the pend operation, and when done, must release the key using the post operation. If the protected resource is busy, then the pending task is blocked until the resource is released. Furthermore, to synchronize a task with the occurrence of multiple events, μC/OS-II provides event flag services such as create an event flag, delete an event flag, pend on an event flag and post an event flag.

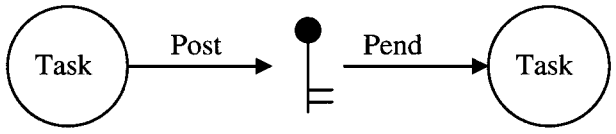


Figure 2-12: Task Synchronization [7]

The tasks synchronization around critical sections has priority-related issues. A **priority inversion** occurs when a lower priority task blocks a higher priority task from execution. Even worse, if a task with middle priority preempts the lower priority task while the higher priority task has been blocked, then the middle priority task will execute before the higher priority task. This is called **unbounded priority inversion** because the blocking time of the higher priority task is not bounded by the critical section execution

time, and potentially includes the execution time of all middle priority tasks. An unbounded priority inversion is depicted in Figure 2-13. Task1 has the highest priority, task2 has the middle priority and task3 has the lowest priority. At t1, task3 successfully locks the protection semaphore and continues executing until t2, where it gets preempted by task1. Task1 keeps running until it requires the semaphore to gain access to the shared resource at t3. Task1 fails to get the semaphore (because the resource is locked to task3) and the control returns to task3. At t4, task2 preempts task3 and continues executing until t5, where task3 resumes. At t6, task3 releases the semaphore and task1 resumes executing. The period between t3 and t6 there is an unbounded priority inversion where task2 and task3 block the higher priority task1 from executing.

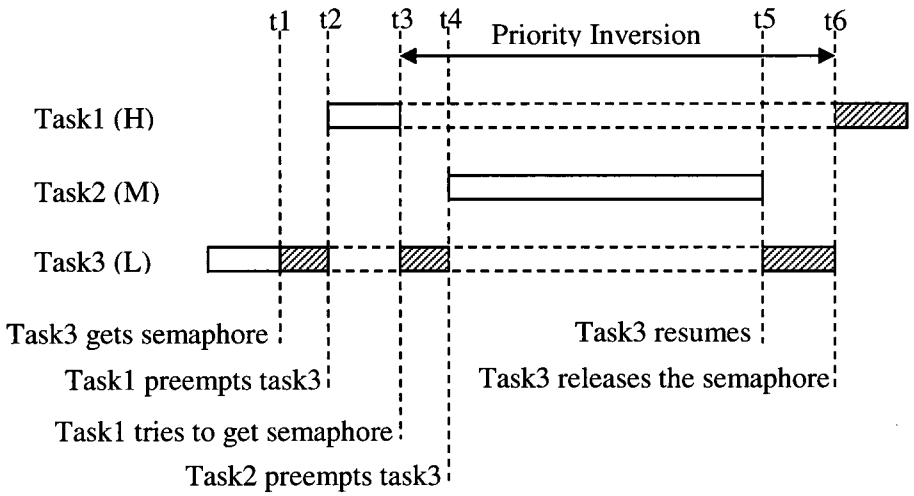


Figure 2-13: Unbounded Priority Inversion Example [7]

Priority inheritance is a potential solution to the unbounded priority inversion problem. This solution involves raising the priority of a lower priority task to that of the highest priority task that is requesting access to the semaphore. For example at t3 in

Figure 2-13 when task1 requests access to the semaphore, priority inheritance would raise the priority of task3 to the priority of task1. The elevated priority would prevent a task with a middle priority from executing until task3 releases the semaphore and task1 had completed. Priority inheritance solves the unbounded nature of the priority inversion, but the priority inversion still can occur.

The problem with priority inheritance is the potential for deadlock. **Deadlock** is the situation where a set of tasks are all blocking each other waiting for a resource that is granted to another task in the set. Figure 2-14 illustrates deadlock for a pair of tasks that both require access to two protected resources. One task gains access to resource1 and is preempted before it can gain access to resource2. Resource2 is granted to the preempted task, which then attempts to access resource1. At this point, the two tasks are deadlocked; they are both blocked waiting for the other to release a resource.

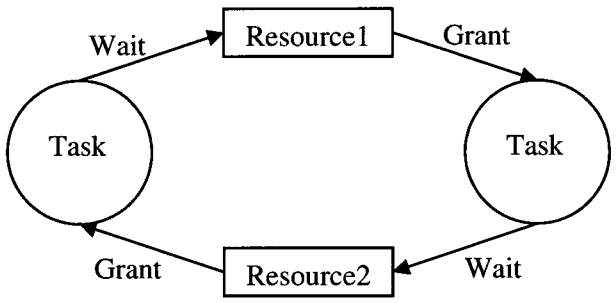


Figure 2-14: Deadlock (modified from [28])

The **priority ceiling protocol** is a deadlock-safe solution to the unbounded priority inversion problem. At the time that a lower priority task locks a resource; the protocol

raises the task's priority to the highest priority of all of the tasks that might require access to the locked resource. Furthermore, the protocol has more strict rules about granting access to the locks themselves. µC/OS-II supports priority inheritance, but does not implement the priority ceiling protocol.

For task communication using messages, µC/OS-II provides a message mailbox with functions to create a mailbox, delete a mailbox, pend on a message receive and post a message to a mailbox. To allow tasks to send one or more messages, µC/OS-II provides a message queue which is an array of mailboxes. As illustrated in Figure 2-15, a task uses a post function to send a message to a mailbox and a receiver task uses a pend function to wait for a message to be received in the mailbox.

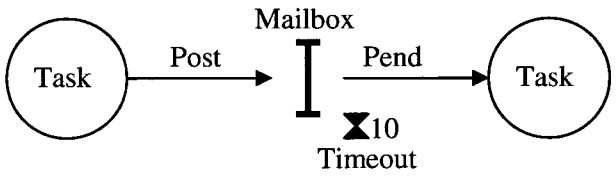


Figure 2-15: Task Communication [7]

µC/OS-II uses Event Control Blocks (ECB) to store information about task synchronization services (mutex and counting semaphore) and task communication services (message mailbox and message queue). µC/OS-II considers each of these services as an event and reserves an ECB for each event. The ECB consists of an event type, a count, a pointer and a waiting list. The event type is one of Mutex, semaphore, message mailbox or message queue. The count is the available resources in case of

semaphore (equal to zero for other event types). The pointer is pointing to a message or a message queue for message mailbox and message queue respectively (equal to null for other event types). The waiting list is used to store the tasks waiting for the event, and it has a similar structure to the ready list described earlier.

An unsuccessful pend on an event causes the pending task to be removed from the ready list and added to the waiting list of that event. Successful posting of an event removes the highest priority task from the waiting list of that event and adds it to the ready list. Pending on an event can include a timeout, specified as a number of ticks. If the timeout expires before the pending task is released, then a timeout error will be generated.

μC/OS-II prevents potential data corruption caused by concurrent access to shared resources by disabling interrupts while accessing TCBs, ECBs and the ready list. This can introduce a jitter in interrupt response time.

μC/OS-II provides many error codes to be used by an application to identify any RTOS runtime errors. For example, the error code 0x01 indicates a wrong event type was passed during access to a semaphore, mutex, message mailbox or message queue. When the timeout for a semaphore or a message mailbox expires, then the error code 0x0A is generated. The error code 0x28 is generated if the application attempts to create a task with a priority that is already in use, which is not allowed because each task's priority

must be unique. A complete list of error codes with brief descriptions of their meanings is given in Appendix A.

The Xilinx Platform Studio, version 9.1i (XPS 9.1i) provides an integrated development environment for creating a complete embedded system. It supports the creation of hardware specification flow (Microprocessor Hardware Specification MHS) and software specification flow (Microprocessor Software Specification MSS). It also provides a project management interface to create or edit source codes as well as a graphical presentation for connections between processors, peripherals and buses in the system. Another program called iMPACT can be used for programming the FPGA.

The AP1000 is a PCI platform FPGA development board from AMIRIX [12, 13]. It has many features which make it a good choice for SOC exploration, most notably that it includes the Xilinx Virtex-II Pro FPGA with an embedded PowerPC processor and a large gate capacity. As shown in Figure 2-16, the Xilinx Virtex-II Pro is the main feature of the board, with surrounding devices such as dual 64 MB DDR SDRAM, dual 2 MB SRAM, 16 MB program flash, 16 MB configuration flash, Xilinx SystemACE compact flash interface, 64-bit/66 MHz system PCI bus and several Ethernet controllers.

The baseline platform from AMIRIX is supported by Xilinx XPS 9.1i. It is the basic hardware infrastructure for SOC design inside the Virtex-II Pro device. Figure 2-17 depicts baseline platform architecture. The major components of the baseline are up to

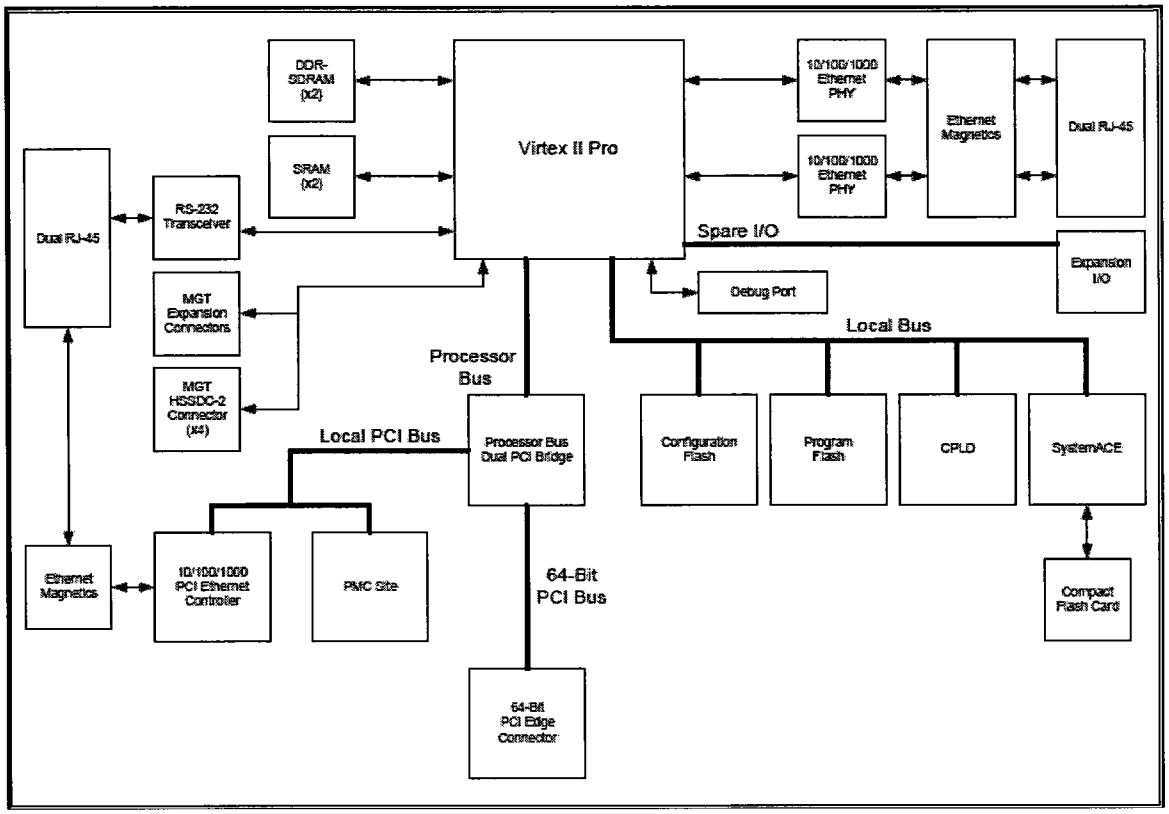


Figure 2-16: AP1000 Architecture [12]

two PowerPC 405 processors as hardware cores, IBM CoreConnect bus communication for SOC including the Processor Local Bus (PLB) to connect high performance resources and the On-chip Peripheral Bus (OPB) to connect lower speed resources. Developers can add custom hardware component to the baseline in the form of PLB User Logic and OPB User Logic. Micrium provides a BSP to access the OPB Interrupt Controller and PPC405's Programmable-Interval Timer (PIT).

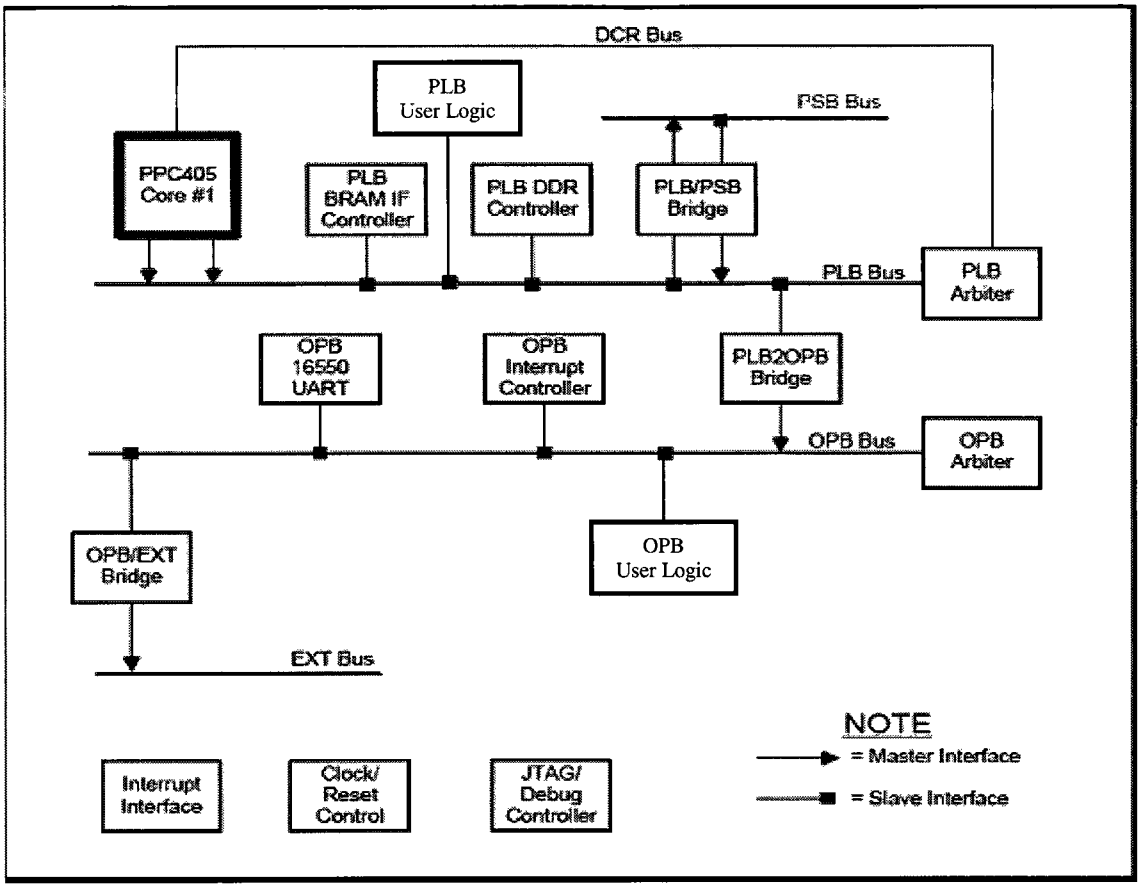


Figure 2-17: Baseline Platform Architecture (modified from [12])

3 State of the Art Research

There has been a substantial amount of previous work to improve the performance and the determinism of real-time systems. In some cases, more deterministic supporting hardware has been developed to support a software RTOS, while others partition the RTOS services between the hardware and the software. Additional work has developed analysis models and used simulation tools to test their models.

3.1 Tradeoffs on Real-Time Systems

Several researchers have studied the design constraints and computer architectural features that affect the predictability and determinism of real-time systems. According to Koopman [14], the need for high-performance embedded real-time systems leads designers to use higher performance microprocessors that provide more speed and processing power but have negative impacts on real-time design constraints such as cost, power, cooling, size, weight and reliability. Furthermore, these high- performance microprocessors often have architectural features that are very good for a general purpose computing environment but do not lend themselves to the determinism required for real-time systems. Features such as cache memory, variable length execution time instructions, prefetch queues, pipelines, dependence on compiler technology and virtual memory can introduce non-deterministic overheads that complicate the predictability of

response times. The precision timed (PRET) machines [15] attempt to address these problems by providing a high-performance processor combined with timing predictability. The machines have FPGA PRET cores extended by features such as pipelines, memory management and programming languages to support time precision.

3.2 Hardware Based Real-Time Systems

Partitioning solutions migrate some RTOS functionality to the hardware to improve the resulting performance and predictability.

Lindh and Stanischewski [16, 17] have implemented an integrated CPU with deterministic time behaviour running in parallel to a real-time kernel in hardware and call their system FASTCHART. Their CPU is a Reduced Instruction Set Computer RISC-CPU with a load/store architecture, while their real-time kernel contains the TCB's, the scheduler, the ready queue, the wait queue, the terminate queue and the system timing. FASTCHART supports up to 64 tasks with 8 priorities using the reduced state transitions shown in Figure 3-1.

Lindh continued the research on the hardware real-time kernel and developed a standalone unit which can be used with standard processors (as oppose to the FASTCHART processor) and migrated addition features to hardware such as rendezvous, interrupts and periodic tasks. He calls the resulting system FASTHARD [18]. It supports

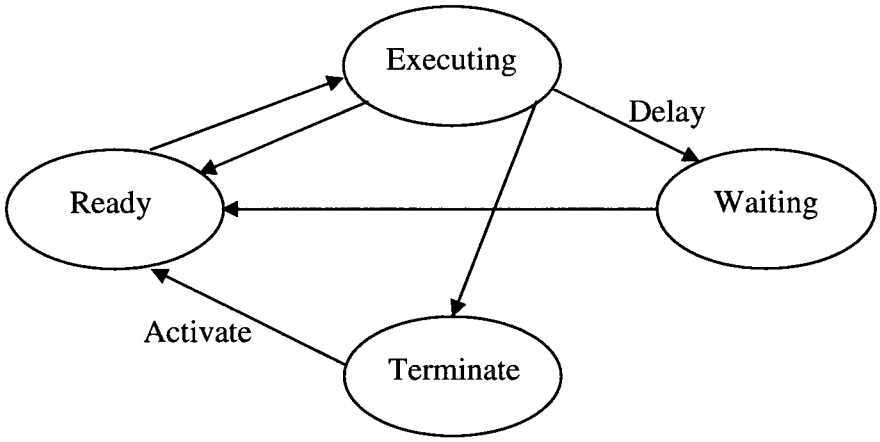


Figure 3-1: FASTCHART State Diagram [16]

256 tasks and 8 priorities with seven state transitions. Comparing with FASTCHART, Figure 3-2 shows the new three states which are periodic start, interrupt and rendezvous to support periodic tasks, interrupt and rendezvous handling, respectively. Context switching is kept in SW as a response to an interrupt from the HW kernel to the CPU.

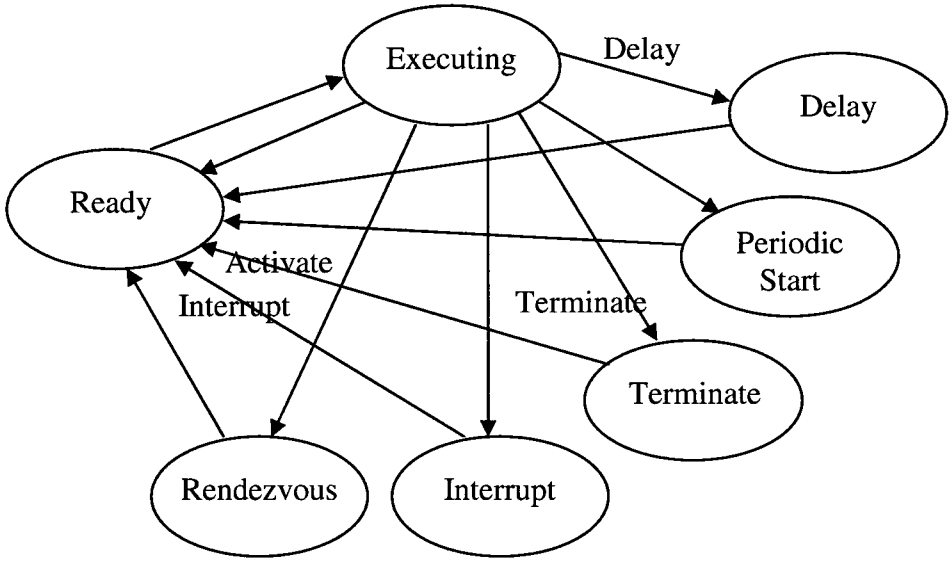


Figure 3-2: FASTHARD State Diagram [18]

Controlling multiple heterogeneous CPUs (up to three) is the main feature of the Real-Time Unit RTU94 [19], which is based on the FASTHARD. Some system services are added to RTU94 such as semaphores, event flags and watch dogs. RTU94 consists of one scheduler and one local queue in each CPU to manage local tasks, and a global queue which could be checked by each scheduler to manage global tasks that can be executed on any CPU.

The RTU is built in an Application Specific Integrated Circuit (ASIC). Researchers show the use of the RTU with single or multi-processors [20] along with an analysis of the timing behaviour of the RTU without application software. The RTU demonstration system is based on a Versa Modular Eurocard (VME) bus as shown in Figure 3-3. The

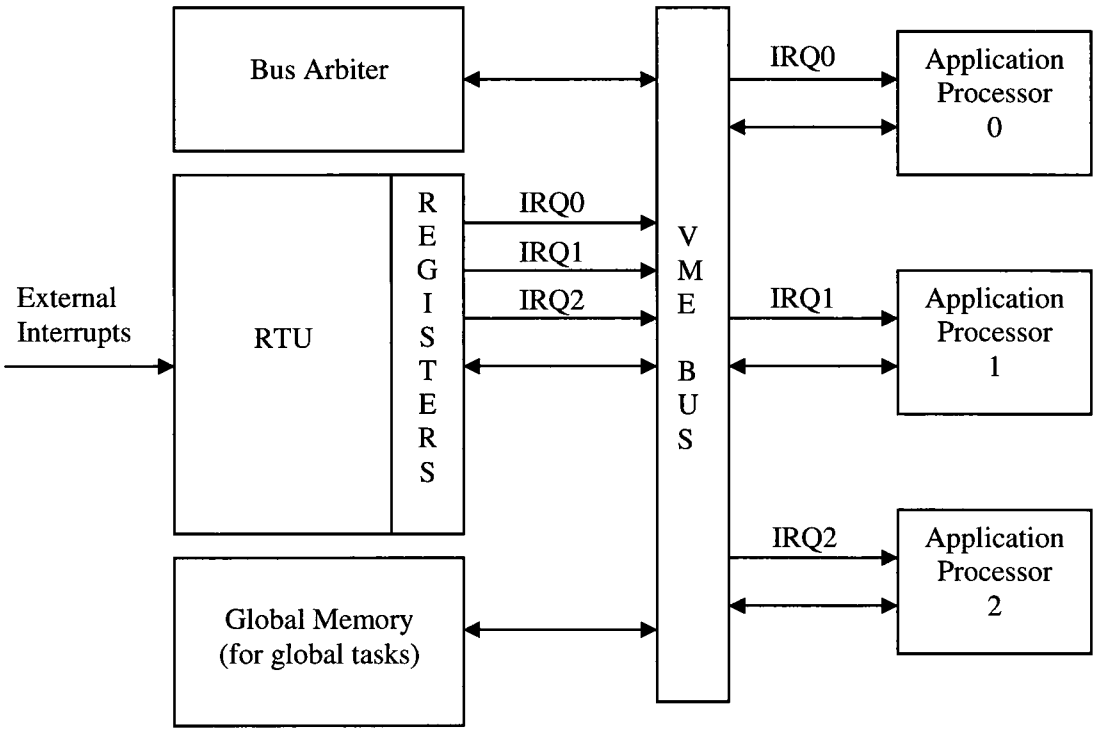


Figure 3-3: A Real-Time System based on RTU [20]

system includes a round-robin bus arbiter and a register interface between the RTU and the CPU(s). Once the RTU decides which task to run next, the RTU sends an interrupt to the target CPU to initiate the context switch.

The result of the timing analysis shows that an RTU based real-time system can increase the performance and determinism of the system [20]. The RTU has been developed to a commercial product named Sierra for uni-processor system [21].

An RTU with a configuration similar to Sierra (shown in Figure 3-4) can be used to speedup a software RTOS [22]. This RTU consists of a scheduler, an interrupt handler, a time manager, semaphores, a Generic Bus Interface (GBI) and a Technology Dependant Bus Interface (TDBI) such as the OPB discussed previously. Two experiments were conducted using the MicroBlaze CPU, one running μC/OS II and the other one running μC/OS II on top of the RTU. The results show that the RTU decreases most of the system call response times by a factor between 0.93 and 0.27.

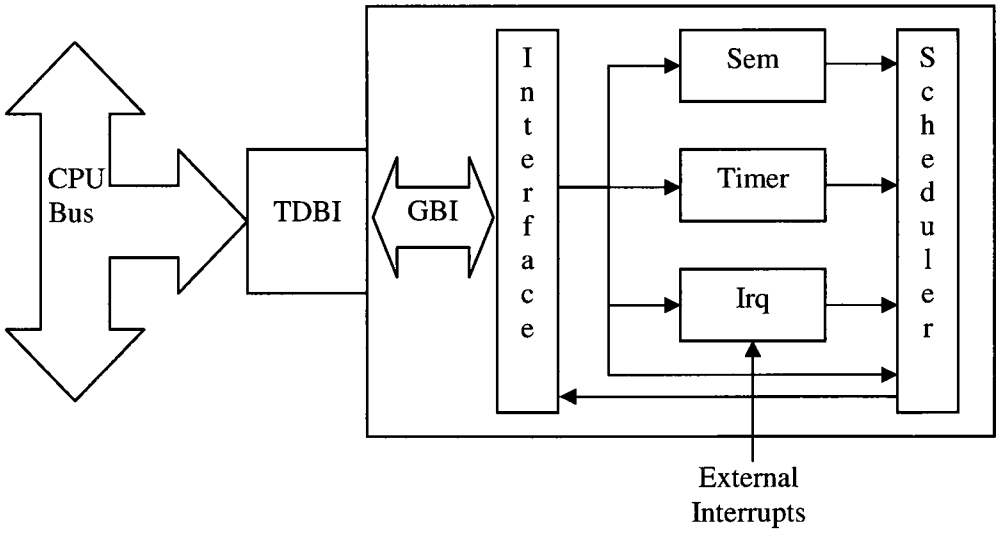


Figure 3-4: RTU Configuration Similar to Sierra [22]

Silicon OS [1] consists of a software kernel and a hardware kernel as shown in Figure 3-5. The software kernel or Micro kernel includes memory pool management, timer management, system management and the interface to the hardware kernel. The hardware kernel or Silicon TRON includes task management, semaphores for task synchronization, event flags, timers, interrupt management and a scheduler. The evaluation results for system calls on Silicon OS shows an improvement in execution time of 6 to 50 times over the pure software OS.

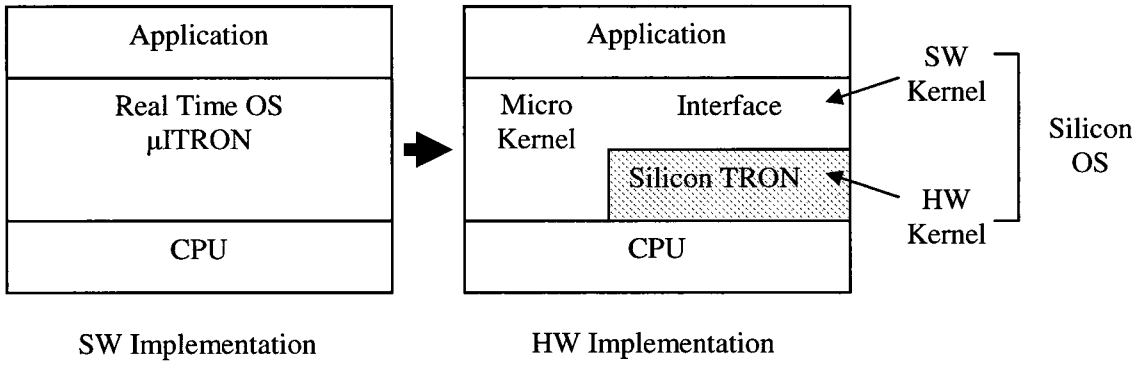


Figure 3-5: Silicon OS [1]

Another approach to improve the performance and predictability of a software RTOS is undertaken using dedicated hardware called the F-Timer [2]. The two main RTOS functions moved to the hardware in this approach are the external interrupt controller and the task scheduler. The microprocessor must send each task's information (task code, time of activation, absolute or relative time and the period for periodic tasks) to the F-Timer, and the scheduler then orders the tasks in time priority order and compares each task in the ordered list with the current time to find the next task to run. Finally, the

scheduler sends an interrupt to the microprocessor with that task's code to initiate the necessary context switch.

Different dynamic priority scheduling algorithms are implemented in a dynamic priority scheduler coprocessor [3] such as EDF and Enhanced Least Laxity First (ELLF). Further research [23] implemented the earliest deadline first scheduling algorithm in a software-only kernel called CoScheduler1 (CS1) on a 32-bit Nios CPU in an Altera FPGA. They then implemented the EDF scheduling algorithm in hardware/software kernel called CS2 by manually partitioning CS1 and moving the scheduling unit into an FPGA. The performance analysis showed an improvement in the kernel response time of about 55% in the case of the hardware/software scheduler coprocessor. A more flexible and configurable hardware scheduler [4] allows the user to configure the hardware scheduler at run time using a Graphical User Interface tool (GUI) to choose between priority based, RM and EDF algorithms.

The δ Framework [24, 25, 26] is an automatic generation tool for a partitioned hardware/software RTOS for a SOC based on the Atalanta multi-processor RTOS kernel. This tool provides a simple graphical user interface to configure the system under design. The user can choose the number of processors to use in the design from a list of supported processors (Advanced RISC Machine (ARM) and PPC). Also, the user can select:

- Which RTOS functions to use in software such as deadlock detection and memory management.

- Which functions to implement in hardware such as the system-on-chip lock cache, the system-on-chip deadlock detection unit and the system-on-chip dynamic memory management unit.
- Which functions for Inter-Process Communications (IPCs) such as semaphore, event, mailbox, queue and mutex.

More work on system-on-chip lock cache was conducted [27] to integrate the priority inheritance mechanism into a SOC multi-processor environment. Also, a deadlock avoidance algorithm and its hardware implementation (deadlock avoidance unit) were integrated [26, 28].

The Real-Time Task Manager (RTM) [5] provides hardware support for a software RTOS by implementing three time consuming RTOS functionalities: task scheduling, time management and event management. A performance test is done using RTM and a simulator of the Texas Instruments TMS320C6201 with μC/OS-II and NOS (custom non-preemptive) RTOSs. The results showed reductions in the RTOS overhead by up to 90% and in the maximum processor response time by up to 81%.

A quantitative analysis of hardware support for real-time operating systems [29] is based on the generic model for fixed priority scheduling derived by Katcher and Strosnider [30]. In the study, the real-time Mach OS is used in two different systems: (i) a software-only version of Mach with generic RISC hardware features and without any OS hardware support; and (ii) a hardware-assisted version of Mach with an Intel 80960XA

microprocessor which provides hardware support for scheduling, tasks and semaphores. The study shows an improvement in operating system performance by an average factor of 3 in the case of partitioned OS hardware support.

3.3 Analysis and Comparison

Many studies are done to analyze and compare different ideas and approaches to improve real-time system performance.

The benefits and disadvantages of FASTCHART are explained by Stanischewski [31]. The greatest benefits are the determinism of the resulting system and eliminating the need to write the operating system code (since it is already integrated). On the other hand, the greatest disadvantages of FASTCHART are the fixed scheduler algorithm used in the system and the fixed number of resources such as the numbers of tasks and semaphores which limit the use of FASTCHART to average systems.

A performance comparison [32] contrasts the RTU hardware RTOS, Atalanta software RTOS and Atalanta RTOS interfaced with hardware support system-on-chip Lock Cache. These three RTOSs are configured using an upgraded framework to support the RTU and the comparison is based on performance in a simulation of a database application example. When compared to the pure software RTOS, the average simulation results showed 36% overall speedup in case of the RTU while the overall speedup was 19% in case of the Atalanta RTOS with hardware support.

A review [33] of four techniques to speedup a software RTOS included using a co-processor, porting some of the software RTOS functions into hardware, building the whole RTOS in hardware, and developing a customized instruction set to support the software RTOS [34]. A more specific comparison conducted on the scheduler [35] explored three implementations: a software scheduler with the application tasks on the same processor (Software Real-Time Scheduler SoRTS), a software scheduler on a co-processor which runs in parallel to the processor that executes the application tasks (Co-processor Software Real-Time Scheduler Co-SoRTS), and using a hardware scheduler in parallel to the processor that executes the application tasks (Hardware Real-Time Scheduler HaRTS). The results of this work showed that HaRTS has the best performance but it is a more complex and expensive implementation compared to Co-SoRTS and SoRTS. The results did not include a quantitative performance analysis.

4 The Thesis

Timing parameters are very important in hard real time systems. As a result, research into system overhead and system determinism is valuable in the context of HRT. Previous research either assumed that there is no system overhead, or tried to compensate for overhead by implementing some of the system services in hardware to speed up the system. Some of the research tried to analyze the system overhead but there were few specific numbers to show the relation between the system overhead and system parameters such as number of tasks in the system and the clock frequency of the periodic tick-timer. These short comings build the motivation for this thesis to address overhead, jitter and non-determinism of the system, and to extend the theory to include system overhead in the analysis.

4.1 Statement of Thesis

For a system of periodic tasks with a rate monotonic priority assignment, it is possible to improve the standard task response time prediction model by including parameters to model the overheads associated with context switching and managing the periodic tick-timer, A partitioned software/hardware RTOS implementation can improve response time by reducing some of the software RTOS overheads, and can improve the determinism of the system behaviour.

4.2 Scope

Predicting task response time requires accounting for runtime behaviours specific to the target platform. In support of this thesis, two systems are developed based on the AP1000 target board. One system uses a pure SW RTOS and the other uses a partitioned SW/HW RTOS. For practicality in the limited timeframe of the research, the partitioned RTOS implements only a representative subset of the SW RTOS services.

The research is limited to using μC/OS-II as the pure SW RTOS on a PowerPC 405 embedded in a Xilinx Virtex-II Pro FPGA. For the partitioned RTOS, μC/OS-II is modified and the task management services (task creation, task scheduling, and manipulating TCBs and the ready list), time management service (periodic tick-timer), task synchronization services (semaphore management and manipulation of ECBs) and a task communication service (message mailbox management) are ported to hardware. The SW RTOS system is described in Chapter 5, and the partitioned SW/HW RTOS system is described in Chapter 6.

Profiling experiments on both systems are presented in Chapter 7. The profiles includes performance timing, system and task parameters including number of tasks in the system, number of context switch, task period, task priority, pending status of each task, and waiting list of each event. The profiles are used in Chapter 8 to extend the standard response time model for systems using the rate monotonic priority assignment strategy.

A performance comparison of the SW RTOS versus partitioned RTOS is given in Chapter 9. Due to practical time limitations, only a representative subset of the service calls is compared. The compared service calls include RTOS initialization, task creation, and successful and unsuccessful pend on a semaphore or message mailbox.

Throughout the research, periodic tasks are the only supported task type and the periodic timing interrupt is the only interrupt considered. Non-periodic tasks and more general interrupt concerns are important issues that must be addressed in future work.

4.3 Contributions to Knowledge

The main contributions to knowledge by the thesis are the improved response time prediction model and the improved determinism by implementing RTOS timing services in hardware.

As a secondary contribution, the implementation work has uncovered a suspected hardware problem in the AP1000 target board.

5 System using a Pure SW RTOS

A pure SW RTOS lacks determinism due to jitter and system overhead. Quantifying these problems requires measuring their characteristic behaviours for a realistic system. The system presented in this chapter uses the μC/OS-II as the pure SW RTOS running on top of the PPC405 processor on the AP1000 board. The design and architecture of the system are presented first, followed by the system implementation and three potential timing problems associated with that implementation. The system is used in Chapter 7 to gather performance measurements, and the measurements are then used in Chapter 8 to construct a more accurate model for predicting task response time when using the pure SW RTOS.

There are several reasons for choosing μC/OS-II as the software RTOS in this research:

1- Free to use for educational purposes.

2- Availability of source code (with the purchase of "MicroC/OS-II The Real-Time Kernel" book or can be downloaded from Micrium web site after registering).

3- Easy to understand with a short learning curve.

4- Small footprint with a maximum 24 Kbytes which can fit easily in the Block RAM (BRAM) of the Virtex-II Pro internal memory.

5- Portability for many processor architectures and availability of a PowerPC 405 port

5.1 Design and Architecture

The pure SW RTOS system is designed for the AP1000 target board. The μC/OS-II is ported for the PPC405 which is embedded in the Xilinx Virtex-II Pro FPGA. The general architecture of this implementation is shown in Figure 5-1. The PPC405 is the heart of the design along with the IBM CoreConnect PLB and OPB buses. The PLB has a bus speed of 80 Mhz and is used to connect the high performance devices (BRAM and Timers). The Block RAM is 64 Kbytes of memory which holds the SW RTOS and the application program. The lower performance UART is connected to the OPB with a bus speed of 40 Mhz. The UART block is used as the input/output of the system to provide a user interface.

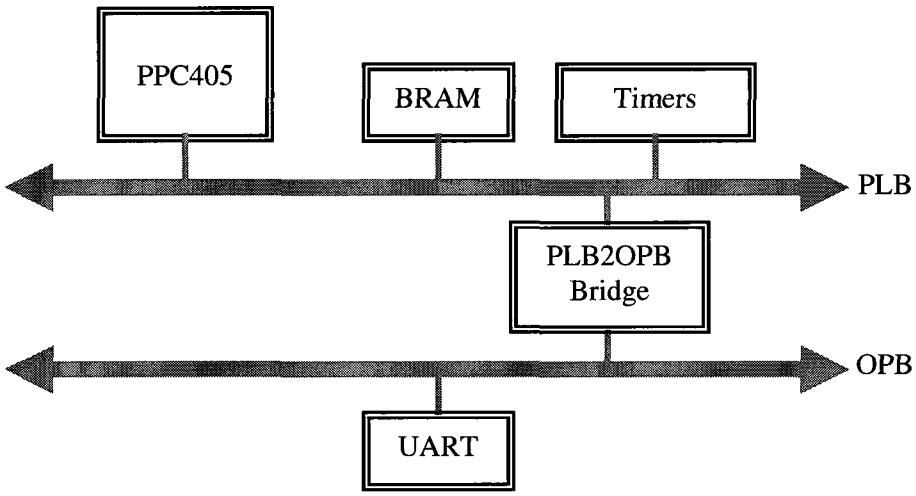


Figure 5-1: System using SW RTOS Architecture

The timers block contains high resolution hardware timers that are used to gather the time measurements used later in performance comparisons and system overhead analysis.

The timers block consists of a stop-watch timer and a continuous timer, both with a clock rate of 80 Mhz (details about the software commands to use the timers are given in Appendix B).

The periodic tick used by the SW RTOS is generated using the PPC405's PIT and the frequency is controlled by a value loaded to the PIT. The frequency control value is calculated as follows:

$$PPC405\_CLK / OS\_TICK\_PER\_SEC$$

where: PPC405_CLK = 240 Mhz for the AP1000, OS_TICK_PER_SEC is the desired frequency and is user defined.

To collect data for RTOS timing analysis for different tick frequencies, it is necessary to configure the PIT clock frequency (OS_TICKS_PER_SEC), and that is possible from the software setting menu using the XPS development tool to modify the bsp.h header file.

5.2 Implementation

System using μC/OS-II must implement periodic tasks using the delay service. This implementation suffers from three potential timing problems which are discussed in this section. The system uses the PIT to generate the periodic tick. This implementation suffers from response-time jitter and a jitter related to the delay service call. The use of the delay service also introduces the potential for time drift.

To understand the response-time jitter, consider the scenarios shown in Figure 5-2. Note that the timing values have been contrived to illustrate the jitter problem, and are not actually times associated with the µC/OS-II. Suppose that the OS tick execution time is 2.5 msec and the actual task execution time being measured is 15 msec. The timer output (measured by a hardware timer) shows the response time including both the task execution time plus the OS tick execution time. The two scenarios in the figure show that the response time may depend on when the task starts executing relative to the OS tick. In Figure 5-2.A, the response time is equal to the actual task execution time plus 1 OS tick execution time which gives a total of 17.5 msec. In Figure 5-2.B, the response time is equal to the actual task execution time plus 2 OS tick execution times which gives a total of 20 msec. The maximum response-time jitter is equal to 1 OS tick execution time which is 2.5 msec in this example.

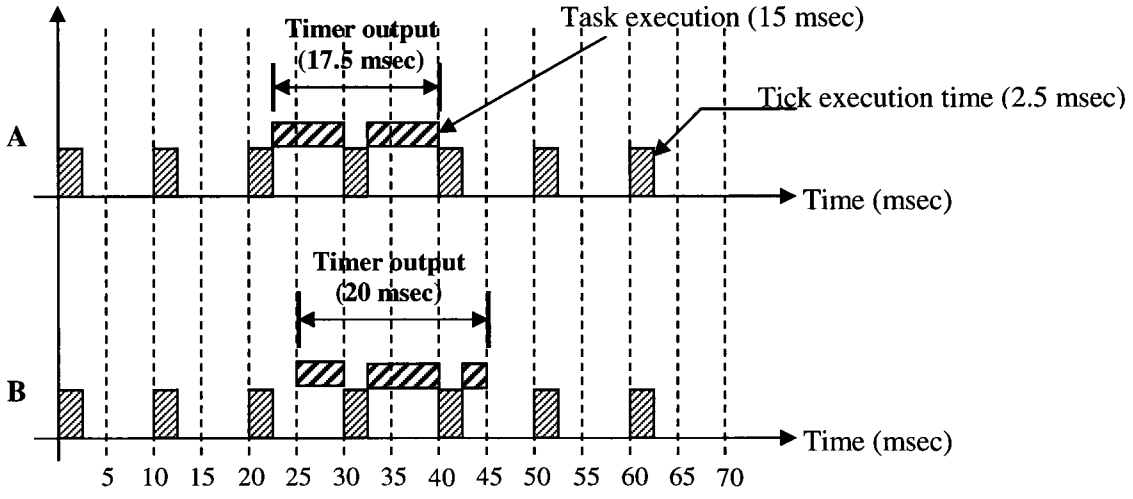


Figure 5-2: Response-Time Jitter

Jitter related to the delay service call is shown in Figure 5-3. In the two scenarios, the delay service is called request a 2 tick delay; however, the delay time depends on when the delay function is called relative to the OS tick. In Figure 5-3.A, the delay time is approximately 15 msec, while the delay in Figure 5-3.B is approximately 17.5 msec. The maximum delay jitter is equal to the OS tick period minus the OS tick execution time. (7.5 msec in this example).

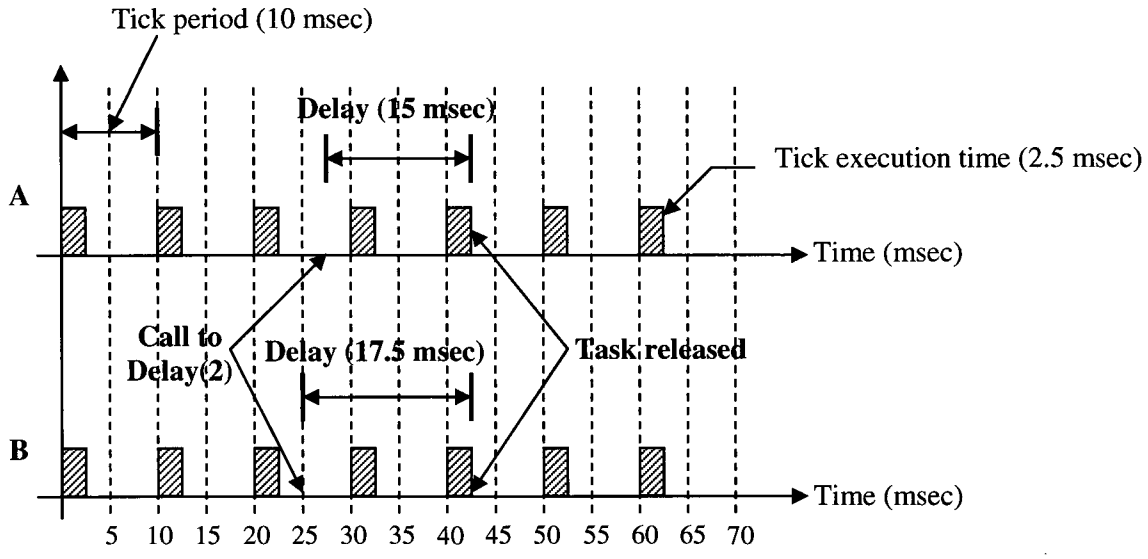


Figure 5-3: Jitter in Delay Service Call

Periodic tasks are common in real-time systems, but μC/OS-II does not support the creation of periodic tasks. Implementing a periodic task in μC/OS-II requires a looping task that uses the delay service between releases. The skeleton of periodic task implemented using this model is shown in Figure 5-4. The first time the task runs it performs local task initialization (i.e. Loop = 0) before entering the infinite loop. Each iteration of the loop represents one periodic release of the task. At the end of each loop

iteration, the task reads the timer and calculates the delay until the next release. The loop counter is needed to adjust the period with each loop correctly. The skeleton code is actually an oversimplification, because it does not account for loop counter or timer roll-over.

```
Task( )
{
Loop = 0;                                        /* task initialization*/
    “
While (1)
  {
    “              /* application code to be executed each periodic release */
    “
    Time = Read timer;                           /* read time */
    Dly_time = (++Loop * Period) - Time;         /*  calculate delay */
    Delay(Dly_time);                             /* delay to next activation */
  }
}
```

Figure 5-4: Periodic Task Implementation in µC/OS-II

The implementation of periodic tasks using the delay service faces a potential problem which is known in industry as “Time Drift”. This problem occurs if the task is preempted in the critical region between reading the time and calling the delay service. Once the time has been read, the amount of delay is fixed. This delay will not account for any preemption that occurs before the delay service is invoked. Protecting the critical region by disabling the interrupt before reading the time will not work because the interrupt gets enabled during timer access. The time drift may vary and depends on the length of preemption. Figure 5-5.A shows the desired behaviour, while Figure 5-5.B shows a time drift due to preemption. The desired period is 5 sec ($Period_A$); however the 1 sec preemption in Figure 5-5.B results in 6 sec period ($Period_B$) between the two

releases of the task. The drift anomaly only occurs when the critical section is preempted; however all subsequent releases are offset by the drift.

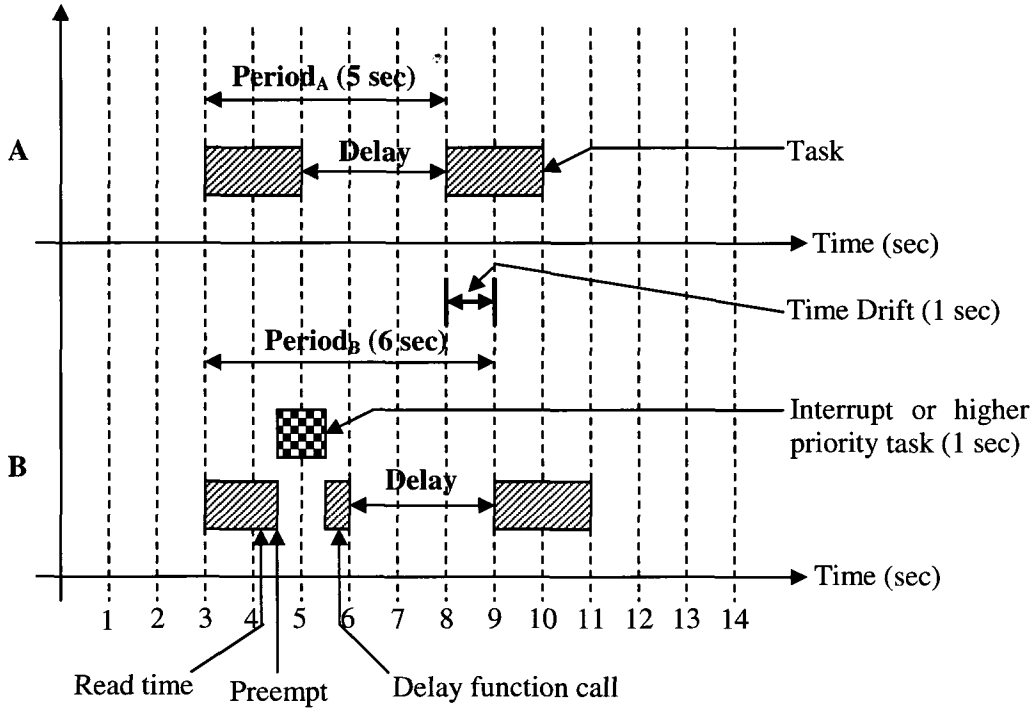


Figure 5-5: Time Drift Problem

6 System using SW/HW RTOS

A partitioned SW/HW RTOS is proposed to improve the real-time system performance and prediction and to overcome the problems associated with the system using pure SW RTOS (i.e. jitter, time drift and overhead). The partitioned SW/HW RTOS includes a modified μC/OS-II RTOS running on top of the PPC405 in parallel to hardware support (HWRTOS) which implements some of the μC/OS-II services. The design of the partitioned SW/HW RTOS is presented first, and the implementation details including state machine and SW/HW interface are then discussed. Only representative details are given here. A complete description of the error codes, the SW/HW interface and the function flow charts are available in appendices A, B and C, respectively.

6.1 Design and Architecture

The design of the partitioned SW/HW RTOS moves some of the μC/OS-II functionalities into the FPGA hardware to create the HWRTOS. The system design is discussed first, followed by the design of the modified μC/OS-II and the HWRTOS. The resulting architecture in Figure 6-1 is similar to the design of the system using pure SW RTOS. The HWRTOS block replaces the Timers block (and incorporates the Timers block functionality), and the Interrupt from the HWRTOS block to the PPC405 is introduced.

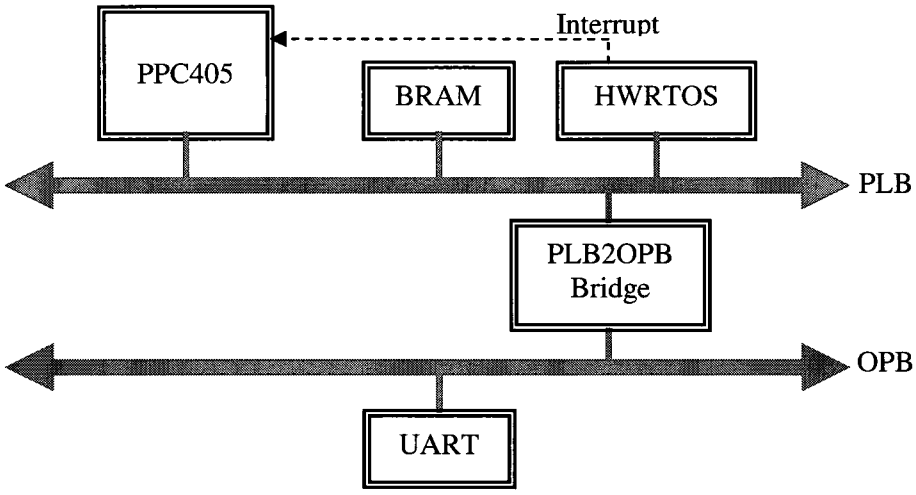


Figure 6-1: System using SW/HW RTOS Architecture

The modified μC/OS-II is stored in the BRAM along with the application program. The HWRTOS is a user logic block or a user Intellectual Property (IP) connected to the PLB bus and communicating through a set of registers with the modified SW RTOS running on PPC405. The Virtex-II Pro FPGA does not support direct access to the processor registers. As a result, the context switch functionality is kept in software. If there is a need to do a context switch, the HWRTOS block sends an interrupt to the PPC405 to do the context switch in software. The HWRTOS is implemented in VHDL as a PLB User Logic block (recall Figure 2-17). The HWRTOS includes the following RTOS services: task management, tick management, task synchronization and communication and test management. Each of these services is discussed in the HWRTOS design section. The partitioned SW/HW RTOS addresses the timing problems inherent to the pure SW RTOS. The pure SW RTOS suffers from response-time jitter due

to the periodic tick and the time drift problem as shown in figures 5-2 and 5-5 respectively. The HWRTOS implements the periodic tick-timer in hardware which eliminates response-time jitter. Furthermore, the HWRTOS supports periodic tasks which are not supported in µC/OS-II. As a result, the time drift problem does not occur in the partitioned SW/HW RTOS.

6.1.1 Modified µC/OS-II Design

The µC/OS-II is modified by moving some of its data structures and services to the HWRTOS. As a result, most of the µC /OS-II initialization functions are no longer needed in the modified version. The TCB, the ECB and the ready list data structures in addition to the periodic tick, scheduler, task, semaphore and message mailbox managers are moved to the HWRTOS. Mutual exclusion, event flag, message queue and memory managers and task's stack are left in µC /OS-II. If there is a need to access the hardware timers or any of the moved services, then the modified µC/OS-II must communicate with the HWRTOS.

The modified µC/OS-II uses a set of read/write (RD/WR) interface registers to communicate with the HWRTOS to access the implemented timers and services in the HWRTOS. A request command is sent to the HWRTOS through a WR register. Following the request, a RD register must be read to check the results and the status of the request. The issuing of a request followed by the subsequent read command must be

implemented as atomic code, and that is done by disabling interrupts around them. An alternate approach that does not require disabling interrupts is presented in Chapter 10.

A task's stack stores all the required CPU registers for context switch. A task's stack is left in the modified μC/OS-II and linked to its task through task priority. The running task priority and the highest priority task are the required information to do a context switch and they are stored in the HWRTOS. If there is a need to do a context switch due to the periodic tick-timer, a context switch interrupt is sent to the PPC405.

Since the context switch interrupt is asynchronous, the HWRTOS stores the context switch information in a temporary register to avoid the potential that the information is overwritten by responses of other requests generated from the modified μC/OS-II. The context switch ISR in the modified RTOS should send a request (Context Switch Information Request) to the HWRTOS in response to the interrupt. The request asks the HWRTOS to put the context switch information into the RD register then the ISR should read the RD register and do the context switch. The returned information from this request includes the priority of the current task and the new task that should take control of the processor. An alternate approach to avoid the overwrite issue without the need for a temporary register is suggested later in Chapter 10.

6.1.2 HWRTOS Design

A design-level discussion of the internal blocks of the HWRTOS is given here. The HWRTOS models the μC/OS-II design for most of the implemented services. Those design decisions unique to the HWRTOS are explicitly declared.

Figure 6-2 shows the block diagram of the HWRTOS. The HWRTOS is connected to the PLB bus in the Xilinx baseline, and the 80 Mhz PLBCLK is used as the system clock of the HWRTOS. The interrupt (context switch interrupt) is an output signal connected to the PPC405 critical interrupt line. Eight 64-bit registers are connected to the PLB bus. There are four input (WR) and four output (RD) registers. The arrows among the blocks represent buses. There is a bus connecting the management blocks with the registers to get the request parameters and put back the request results. Commands to the timers and tick management blocks do not have any request parameters, and the blocks only put results in the registers. There is another bus connecting the scheduler, list manipulation and data structures to the management blocks. A third bus sends enable signals generated from the controller to all management blocks. The enable signals activate the appropriate block when needed.

Each of the blocks in the HWRTOS depicted in Figure 6-2 are described below.

Registers (0-7): The modified SW RTOS communicates with the HWRTOS using eight 64-bit registers. Reg0 to Reg3 are used as write (WR) registers to send command requests to the HWRTOS, and Reg4 to Reg7 are used as read (RD) registers to read

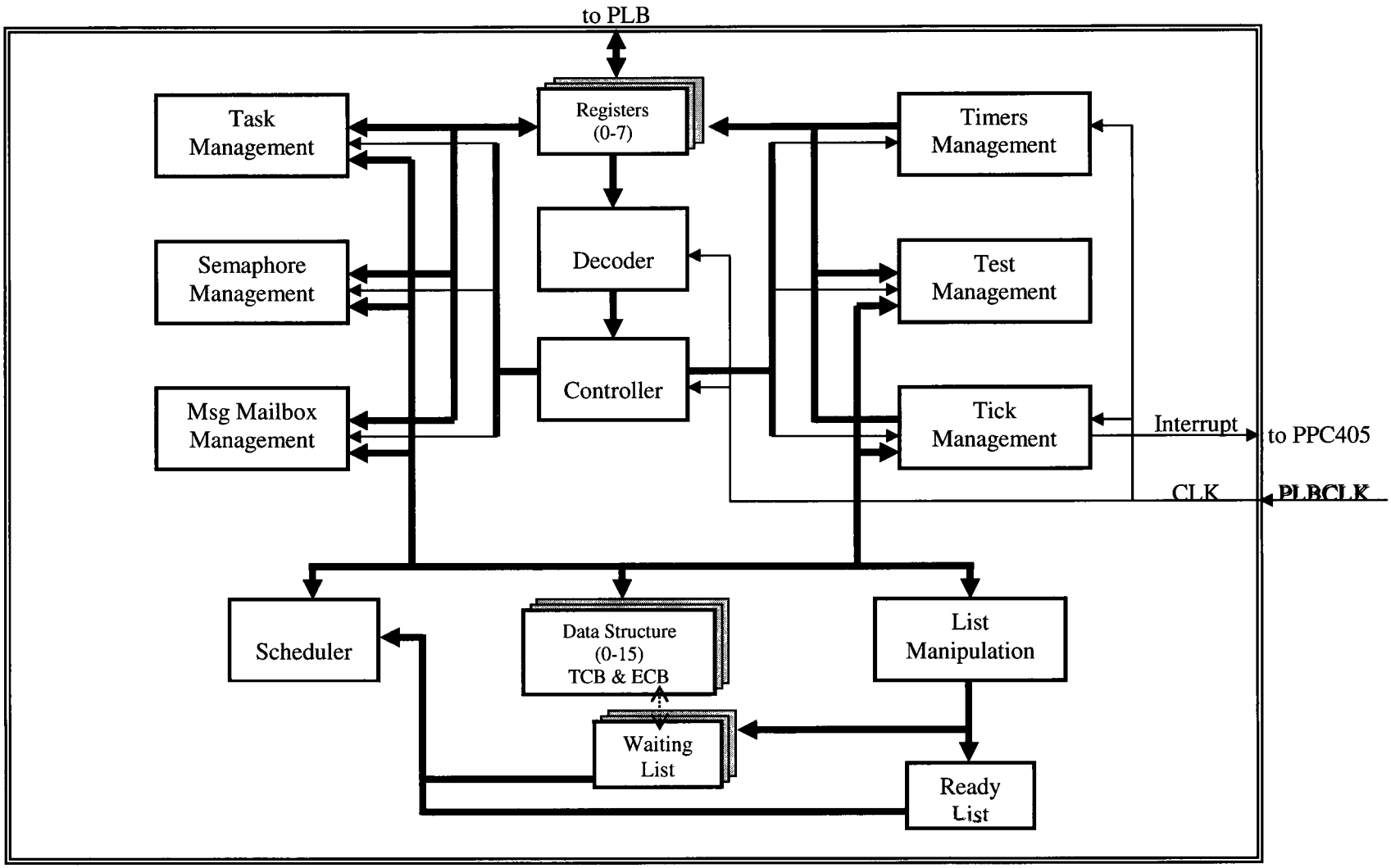


Figure 6-2: HWRTOS Block Diagram

results from the HWRTOS. In the current design, only Reg0, Reg4, Reg5 and Reg6 are used and the other registers are provided for possible future use. The management blocks access the request information from the WR register (Reg0) and put the results into the RD register (Reg4). Reg5 and Reg6 are used only by the test management block. These registers are not present in the μC/OS-II because there is no hardware part in the μC/OS-II. Further details about bit encodings in the register are given in Section 6.2.2.

Decoder: The least significant byte of the write register Reg0 (bits 7 to 0 in the application code) is used to send command request to the HWRTOS. The content of Reg0 is latched and the block decodes these commands and generates the necessary signals to the controller. Reading the results of the request by the application automatically resets the latched request data. The decoder is not available in the μC/OS-II because there is no external command to decode.

Data Structure: The two main data structures are the TCB and the ECB. The implemented fields for the TCB and the ECB are similar to the μC/OS-II. Other fields used by μC/OS-II are not relevant to this research and have not been implemented. A TCB holds a task's information, including the following fields:

- The unique task priority which is used as TCBs index.
- An indication to show if the TCB is free or reserved by another task.
- Delay and Dec_delay fields to hold the task's period and the current value of the decremented period with each tick respectively.

- Pre-calculated values x, y, bitx and bity which are used in list manipulation.
- ECB number that the task is pending on.
- A pointer to the message received from a mailbox.
- A context switch counter which is logged for profiling purpose.
- The task's ASCII name (limited to 1 character).

The ECBs array index is used as the ECB ID. An ECB holds a semaphore or message mailbox information, including the following fields:

- An event type (semaphore or message mailbox).
- An indication to show if the ECB is free or reserved by another event.
- A pointer to a message in the case of a mailbox.
- A counter for the available resources in case of a semaphore.
- A waiting list to indicate all tasks that are waiting for the current event.
- The event's ASCII name (limited to 1 character).

Controller: The controller is the heart of the HWRTOS and it organizes and synchronizes the work of all of the management blocks. The controller provides important synchronization between external asynchronous requests from the application (through registers) and the internal requests from the tick management block. The synchronization protects the shared TCB and ready list resources from possible corruption during concurrent requests. The synchronization is implemented using a state machine which enables the appropriate block according to the request. The state machine is discussed in detail in Section 6.2.1.

In addition to request synchronization, the controller is responsible for initializing and starting the HWRTOS in response to external requests from the application. The initialization request is used to set the frequency of the internal tick-timer while the start command is used to ask the HWRTOS to start the multitasking.

List Manipulation: The list manipulation block manipulates the ready and the waiting lists. The ready list contains the priorities of all the tasks ready to run, and a waiting list contains the priorities of all the tasks waiting for an event. This block also supports adding and removing tasks in these lists similarly to the μC/OS-II.

Scheduler: The HWRTOS models the μC/OS-II priority-based scheduler (lowest number means highest priority). The scheduler is invoked from many of the blocks in the HWRTOS. It can find the highest priority task in both the ready and waiting lists since the lists are implemented with the same logical structure.

Task Management: The task management block is responsible for task creation. The maximum number of tasks is currently set to 16 for test purposes, and includes the idle task. This number can be increased easily to the logic limits in the chip. The process of task creation reserves one TCB for each task. Under the HWRTOS controller permission, TCB fields are updated according to the input parameters that are passed in task creation request.

Each task must have a unique priority which is used both as the task ID and to reference the task's TCB. At the end of the task creation process, the task is added to the

ready list. The task creation request must include the task period in number of ticks. The period must be set to zero if the task is not periodic. Delaying a non-periodic task for a specific number of ticks is not implemented and is left for future work as discussed in Chapter 10.

Tick Management: The implementation of the tick-timer is done in the tick management block. The tick-timer implements the timer interrupt functionality of the μC/OS-II. It generates periodic ticks to manage periodic tasks and timeouts. With each tick, the tick manager loops through all of the tasks and checks the period field (Dec_delay) in the TCB of each task which is initialized during task creation. If the period is not equal to zero, then the tick manager decrements the period by one. When the period reaches zero, the field is reset to its period value (Delay) and the associated task is added to the ready list. After checking all tasks, the scheduler is invoked. If a context switch should be performed, then the tick manager generates a context switch interrupt to the PPC405. The defined constant (HWRTOS_TICKS_PER_SEC) in the bsp.h header file is used to configure the tick-timer frequency.

Semaphore and Message Mailbox Management: The semaphore management block services implement semaphore creation, pending on a semaphore and posting to a semaphore. The message mailbox management block services implement mailbox creation, pending on a message to be received in a mailbox and posting a message to a mailbox. The HWRTOS models the μC/OS-II ECB, so each semaphore and message

mailbox reserves an ECB. Requesting any of the semaphore or message mailbox services requires accessing the ECB, and the access is controlled by the HWRTOS controller.

The maximum number of events (semaphores and message mailboxes) supported by the HWRTOS is currently limited to 16 for testing purposes. This number could easily be increased to the logic limits in the chip.

Timers Management: The hardware timers that are used for analysis and performance measurements are implemented in the timer management block. There are two types of timers, stop-watch and continuous. Reading the count value of the stop-watch timer stops the timer and resets it automatically. Reading the count value of the continuous timer is done dynamically and does not influence the timer's operation. The clock input of the timers is tied to the 80 Mhz system clock which results in a time resolution of 12.5 ns per timer increment. These timers are unique to the HWRTOS.

Test Management: The test management block is unique to the HWRTOS and is very useful during development and debugging. This block delivers application-selected information. This information includes the TCB of a selected task, the ECB of a selected event and system parameters. Accessing these shared resources is managed by the HWRTOS controller.

The HWRTOS models some of the μC/OS-II error codes. These error codes are generated by different blocks in the HWRTOS and are sent to the modified μC/OS-II through the interface registers and eventually are sent to the application.

6.2 Implementation

The HWRTOS behaviour is directed by the controller block in Figure 6-2. The block is implemented as a state machine that must account for asynchronous requests from the application and from the tick management block. This section presents a detailed description of the controller state machine and the register-based communication used in the partitioned SW/HW RTOS. Representative examples of the register interface and function flow charts are also presented, while complete details are given in the appendices B and C.

6.2.1 HWRTOS State Machine

The HWRTOS controller manages the operations of the HWRTOS using the state machine shown in Figure 6-3. The state machine issues signals that trigger the behaviour of the management blocks.

The state machine has 4 states and 9 transitions. Each transition is labeled to indicate the requests that trigger the transition. Requests can originate from various sources. The EXT _REQ is triggered at the arrival of any external request in Reg0. When the internal periodic tick-timer outputs the periodic tick, the function of looping through the delay field of each task is activated and that is referred to INT_REQ. The INT_EXT_REQ represents the case where an external request and an internal tick occur at the same time.

The RST is generated by the occurrence of a register read operation or the end of the internal tick function.

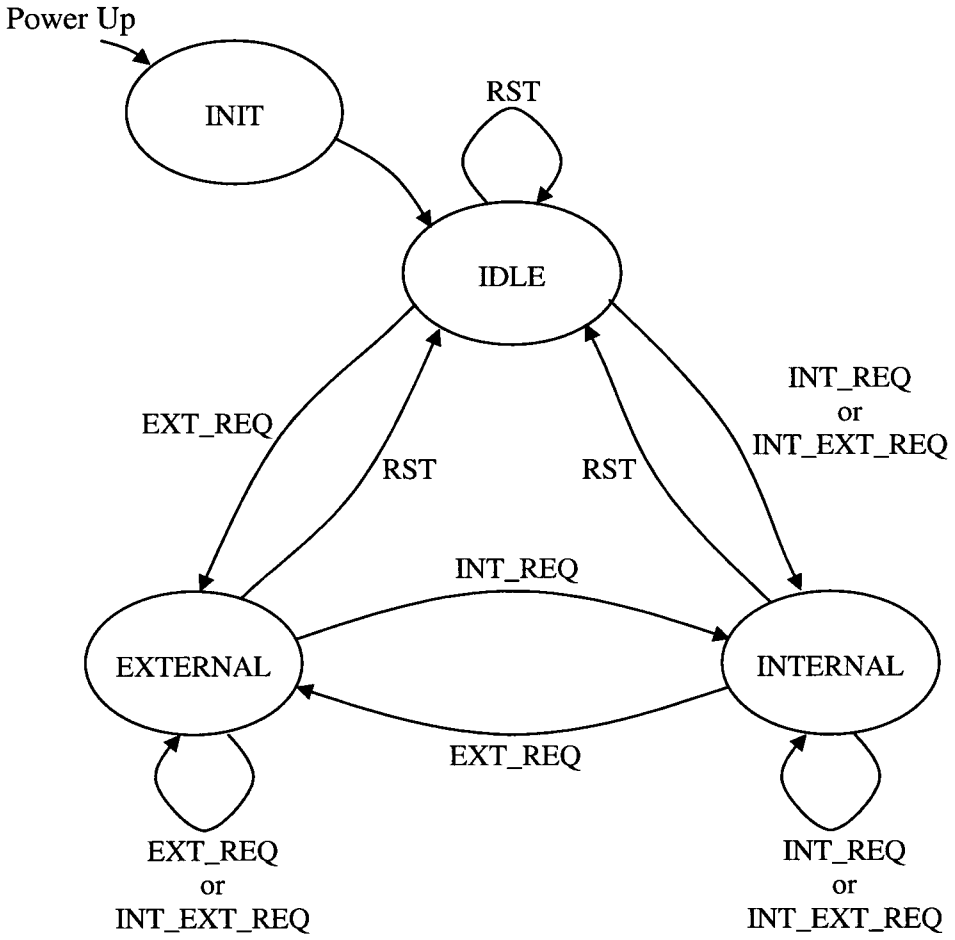


Figure 6-3: HWRTOS State Machine

The system powers up in the INIT state where all system data structures are initialized, and then goes to the IDLE state where the controller is ready to receive requests. While in the IDLE state, EXT_REQ causes the controller to make a transition to the EXTERNAL state while INT_REQ or INT_EXT_REQ causes a transition to the INTERNAL state. Timing functionality gets priority over external requests in the IDLE

state. The RST command always forces the controller to the IDLE state. While in the INTERNAL state, the internal tick-timer gains safe access to the shared resources. Any INT_REQ or INT_EXT_REQ keeps the controller in the INTERNAL state. For the INT_EXT_REQ, the external request gets processed after internal function processing. EXT_REQ causes a transition to the EXTERNAL state. While in the EXTERNAL state, the external request gains safe access to the shared resources. EXT_REQ or INT_EXT_REQ keeps the controller in the EXTERNAL state while INT_REQ causes a transition to the INTERNAL state.

As shown in the pseudo-HDL in Figure 6-4, this is a Mealy state machine as the output depends on the current state and the state inputs. The machine is synchronized to the rising edge of the system clock. At each system clock, the current state decides the

```
If rising_edge'clk then
    Case state is
        When INTERNAL =>
            If      CMD = RST then
                    state = IDLE
                    clear request
            Elsif   CMD = INT_REQ or INT_EXT_REQ then
                    state = INTERNAL
                    enable the appropriate block based on request type
            Elsif   CMD = EXT_REQ then
                    state = EXTERNAL
                    enable the appropriate block based on request type
        When EXTERNAL =>
            If      CMD = RST then
                    state = IDLE
                    "
                    "
```

Figure 6-4: HWRTOS Controller State Machine Sample Code

next state according to the input command and provides safe access to the shared resources by enabling the appropriate block. RST forces the state machine to the IDLE state and either clears the INT_REQ if the end of internal tick function occurred or clears the latched EXT_REQ if the read command occurred.

6.2.2 SW/HW Interface

The communication between the modified SW RTOS and the HWRTOS is done through a set of 64-bit registers. The registers can be accessed using the PLB bus. Figure 6-5 shows these registers along with the direction of the data flow.

The write registers (Reg0..Reg3) are used to send external requests to the HWRTOS. Each request consists of a command field as the least significant byte, followed by argument(s) field(s) as needed. The number of registers used per request depends on the type of the command and the number and size of the argument(s). In the SW/HW RTOS, only Reg0 is used.

The read registers (Reg4..Reg7) are used to read the status of the requests as well as to feed back results. The least significant byte of each status register is set equal to "0x01" if the last external request was successful, otherwise it is set equal to "0x00". The rest of the 64-bits are used for results appropriate to the completed command.

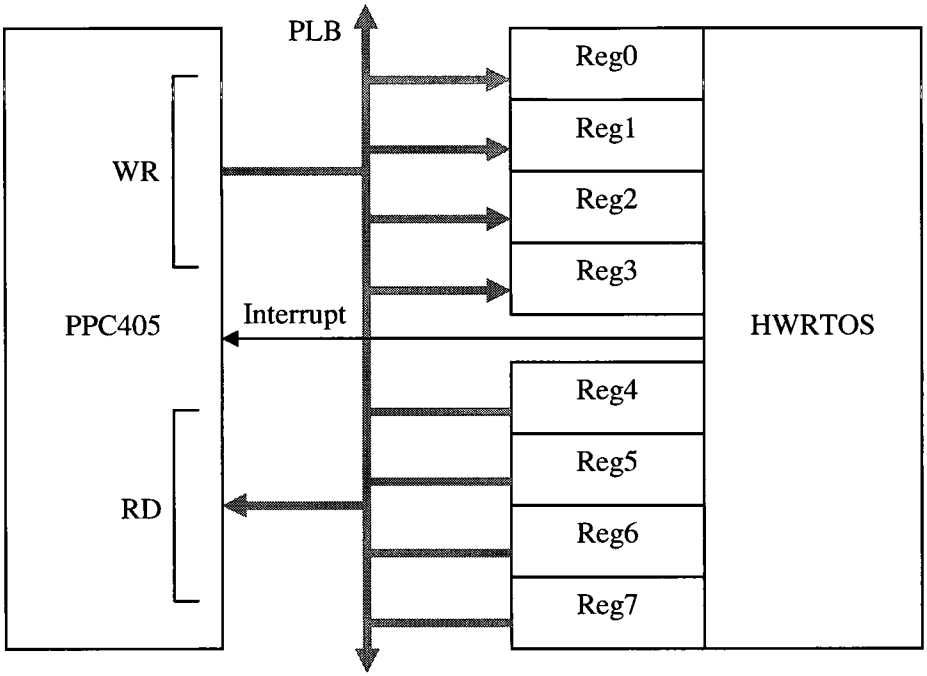


Figure 6-5: SW/HW Register Interface

Figure 6-6 shows an application model that uses some of service calls provided by the SW/HW RTOS. OSTaskCreatePeriodic( ) and OSSemPost( ) services are discussed in detail from the application, modified SW RTOS and HWRTOS perspectives.

The application model consists of the main function and the created task function. The initialization call (OSInit) in the main function sets the tick-timer frequency. The request to create a semaphore (OSSemCreate) has one argument which is the number of available resources for the semaphore. The returned value is the semaphore ID reserved by the OS. The arguments of creating a periodic task (OSTaskCreatePeriodic) include the task priority, task period and task name. At the end of the main function, there is a call (OSStart) to start OS multitasking.

```
Main( )
{
     “
OSInit( );
     “
Sema4_1 = OSSemCreate(argument);
     “
Error = OSTaskCreatePeriodic(arguments);
     “
OSStart( );
}

Task( )
{
     “
While(1)
     {
          “
     Err = OSSemPost(Sema4_1);
          “
     OSTaskDone( );
     }
}
```

Figure 6-6: SW/HW RTOS Application Model

The created task is an infinite loop. The task code includes posting a semaphore (OSSemPost) using the semaphore ID (Sema4_1) returned during the earlier semaphore creation request. As the last act in the loop, the task relinquishes the CPU by calling OSTaskDone service. The SW/HW RTOS then blocks the task until it is released again at the beginning of its next period.

SW/HW RTOS supports periodic tasks in hardware. As a result, the time drift problem is solved. The application must call the OSTaskCreatePeriodic( ) function and pass the task period as one of the arguments (or pass zero if task is not periodic). The

modified SW RTOS checks for interrupt nesting error, initializes task's stack, and then accesses the HWRTOS as shown in Figure 6-7. Interrupts are disabled around the HWRTOS access to provide atomic access. If there is no error in the returned create result and a context switch is required, then a context switch is done in software. Otherwise, the error status returns to the application.

```
OSTaskCreatePeriodic(arguments)
{
If (interrupt nesting)
     Return (create a task from within an ISR error);
Initialize task's stack;
Disable interrupt;                    /* lock critical protection */
HWRTOS_Request_Reg0(Board_Addr, create task arguments);
HWRTOS_Read_Reg4(Board_Addr, result);
If (bit 9 in result = 1)              /* successfully reserves priority */
     {
     If (context switch is required)
          {
          Do context switch;
          }
     Enable interrupt;                /* unlock critical protection */
     Return (no error);
     }
Enable interrupt;                     /* unlock critical protection */
Return (priority already exists error);
}
```

Figure 6-7: Create Periodic Task Model in Modified SW RTOS

The create task request is sent to the HWRTOS using Reg0. Table 6-1 shows the fields, shaded fields are not used (zeros).

Arguments					Command	Reg0
[63..56]	[55..48]	[47..32]	[31..16]	[15..8]	[7..0]	
	Name	Period	Id	Prio	0x05	

Table 6-1: Create Task Request

Command 0x05: Request to create a task with the provided arguments.

Arguments:

Prio: Unique task priority from 0 to 15 (0 is the highest priority).

Id: Task id (could be anything, it is logged in the TCB, but not used by the HWRTOS).

Period: Task's period in number of ticks (period = 0 if task is not periodic).

Name: One ASCII character that represents the task's name.

The create task request is followed by a read operation to check for the results of that request. The modified SW RTOS uses Reg4 for all read operations (except for test information read). Table 6-2 shows the fields returned following a task create request.

Data					**Status**	**Reg4**
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Ctx_Sw	Free	Stat	

Table 6-2: Create Task Info Read

Status: The status of the create task request.

Data:

Free: If this field is equal to "0x00", then the requested task priority was not reserved previously by any task and is now reserved for the newly created task. If the requested priority is already reserved by another task, then the "priority already exists (0x28)" error will be returned in the field.

Ctx_Sw: If the task creation requires the system to make a context switch, then this field is equal to "0x01". Otherwise, this field is equal to "0x00".

Prio_Cur: Priority of the current task that needs to be switched in the case that a context switch is required (i.e. when Ctx_Sw = "0x01").

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor in the case of Ctx_Sw = "0x01".

The flowchart for the creation of a task in the HWRTOS is shown in Figure 6-8. The HWRTOS uses task priority to index the TCBs, so it starts by checking whether the provided priority is used or not. If the priority is already reserved, then this information is

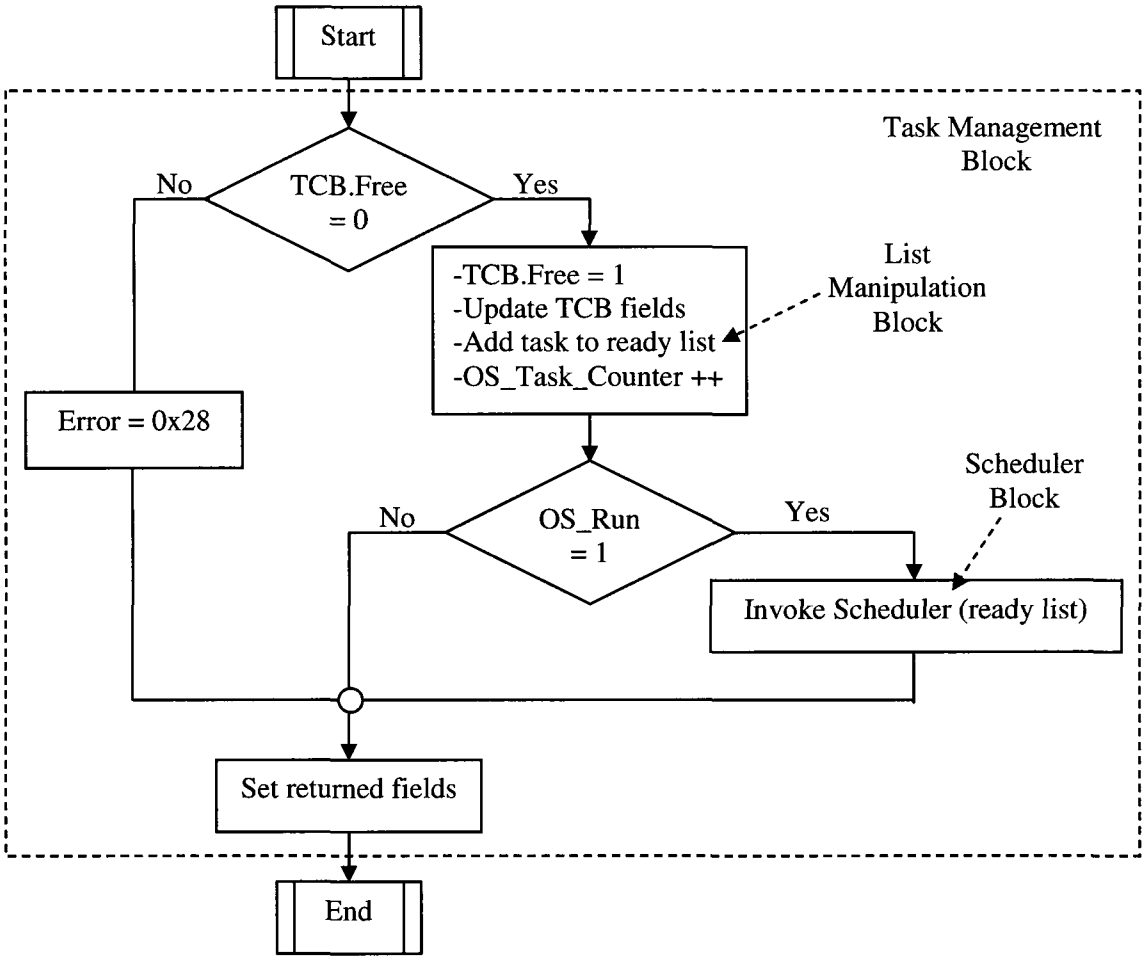


Figure 6-8: HWRTOS Create Task Flowchart

returned to the modified SW RTOS. If the priority is free, then the HWRTOS reserves the TCB of that priority, updates the TCB fields, adds the task to the ready list and if the OS is running, invokes the scheduler. (Calling OSStart() to start the OS multitasking changes the OS status field in the HWRTOS from 0 to 1).

Creating a semaphore returns the reserved ECB ID (recall Sema4_1 in Figure 6-6). The ECB ID is used in any access to the created semaphore. Figure 6-9 shows posting to a semaphore. The modified SW RTOS starts by checking for an ECB ID error, and then accesses the HWRTOS. A context switch is implemented if it is required by the returned information from the HWRTOS or an error is returned to the application.

```
OSSemPost(ECB)
{
If (ECB error)
     Return(invalid ECB error);
Disable interrupt;                  /* lock critical protection */
HWRTOS_Request_Reg0(Board_addr, post a semaphore argument);
HWRTOS_Read_Reg4(Board_addr, result);
If (context switch is required)
          {
          Do context switch;
          Enable interrupt;         /* unlock critical protection */
          Return (no error);
          }
Enable interrupt;                   /* unlock critical protection */
Return (error code from result);    /* event type error or semaphore count overflow */
}
```

Figure 6-9: Post Semaphore Model in Modified SW RTOS

The modified SW RTOS sends the post semaphore request to the HWRTOS using Reg0. Table 6-3 shows the fields.

Arguments		Command	Reg0
[63..16]	[15..8]	[7..0]	
	Event	0x0A	

Table 6-3: Semaphore Post Request

Command 0x0A: Request to signal a semaphore.

Arguments:

Event: This is the semaphore ID to signal.

The modified SW RTOS uses Reg4 for read operation that follows the post semaphore request. Table 6-4 shows the fields returned following a post semaphore request.

Data					Status	Reg4
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Ctx_Sw	Err	Stat	

Table 6-4: Semaphore Post Info Read

Status: The status of the semaphore post request.

Data:

Err: This is an error code. When it is equal to "0x00", there is no error. A value of "0x01" indicates that the provided semaphore number is not for a valid semaphore. A value of "0x32"indicates that the post has resulted in semaphore count overflow.

Ctx_Sw, Prio_Cur and Prio_H: The values as discussed previously.

The process of posting a semaphore in the HWRTOS is shown in Figure 6-10. The HWRTOS starts by checking whether the provided ECB is for a valid semaphore. If the ECB is not a valid semaphore, then "0x01" error is returned to the modified SW RTOS. If the type is correct, then the HWRTOS checks the waiting list for the provided semaphore. An empty waiting list causes the HWRTOS to increment the semaphore count, or send "0x32" error to indicate an overflow. If there is at least one task waiting at the semaphore, then the HWRTOS finds the highest priority waiting task, adds it to the ready list, removes it from the waiting list and invokes the scheduler.

The major differences between the pure SW RTOS and the partitioned SW/HW RTOS are summarized here:

- The SW/HW RTOS supports periodic tasks, while the pure SW RTOS does not. Supporting periodic tasks directly in hardware solves the time drift problem since the delay management can not be preempted.
- The pure SW RTOS has a timer interrupt, while the SW/HW RTOS implements the tick-timer in hardware. This eliminates response-time jitter due to the timer interrupt.
- The context switch in both systems is done in software, but the SW RTOS does that as a response to a system call, while the SW/HW RTOS does it as a response to the external context switch interrupt or the returned result of the read operation.

- The SW RTOS disables interrupts around the access to the shared resources such as the TCB and the ready list which is not the case in the SW/HW RTOS because these resources are implemented in hardware. Also, the SW/HW RTOS disables interrupts to access the HWRTOS, while there is no HWRTOS in the pure SW RTOS. Disabling interrupts is not desirable because it increases the interrupt latency. Disabling interrupts in the SW/HW RTOS is only during HWRTOS access which is less than in the pure SW RTOS where disabling interrupts is done with every timer interrupt and every access to shared resources.

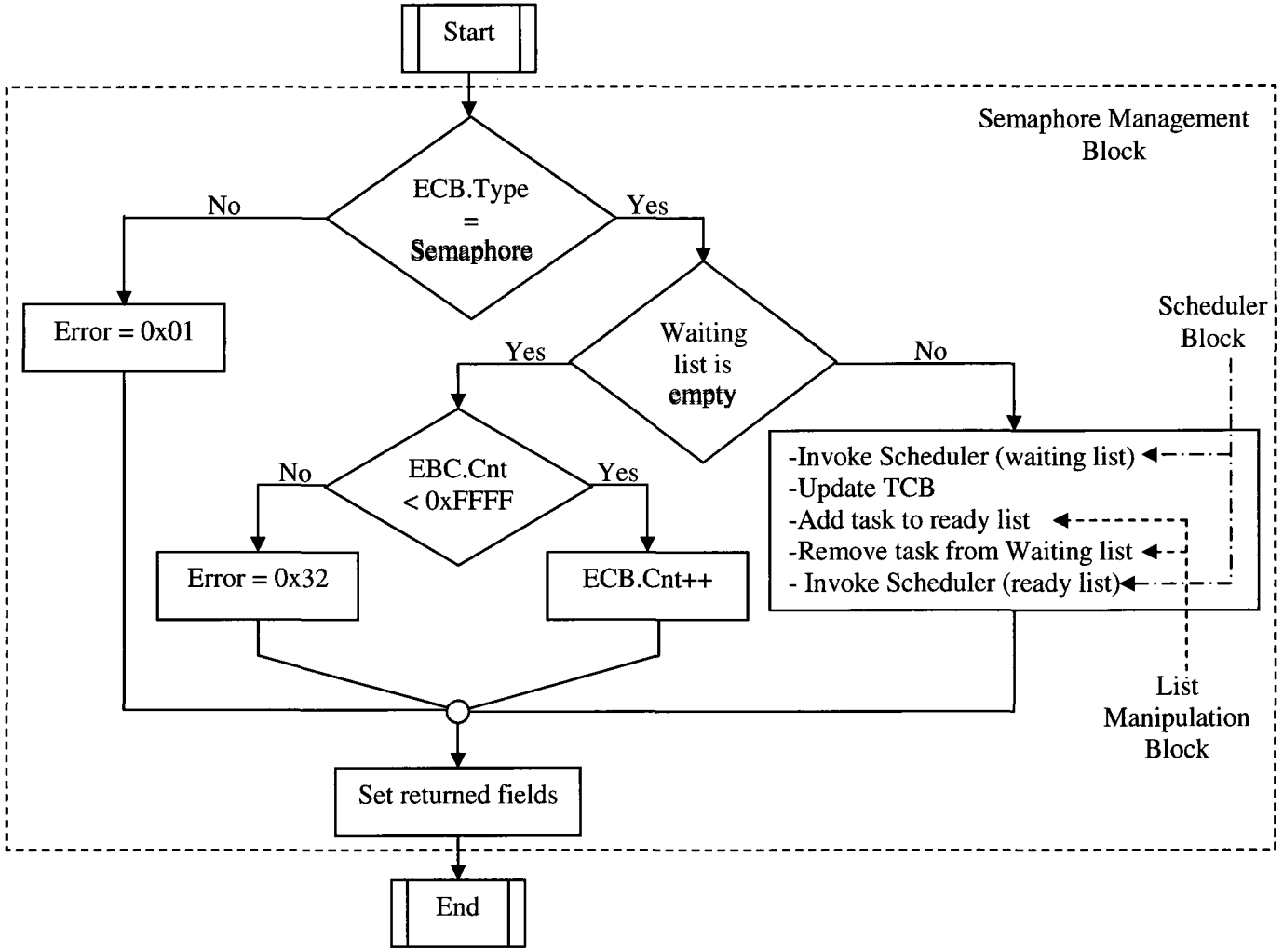


Figure 6-10: HWRTOS Post a Semaphore Flowchart

7 Experiments and Results

A quantitative performance comparison of the pure SW RTOS and the partitioned SW/HW RTOS requires three experiments to measure the overheads and behaviours of both systems. Throughout the experiments, the cache is disabled to eliminate its negative effect on system determinism, and compiler optimization is turned off to avoid its unpredictable output. The experiments and a summary of the results are presented in this chapter. The complete collected data and comparison graphs are presented in Appendix D and E. The data obtained in these experiments is used in Chapter 8 to derive more accurate response time prediction equations.

To help in understanding the results, note that on the target board, the PPC405 performs read or write operation with BRAM devices on the PLB bus in about 75 ns, and the read or write operation with the HWRTOS registers takes about 62.5 ns. Also, caution must be exercised in all interpretations of the results, as there could be up to a 1% error in time measurement due to a target system anomaly which is discussed in detail in Chapter 10.

The first experiment measures the overhead due to the periodic tick-timer for the pure SW RTOS. A hardware timer is used to measure the execution time of a simple "for loop" inside a task. The hardware timer is triggered before the loop, and the timer count is read after the loop as shown in Figure 7-1. The body of the loop executes a simple

addition to consume time. The measured execution time is the actual execution time of the loop iterations plus the execution time of the periodic ticks that occur during the loop iterations. The test starts by creating two tasks in the main program. The higher priority task delays itself for the maximum tick numbers (65535) to make sure that it does not preempt the lower priority task (which includes the "for loop") during the measurement. The periodic ticks decrement the delay of the higher priority task with each tick. After starting OS multitasking, the higher priority task starts and then delays itself, at which point the lower priority task starts running and the timer measurements are collected.

```
Higher_priority_task()
{
     "
While(1)
     {
          "
     Delay(max_delay);
     }
}

Lower_priority_task()
{
     "
Overhead = 4000;
     "
While(1)
     {
          "
     Start timer;
     For (i=0; i<1000000; i++) {
          Test = i + overhead;          /* waste some time */
          }
     Read timer;
          "
     }
}
```

Figure 7-1: "For Loop" Code

The test is extended for different tick-timer frequencies and different numbers of created tasks, but the lowest priority task always includes the "for loop" and higher priority tasks delay themselves to the maximum tick numbers. Increasing the tick-timer frequency causes more interrupts to occur during the execution of the "for loop" which in turn results in higher tick-timer overhead. Similarly, increasing the number of created tasks causes each periodic tick to require more time to perform delay management, which causes further overhead.

The same measurements are repeated for the partitioned SW/HW RTOS where the tick-timer is implemented in hardware. In this test, the measured execution time is the time consumed by the processor executing the "for loop" code and there is no tick-timer overhead involved because the periodic tick function is implemented in hardware.

Each configuration runs about 50 times. For the pure SW RTOS, differences in the results of each run are at most the execution time of one tick-timer. For the partitioned SW/HW RTOS, the results of each run are the same due to implementing tick-timer in hardware.

The overhead due to the periodic tick-timer is calculated by subtracting the measured execution time in the SW/HW RTOS from the measured execution time in the SW RTOS. As might be expected, the data shows that this overhead increases proportionally with the number of tasks and the number of ticks per second (tick-timer frequency) as shown in Figure 7-2. Note that the overhead axis has a logarithmic scale. The overhead of

the tick-timer with frequency of 100 tick/sec and 2 created tasks is about 2.1 msec and increases to about 3.6 msec for the same frequency and 10 created tasks. The overhead increases to about 92.7 msec for a frequency of 4000 tick/sec and 2 created tasks. Implementing the tick-timer in hardware eliminates this overhead.

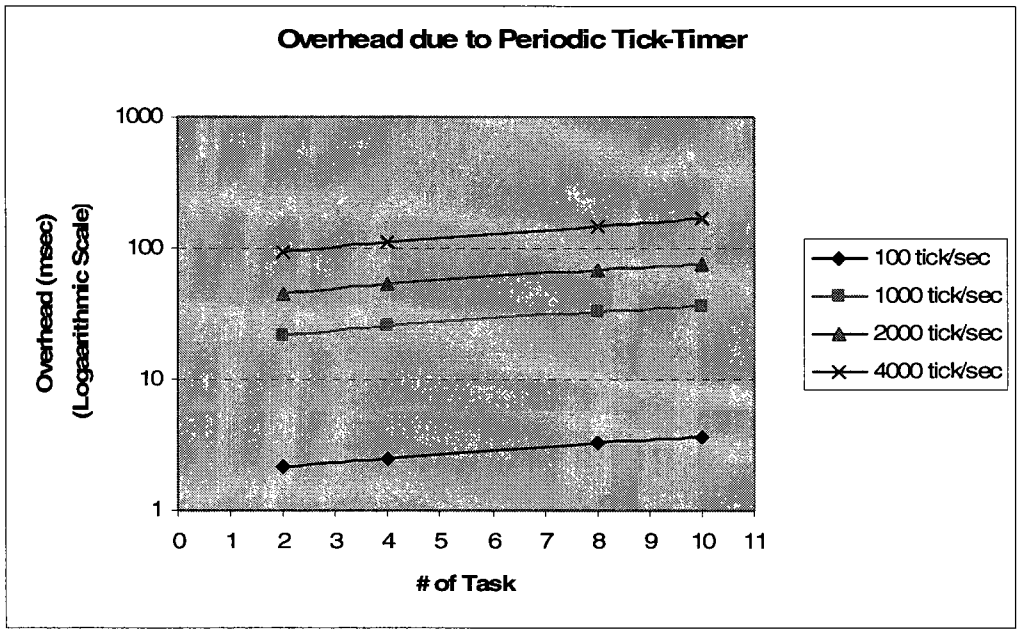


Figure 7-2: Overhead due to Periodic Tick-Timer

MATLAB is used to solve for an equation which describes the relationship between the overhead due to the tick-timer interrupt, the number of tasks (T) and the frequency of the tick-timer in the system. The available data is imported to MATLAB and the following equations are generated:

Overhead per tick = 1.9 x T + 20 in μsec (Equation 7-1)

Total overhead due to tick-timer =

(overhead per tick) x (number of ticks that occurs during the task's execution)

(Equation 7-2)

The number of ticks in a task's execution depends on the tick frequency. For example, the total overhead due to tick-timer with frequency of 2000 tick/sec =

$$(1.9 \times T + 20) \times (3.9E\text{-}5 \times F^2 + 0.9 \times F + 4.7) \text{ in } \mu\text{sec} \qquad \text{(Equation 7-3)}$$

where T is the number of tasks, F is the number of ticks that occurs during the task's execution and the quadratic F^2 is for curve fitting.

Note that the coefficients in the second term of Equation 7-3 change with different tick-timer frequencies as shown in Table 7-1. In the experiment, the calculated overhead (e.g. using Equation 7-3) is compared to the measured overhead for the system. The error percentage is between 0.09% and 0.52% with an average of 0.3%. The response-time jitter which is the overhead of one tick can contribute about 0.05% error. The remaining error may be due to the target system anomaly described later.

Tick-timer Frequency (tick/sec)	Number of Ticks Equation
100	$1.7E\text{-}5 \times F^2 + 0.95 \times F - 2.7$
1000	$2.3E\text{-}5 \times F^2 + 0.94 \times F + 1.3$
2000	$3.9E\text{-}5 \times F^2 + 0.9 \times F + 4.7$
4000	$4.9E\text{-}5 \times F^2 + 0.89 \times F + 7$

Table 7-1: Number of Ticks Equation as a Function of Tick-Timer Frequency

The second experiment finds the overheads due to different context switches, and the collected data is shown in Table 7-2. The overhead due to a context switch at a critical instant is the time consumed to launch the highest ready priority task for the first time when the RTOS begins multitasking. This context switch is different from others because it does not include swapping out a currently running task before launching the first task. A hardware timer is triggered before starting multitasking in the main function, and then the timer count is read at the beginning of the highest priority task. There is a different context switch overhead associated with preemption by a higher priority task. This preemption is done by the tick-timer function in the SW RTOS, and it is done from the context switch ISR in the SW/HW RTOS. A hardware timer is triggered before performing the context switch in the tick-timer function, and then the timer count is read at the beginning of the next task to run. A final type of context switch overhead occurs when a task completes executing in its current period. This context switch relinquishes the processor to the next ready task, and is initiated in both systems when the application task calls the delay function. In the experiment, a hardware timer is triggered before calling the delay function, and then the timer count is read at the beginning of the next task to run. The measurements obtained in the experiment are shown in Table 7-2.

	SW RTOS Overhead (µsec)	**SW/HW RTOS Overhead (µsec)**
Context switch @ critical instant	8.5125	11.575
Context switch due to preemption	17.6875	19.1875
Context switch due to task completes	17.6125	17.0375

Table 7-2: Overhead due to Context Switches

For all context switches in the partitioned RTOS system, the modified SW RTOS consumes extra time to communicate with the HWRTOS. It must first construct the command that asks the HWRTOS for the context switch information, write the request to a WR register, and then read the results from the RD register before performing the context switch.

Although the extra hardware register access is not required in the SW RTOS, the third line in the table shows that the context switch overhead due to a task completing in the SW RTOS is about 3.4% more than in the SW/HW RTOS. This is likely due to the SW RTOS removing the completed task from the ready list before finding the highest priority task and performing the context switch. The list overhead is not present in the SW/HW RTOS since ready list manipulation is done in hardware.

The overhead due to context switches because of preemption and at the critical instant in the SW RTOS are about 7.8% and 26% less than in the SW/HW RTOS respectively. For a context switch due to preemption in the SW/HW RTOS, the context switch interrupt is asynchronous. Since the modified SW RTOS disables interrupts during the HWRTOS access, then there is a possibility that the task changes as a results of a request sent from the modified SW RTOS while the context switch interrupt is pending. A check procedure is implemented in the context switch ISR to account for this scenario and that check consumes extra time. For a context switch at a critical instant, the SW RTOS only finds the highest priority task before starting to execute it compared to the process of the HWRTOS communication in the SW/HW RTOS.

The final experiment measures the two factors shown in Figure 7-3. The first is the time consumed by executing from the beginning of the task to the beginning of the infinite loop (labeled by 1 in the figure). A hardware timer is triggered at the beginning of the task, and then the timer count is read just before the infinite loop. This time depends on the code that is used in each task and would typically include variable initialization, timer interaction, I/O initialization, and so on. The second factor is the overhead in managing the delay to the next release of the task, which is required when implementing a periodic task in the SW RTOS (labeled by 2 in the figure). A hardware timer is triggered before delay management, and then the timer count is read before delay service call. For the SW/HW RTOS, this software overhead is not present because the SW/HW RTOS manages support for periodic tasking in hardware. The overhead in managing the delay for the SW RTOS is 7.7 μ sec. The complete results for these two factors are presented in Appendix D.

```
Task()
{
     “                                  ┐  /* 1 */
A = 50;                                 │  /* variable, depends on application code */
     “                                  ┘  /* executes once */
While (1)
  {
     “
     Time      = Read timer;            ┐  /* 2 */
     Dly_time = (++Loop * Period) - Time;  /* overhead to implement periodic */
     Delay(Dly_time);                 ┘  /* task, executes every release */
  }
}
```

Figure 7-3: Example of Code at the Beginning of a Task and Calculating Delay

8 System Overhead Analysis

Developers of hard real-time systems need practical techniques to predict whether tasks will meet their deadlines. This chapter uses the results of previous chapters to improve upon the standard response time analysis model by more accurately modeling system overheads. Improved models are developed for both the SW RTOS and the SW/HW RTOS. A case study is then presented to show the effect of the system overhead on the analysis of hard real-time systems. The standard response time analysis is calculated first. The improved response time analysis models are then applied to both the pure SW RTOS and the partitioned SW/HW RTOS. A comparison of the results reveals that the improved models are superior to the standard model, and the SW/HW RTOS model is more accurate than the pure SW RTOS model.

The following conventions are used in the discussion:

O_tick: Overhead per tick due to a tick-timer.

E_start_i : Time consumed in executing the code from the beginning of the task to the start of the infinite loop for $task_i$.

O_cal_i : Overhead due to managing the delay to the next release of $task_i$.

O_ctx_{ci} : Overhead due a context switch at critical instant.

O_ctx_{pre} : Overhead due to preemption by a higher priority task.

O_ctx_{te} : Overhead due to a context switch when the task completes executing in its current period.

8.1 Improved Response Time Models

Recall that the standard response time model includes the response time of each task and the delay caused by all higher priority tasks, but does not include overheads.

$$R_i = e_i + \sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil e_j$$

The improved response time model for the **SW RTOS** for all tasks is presented in Equation 8-1. Assuming that all tasks are created before calling OS_Start, the improved model includes task execution time and system overhead for each task and all higher priority tasks at the first critical instant as follow:

$$R_i = E\_start_i + e_i + O\_cal_i + O\_ctx_{te} \quad \leftarrow 1$$

$$+ O\_ctx_{ci} \quad \leftarrow 2$$

$$+ \sum_{j=0}^{i-1} E\_start_j + \sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil \left(e_j + O\_ctx_{pre} + O\_cal_j + O\_ctx_{te}\right) \quad \leftarrow 3$$

(Equation 8-1)

where *i* denotes the task for which the response time is being calculated, and *j* is used for all higher priority tasks.

1: terms due to $task_i$,

2: term due to starting the highest priority task at critical instant, and

3: terms due to $task_j$

Note that O_ctx_{ci} is only included once because it only occurs at the launch of the highest priority task at critical instant. The E_start_x term is only included once for each task because after the first execution, the task stays inside the infinite loop. The model assumes that every release of all higher priority tasks begins with a context switch due to preemption; however this is not necessarily the case and some of the switches may be due to completion of even higher priority tasks. Therefore, O_ctx_{pre} is included in the calculation of preemption by higher priority tasks, even though it might not have been incurred. This may result in a slightly pessimistic prediction, but there does not appear to be a simple way to determine exactly whether this overhead is incurred. The other overheads (O_cal_x, O_ctx_{te}) are included in all calculations for each task.

Every task is preempted by the higher priority tick-timer interrupt. For the overhead due to the tick-timer (i.e. $\sum$ terms in Equation 8-1), E_start, O_cal, O_ctx_{pre} and O_ctx_{te} are equal to zero, e is equal to O_tick and p is equal to the tick-timer period.

Similarly, the model for the **SW/HW RTOS** for all tasks includes task execution time and system overhead for each task and all higher priority tasks as follow:

$$R_i = E\_start_i + e_i + O\_ctx_{te} \quad \leftarrow 1$$

$$+ O\_ctx_{ci} \quad \leftarrow 2$$

$$+ \sum_{j=0}^{i-1} E\_start_j + \sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil \left(e_j + O\_ctx_{pre} + O\_ctx_{te} \right) \quad \leftarrow 3$$

(Equation 8-2)

where *i* denotes the task for which the response time is being calculated, and *j* is used for all higher priority tasks.

1: terms due to $task_i$,

2: term due to starting the highest priority task at critical instant, and

3: terms due to $task_j$

The O_cal_x and the overhead due to the tick-timer interrupt are zero for the partitioned RTOS because the tick-timer and the periodic task support are implemented in hardware.

8.2 Case Study

The case study includes three periodic tasks with the parameters in Table 8-1. The measured execution time for T1 is about 2.499 sec and for T2, T3 is about 1.249 sec. Task execution times are large for an embedded real-time system, but the utilizations are realistic and more relevant for the analysis. All detailed calculations for this chapter are in Appendix F and the data for E_start_i and O_cal_i for the case study are in Appendix D.

	Execution Time (e) in sec	Period (p) in sec	Deadline (d) in sec	Utilization (u = e/p)
Task1 (T_1)	2.5	3.75	3.75	0.66667
Task2 (T_2)	1.25	7.5	7.5	0.16667
Task3 (T_3)	1.25	15	15	0.08333

Table 8-1: Case Study Parameters

The analysis starts with the utilization test:

$$U_{total} = \sum_{i=1}^{3} u_i = 0.91667$$

$$U_{RM}(3) = 3(2^{1/3} - 1) = 0.77976$$

No conclusions can be drawn since $1 \geq U_{total} > U_{RM}(3)$, and the response time analysis is done next.

For the Standard Response Time Analysis, using Equation 2-4 on Page 15:

T_1 : ($R_1 = 2.5$) < ($d_1 = 3.75$) => Task 1 meets its deadline

T_2 : ($R_2 = 3.75$) < ($d_2 = 7.5$) => Task 2 meets its deadline

T_3 : ($R_3 = 7.5$) < ($d_3 = 15$) => Task 3 meets its deadline

The final result is that all tasks are schedulable for this case.

Analysis using the improved response time model including system overhead for SW RTOS (Equation 8-1) is conducted for various tick-timer frequencies. In the analysis, there are 3 tasks (T = 3), and Equation 7-1 predicts that the overhead per tick is O_tick = 25.7 μsec.

For the tick-timer frequency of 100 tick/sec, Equation 8-1 predicts:

T_1 : ($R_1 = 2.50652$) < ($d_1 = 3.75$) => Task 1 meets its deadline

T_2 : ($R_2 = 6.26627$) < ($d_2 = 7.5$) => Task 2 meets its deadline

T_3 : ($R_3 = 13.78576$) < ($d_3 = 15$) => Task 3 meets its deadline

Therefore, all tasks are schedulable in this case.

For the tick-timer frequency of 4000 tick/sec, Equation 8-1 predicts:

T_1 : (R_1 = 2.78655) < (d_1 = 3.5) => Task 1 meets its deadline

T_2 : (R_2 = 6.96631) < (d_2 = 7.5) => Task 2 meets its deadline

T_3 : (R_3 = 22.2921) > (d_3 = 15) => Task 3 does not meet its deadline

Therefore, the task set is not schedulable when the number of OS ticks = 4000 tick/sec.

Analysis using the improved response time model including system overheads for SW/HW RTOS (Equation 8-2) does not involve tick-timer overhead because the tick-timer is implemented in hardware. Therefore, the results are the same regardless of the tick-timer rate.

Equation 8-2 predicts:

T_1 : (R_1 = 2.50006) < (d_1 = 3.75) => Task 1 meets its deadline

T_2 : (R_2 = 6.25013) < (d_2 = 7.5) => Task 2 meets its deadline

T_3 : (R_3 = 13.75028) < (d_3 = 15) => Task 3 meets its deadline

Therefore, all tasks are schedulable for the SW/HW RTOS regardless of the tick frequency. Note that the tasks are schedulable for the SW/HW RTOS in the case where the tick-timer rate is 4000 tick/sec and recall that the tasks are not schedulable for pure SW RTOS at this tick-timer frequency.

Measurements on the real system allow a comparison between the calculated analysis results with the actual measured results. Table 8-2 shows the calculated and measured results along with the percent difference between them. The data shows that the measured values are consistent with the predicted values. The error percentage for the SW RTOS is

likely due to the target system anomaly. Table 8-3 shows the predicted results using the standard model versus the measured results. The worst case prediction error reaches about 65.8% when using the standard model.

The results show that taking system overhead into consideration during the analysis leads to results that reflect the real performance, which is very important in hard real-time systems. Furthermore, using the partitioned SW/HW RTOS improves system performance and allows the applications to have a higher utilization of the processor.

# of Ticks	Task	SW RTOS			SW/HW RTOS		
		Calculated End (sec)	Measured End (sec)	Diff. %	Calculated End (sec)	Measured End (sec)	Diff. %
100	T1	2.50652	2.5054	0.04468	2.50006	2.50006	0.00012
	T2	6.26627	6.26348	0.04455	6.25009	6.25013	0.00016
	T3	13.78576	13.77964	0.04444	13.75023	13.75028	0.00031
4000	T1	2.78655	2.73827	1.76292	2.50006	2.50006	0.00011
	T2	6.96631	6.84569	1.76204	6.25009	6.25013	0.00016
	T3	22.2921	21.9074	1.75602	13.75023	13.75028	0.00031

Table 8-2: Overhead Analysis Comparison between Calculated and Measured Results

Task	SW RTOS				
	Standard Model Calculated End (sec)	Measured End (100 tick/sec) (sec)	Diff. %	Measured End (4000 tick/sec) (sec)	Diff. %
T1	2.5	2.5054	0.21553	2.73827	8.70148
T2	3.75	6.26348	40.12913	6.84569	45.22101
T3	7.5	13.77964	45.57187	21.9074	65.76499

Table 8-3: Standard Model Prediction VS. Measured Results

9 Performance of some RTOS Services

The performance of some of the main RTOS services is measured for the pure SW RTOS and the partitioned SW/HW RTOS and the results are shown in Table 9-1. A discussion of each service and a performance comparison and analysis is discussed next. To obtain the measurements, a hardware timer is triggered before each service call, and then the timer count is read after the call. Some of the results of the services that include context switch are expected to be close to the results in Table 7-1.

The RTOS initialization process (OS_Init) initializes all system variables, the TCB list, the ECB list, the ready list and the idle task. The measured results show about 83% performance improvement in the case of the SW/HW RTOS. The improvement is due to the implementation of the TCB, ECB and ready list in hardware for the SW/HW RTOS, so there is no need to initialize them in software. Most of the measured time (about 91 μsec) is used for idle task creation which is done during the software part of the RTOS initialization process, before invoking the hardware initialization.

The time consumed for the task creation service (Create_Task) is also measured, with no context switch involved. Task creation includes initializing the task's stack which is done in software. The results show about 5.8% performance improvement in the case of the SW/HW RTOS. This low improvement is due to the need for communication with the HWRTOS in the partitioned SW/HW RTOS.

System Calls	Pure SW RTOS (µsec)	Partitioned SW/HW RTOS (µsec)
OS_Init	604.4	99.675
Create_Task	97.663	91.088
Sema4_Pend (Key)	4.475	9.575
Sema4_Pend (No Key)	21.325	20.375
Mbox_Pend (Msg)	4.213	9.125
Mbox_Pend (No Msg)	20.988	20.100

Table 9-1: Performance Comparison between SW RTOS and SW/HW RTOS

Sema4_Pend(no key) and Mbox_Pend(no msg) services involve a context switch after each service call. The test is done to measure the time consumed in requesting a semaphore or a message mailbox when the semaphore is not available because it is locked by another task, or a message is not available because the mailbox is empty. In this measurement, a hardware timer is triggered before the service call and the timer count is read at the beginning of the higher priority task that takes control after the context switch. The results show about 4.5% performance improvement in the case of the SW/HW RTOS. Again, this relatively small improvement is also due to the need for HWRTOS communication. Note that there is twice as much required HWRTOS communication in this case because the measured activity includes both the service and a context switch.

The last two services under test are Sema4_Pend(key) and Mbox_Pend(msg). There is no context switch and waiting required in this case because the semaphore or the

message is available. The results do not perform as well in the case of the SW/HW RTOS due to the need for several function calls for HWRTOS communication which involves constructing the command then accessing the bus for write and read operations.

10 General Discussion

There are some encouraging results in Chapter 7, 8 and 9. Using the extended response time model significantly improves real-time system performance prediction over the use of the standard model. Also, the partitioned SW/HW RTOS improves system performance by eliminating some overheads. Implementing the periodic tick-timer and supporting periodic task in hardware also eliminates the time drift problem and the response-time jitter typical of a pure SW RTOS. Unfortunately, some of the results of this research have been constrained by the limitations of the target system and the limited time frame for the research, and this chapter discusses these limitations and suggests some solutions to improve the system in the future.

The partitioned SW/HW RTOS does not support general interrupts. The successful implementation of tick-timer interrupt functionality in hardware might be built upon to implement more interrupts in hardware in the future. Ideally, this might help to reduce the jitter typical of interrupts sharing a processor, and ultimately result in faster response times. An interrupt management block needs to be designed to manage all interrupts implemented in hardware.

The current design of the partitioned RTOS requires disabling interrupts in the modified SW RTOS around request and read commands to the HWRTOS. Perhaps this is not the best solution for creating atomic access. The need to disable interrupts could be eliminated using a register in task's TCB to store the last request result. The HWRTOS

could store the result of each request in the TCB of the task that sends the request. If a context switch interrupt occurs between request and read commands, then any further results will be stored in the new task's TCB. On returning from the interrupt, the preempted task could still read the stored results of its last request from its TCB.

The current design of the partitioned RTOS requires performing the context switch in software, and the related communication with the HWRTOS consumes valuable time. Having direct access to the CPU registers would allow the context switch to be performed in hardware and could eliminate the needs for any software actions associated with context switches. The HWRTOS TCBs could be expanded to include storage of the CPU registers. If a context switch is required, then the CPU registers could be saved and restored directly from the TCB storage. A task software stack is still needed to store local variables and program counter during function calls, but these are attributes of the application, and the stack would not be used during context switching.

The current approach to providing support for periodic tasks in the partitioned RTOS prevents non-periodic tasks from using timed services. If zero is passed as the period during task creation, then the created task is not periodic. Once this task starts executing, then it keeps executing until a context switch interrupt occurs and a higher priority task takes CPU control. Implementing the delay function for non-periodic tasks could be done in future work. That would involve expanding the tick-timer functionality to manage task periods as well as timed service delays.

Semaphore and message mailbox are implemented in the HWRTOS. The system overhead analysis could be extended in the future to add task synchronization and communication overhead. Also, the priority ceiling protocol could be implemented in the HWRTOS as a solution for the unbounded priority inversion, deadlock and chain blocking problems.

An apparent anomaly was experienced in the use of the AP1000 boards. While measuring the execution time of the simple "for loop" code (recall Figure 7-1), the results sometimes depended on changes to code that had no theoretical connection to the execution times. For example, adding a line of code outside of the for loop changed the execution time of the for loop. A considerable amount of effort was invested by the Xilinx University Program support team to isolate the cause of this anomaly; however, they were unable to reproduce the problem using the same versions of the development tools and their Virtex-II Pro target boards. The problem was reproduced on the AP1000 target boards by the CMC support engineers (recall that CMC donated the AP1000 target board to encourage research in embedded real-time systems); however they too were not able to isolate the cause. These experiences suggest that the problem may lie in the particular target board. During experiments to isolate the anomaly, the experienced timing variations were all within 1%. This small error does not negate the results, but requires caution when interpreting the results.

11 Conclusions and Future Work

System overheads and jitter associated with using real-time operating systems can cause non-deterministic behaviour which hampers the ability to predict response times in a HRT environment. This thesis studied these problems on a system using a PowerPC 405 and the μC/OS-II RTOS in several configurations. The pure SW RTOS provides a baseline for comparing analysis models, and also for comparing subsequent improvements obtained by introducing direct hardware support to the RTOS. A partitioned SW/HW RTOS is developed to provide real data about direct hardware support for an RTOS. Throughout the experiments, time measurements were gathered using hardware timers and hardware supported profiling. The collected data is used to extend and improve the standard response time model to include system overhead due to context switches and tick-timer management.

The improved response time model gives a more accurate analysis which improves the practicality of applying the response time analysis. The case study reveals several orders of magnitude increase in the accuracy of the extended model over the standard model in the case of the partitioned SW/HW RTOS system. Also, using the SW/HW RTOS as part of the system increases the determinism and performance of the real-time system by reducing system overhead and jitter.

The presented case study verifies the correctness of the improved response time model by calculation and measurements, and shows the advantage of using the SW/HW

RTOS in the real-time systems. The system using the SW/HW RTOS successfully schedules all tasks in the case study at 4000 tick/sec, whereas the pure SW RTOS is unable to schedule the tasks.

Implementing a profiling technique that uses a high resolution hardware timers and hardware-based data collection greatly simplified system testing and the validation of HRT constraints.

Implementing context switching in hardware is a promising direction for future work, as it may further reduce software overheads and simplify the interface between the software and hardware components of a partitioned RTOS. Accomplishing a context switch in hardware requires direct hardware access to the processor's registers. Future work could investigate the use of soft core processors such as the "MicroBlaze" or newer version of Xilinx FPGA that provides direct access to the processor's registers. In addition, expanding the CPU register set would make it possible to place the requests and results directly in the CPU registers and eliminate the need to perform bus access during HWRTOS communication.

Future work could also investigate adding more system services to the hardware RTOS to minimize system overhead including adding the ability of delaying periodic tasks, interrupts management to support general interrupts and implementing a priority ceiling protocol in hardware to go with the analysis one step further by allowing task synchronization and communication.

12 References

[1] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai, "Hardware Implementation of a Real-Time Operating System", *Proceeding of IEEE International Symposium of the 12th TRON Project*, Tokyo, Japan, 1995.

[2] A. Parisoto, A. Souza, L. Carro, M. Pontremoli, C. Pereira, A. Suzim, "F-Timer: Dedicated FPGA to Real-Time Systems Design Support", *Proceeding of Ninth Euromicro Workshop on Real-Time Systems*, Toledo, Spain, 1997.

[3] J. Hildebrandt, D. Timmermann, "An FPGA Based Scheduling Coprocessor for Dynamic Priority Scheduling in Hard Real-Time Systems", *Proceeding of FPL 2000, Tenth International Conference on Field Programmable Logic and Applications*, Villach, Austria, 2000.

[4] P. Kuacharoen, M. Shalan, V. Mooney III, "A Configurable Hardware Scheduler for Real-Time Systems", *Proceeding of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Nevada, USA, 2003.

[5] P. Kohout, B. Ganesh, B. Jacob, "Hardware Support for Real-Time Operating Systems", *Proceeding of the First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis*, California, USA, 2003.

[6] M. Timmerman, L. Perneel, "RTOS State of the Art: Understanding RTOS Technology and Markets", Doc no. DSE-RTOS-EVA-001b by Dedicated Systems Experts, Belgium, 2005.

[7] J. Labrosse, *MicroC/OS-II The Real-Time Kernel*, 2nd Edition, Published by CMP Books, California, USA, 2002.

[8] Jane W. Liu, *Real-Time Systems*, 1st Edition, Published by Prentice Hall PTR, USA, 2000.

[9] C. L. Liu, James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the Association for Computing Machinery*, Vol. 20, USA, 1973.

[10] Micrium Inc., "μC/OS-II and the PowerPC 405", Application Note AN-1405

[11] Xilinx Inc., "Embedded System Tools Reference Manual", EDK9.1i, UG111 (v7.0), 2007.

[12] AMIRIX Systems Inc., "AP1000 FPGA Development Board Users Guide", DOC-004017 Version 02, 2005.

[13] CMC Microsystems, "Getting Started with the Virtex-II Pro FPGA Prototyping Station", ICI-155 V4.1, 2007.

[14] Philip J. Koopman Jr., "Design Constraints on Embedded Real-Time Control Systems", *Proceeding of Systems Design and Networks Conference*, Santa Clara, California, USA, 1990.

[15] S. Edwards, E. Lee, "The Case of the Precision Timed (PRET) Machine", *Proceeding of the 44th ACM/IEEE Design Automation Conference*, California, USA, 2007.

[16] L. Lindh, F. Stanischewski, "FASTCHART - A Fast Deterministic CPU and Hardware Based Real-Time Kernel", *Proceeding of Euromicro'91 Workshop on Real-Time Systems*, Paris-Orsay, France, 1991.

[17] L. Lindh, F. Stanischewski, "FASTCHART - Idea and Implementation", *Proceeding of IEEE International Conference on Computer Design*, Cambridge, Massachusetts, USA, 1991.

[18] L. Lindh, "FASTHARD - A Fast Time Deterministic Hardware Based Real-Time Kernel", *Proceeding of Fourth Euromicro Workshop on Real-Time Systems*, 1992.

[19] L. Lindh, J. Starner, J. Furunas, "From Single to Multiprocessor Real-Time Kernels in Hardware", *Proceeding of Real-Time Technology and Applications Symposium*, Chicago, Illinois, USA, 1995.

[20] J. Adomat, J. Furunas, L. Lindh, J. Starner, "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High Performance Real-Time Systems", *Proceeding of Eighth Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, 1996.

[21] Sierra Real-Time Kernel, Sweden, Internet:www.agstu.com/index.html, [Jan. 5, 2010].

[22] S. Nordstrom, L. Lindh, L. Johansson, T. Skoglund, "Application Specific Real-Time Microkernel in Hardware", *14th IEEE-NPSS Real-Time Conference*, Stockholm, Sweden, 2005.

[23] A. Morton, W. Loucks, "A Hardware/Software Kernel for System on Chip Designs", *Proceeding of the 2004 ACM Symposium on Applied Computing*, Nicosia, Cyprus, 2004.

[24] V. Mooney III, D. Blough, "A Hardware/Software Real-Time Operating System Framework for SOCs", *IEEE Design & Test of Computers*, USA, 2002.

[25] V. Mooney III, "Hardware/Software Partitioning of Operating Systems", *Proceeding of Conference and Exhibition on Design, Automation and Test in Europe*, Munich, Germany, 2003.

[26] J. Lee, V. Mooney III, "Hardware/Software Partitioning of Operating Systems: Focus on Deadlock Detection and Avoidance", *IEE Proceeding, Computer and Digital Techniques*, UK, 2005.

[27] B. Akgul, V. Mooney III, H. Thane, P. Kuacharoen, "Hardware Support for Priority Inheritance", *Proceeding of 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, 2003.

[28] J. Lee, V. Mooney III, "RTPOS: A Novel Deadlock Avoidance Algorithm and Its Hardware Implementation", 2004.

[29] S. Chatterjee, J. Strosnider, "Quantitative Analysis of Hardware Support for Real-Time Operating Systems", Real-Time Systems, Netherlands, 1996.

[30] D. I. Katcher, H. Arakawa, J. K. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers", *IEEE Transactions on Software Engineering*, USA, 1993.

[31] F. Stanischewski, "FASTCHART - Performance, Benefits and Disadvantages of the Architecture", *Proceeding of Fifth Euromicro Workshop on Real-Time Systems*, 1993.

[32] J. Lee, V. Mooney III, A. Daleby, K. Ingstrom, T. Klevin, L. Lindh, "A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS", *Proceeding of the 2003 Asia and South Pacific Design Automation Conference*, Kitakyushu, Japan, 2003.

[33] M. Sindhwani, T. Oliver, D. Maskell, T. Srikanthan, "RTOS Acceleration Techniques – Review and Challenges", *Sixth Real-Time Linux Workshop*, Singapore, 2004.

[34] T. Oliver, S. Mohammed, N. Krishna, D. Maskell, "Accelerating an Embedded RTOS in a SOPC Platform", *TENCON 2004, 2004 IEEE Region 10 Conference*, Chiang Mai, Thailand, 2004.

[35] M. Vetromille, L. Ost, C. Marcon, C. Reif, F. Hessel, "RTOS Scheduler Implementation in Hardware and Software for Real-Time Applications", *Proceeding of the 17th IEEE International Workshop on Rapid System Prototyping*, Greece, 2006.

13 Appendices

13.1 Appendix A: µC/OS-II Error Codes

Error Code	Cause
0x01	Wrong event type is passed during accessing any of the semaphore, mutex, message mailbox and message queue.
0x02	Pending on a semaphore or on a message from within an ISR which is not allowed because that could cause a context switch if the semaphore or the message was not available and that causes non-determinism of the ISR's response time.
0x03	The pointer to the message that is used in posting a message to a mailbox operation is null or empty.
0x04	The ECB that is reserved for an event is invalid.
0x0A	The timeout for a semaphore or a message mailbox is expired.
0x14	Posting a message to a full mailbox (already contains a message).
0x28	Trying to create a task with priority already exist which is not allowed because task priority must be unique.
0x2A	Using an invalid (not exist) priority to reference a task.
0x32	Semaphore count is overflowed (more than 0xFFFF). Since pending on a semaphore decrements the semaphore count and posting a semaphore increments its count, then the error indicates that the application keeps posting the semaphore without pending on it.
0x3C	Trying to create a task from within an ISR which is not allowed because creating a task adds the task to the ready list and calls the scheduler which could cause a context switch and that is not allowed from an ISR.

Table 13-1: µC/OS-II Error Codes

13.2 Appendix B: SW/HW Interface

All requests require one write register (Reg0) and all reads require one read register (Reg4) except read test information requires three read registers (Reg4, Reg5 and Reg6). The shaded fields in all request and read tables are not used (zeros).

HWRTOS Initialization Request:

Initialization request is used to initialize some of the HWRTOS parameters. Table 13-2 shows the fields.

Arguments				Command	
[63..32]	[31..17]	[16]	[15..8]	[7..0]	Reg0
T_Value		Int	I_Prio	0x01	

Table 13-2: HWRTOS Initialize Request

Command type: Initialize the HWRTOS (0x01).

Arguments:

I_Prio: Idle task priority which is equal to the lowest priority (0x0F).

Int: Setting this bit to '1' simulates the context switch interrupt generated from the scheduler to the PPC405 (used for test purpose).

T_Value: The value that should be loaded to the internal hardware tick- timer to set it to the required frequency. It is the output of the following equation:

$$HWRTOS\_PLB\_CLK\_FREQ / HWRTOS\_TICKS\_PER\_SEC$$

Where: HWRTOS_PLB_CLK_FREQ = 80 MHZ for the Amirix target board and HWRTOS_TICKS_PER_SEC is settable from "bsp.h" header file

HWRTOS Tick-Timer Enable Request:

Tick-timer enable request is used to enable the internal tick-timer. Table 13-3 shows the fields.

Arguments [63..8]	Command [7..0]	Reg0
0x00000000000000	0x02	

Table 13-3: HWRTOS Tick-Timer Enable Request

Command type: Enable the tick-timer (0x02).

Arguments: None.

HWRTOS Tick-Timer Disable Request:

Tick-timer disable request is used to disable the internal tick-timer. Table 13-4 shows the fields.

Arguments [63..8]	Command [7..0]	Reg0
0x00000000000000	0x03	

Table 13-4: HWRTOS Tick-Timer Disable Request

Command type: Disable the tick-timer (0x02).

Arguments: None.

Context Switch Information Request:

Context switch information request is used to response to a context switch interrupt from the HWRTOS. It asks the HWRTOS to put the context switch information to read register (Reg4). Table 13-5 shows the fields.

Arguments		Command	Reg0
[63..16]	[15..8]	[7..0]	
	Ctx_Sw	0x04	

Table 13-5: Context Switch Info Request

Command type: Request context switch information (0x04).

Arguments:

Ctx_Sw: If = "0x00", then asks the HWRTOS to put the context switch information in read register (Reg4) and reset the interrupt.

If = "0x01", then reset the HWRTOS interrupt only because there is interrupt nesting and a context switch is not allowed in this case.

Context Switch Information Read:

Context Switch Information Read follows the context switch information request to read the data provided by the HWRTOS related to the context switch. Table 13-6 shows the fields returned following a context switch information request.

Data			Status	Reg4
[63..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Stat	

Table 13-6: Context Switch Info Read

Status: The status of the context switch information request.

Data:

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor.

Create Task Request:

Create task request is used to create a task with the parameters provided in the arguments. Table 13-7 shows the fields.

Arguments					Command	Reg0
[63..56]	[55..48]	[47..32]	[31..16]	[15..8]	[7..0]	
	Name	Period	Id	Prio	0x05	

Table 13-7: Create Task Request

Command type: Request to create a task with the provided parameters (0x05).

Arguments:

Prio: Unique task priority from 0 to 15 (0 is the highest priority).

Id: Task id (could be any thing, it is logged in the TCB, but not used by the HWRTOS).

Period: Task's period in number of ticks (period = 0 if task is not periodic).

Name: ASCII code of one character that represents the task's name.

Create Task Information Read:

Create task information read and it follows create task request to read the data provided by the HWRTOS related to task creation. Table 13-8 shows the fields returned following a create task request.

Data					Status	Reg4
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Ctx_Sw	Free	Stat	

Table 13-8: Create Task Info Read

Status: The status of the create task request.

Data:

Free: If this field is equal to "0x00", then the requested task priority was not reserved previously by any task and is now reserved for the newly created task and the application code should proceed. If the requested priority is already reserved by another task, then "priority already exist (0x28)" error will be returned in the field.

Ctx_Sw: If the task creation requires the system to make a context switch, then this field is equal to "0x01". Otherwise, this field is equal to "0x00".

Prio_Cur: Priority of the current task that needs to be switched in the case that a context switch is required (i.e. when Ctx_Sw = "0x01").

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor in case of Ctx_Sw = "0x01".

Task Done Request:

Task done request is used to inform the HWRTOS that the current task completed executing for this period and should be removed from the ready list and wait to the next period to start execution again. Table 13-9 shows the fields.

Arguments [63..8]	Command [7..0]	Reg0
0x00000000000000	0x06	

Table 13-9: Task Done Request

Command type: Informs the HWRTOS that the tasks reached the end of its code (0x06).

Arguments: None.

Task Done Information Read:

Task done information read follows task done request to read the data provided by the HWRTOS. Table 13-10 shows the fields returned following a task done request.

Data				Status	Reg4
[63..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Ctx_Sw	Stat	

Table 13-10: Task Done Info Read

Status: The status of the task done request.

Data:

Ctx_Sw: This field is always equal to "0x01" which means the context switch is required.

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor.

Create Semaphore Request:

Create semaphore request is used to create a semaphore to protect access to a shared resources. Table 13-11 shows the fields.

Arguments		Command	Reg0
[63..24]	[23..8]	[7..0]	
	Cnt	0x07	

Table 13-11: Create Semaphore Request

Command type: Request to create a semaphore for tasks synchronization (0x07).

Arguments:

Cnt: Initial value to specify the number of available resources, if it is 0, then it means no resource is available.

Create Semaphore Information Read:

Create semaphore information read follows create semaphore request to read the semaphore ID reserved by the HWRTOS. Table 13-12 shows the fields returned following a create semaphore request.

Data		Status	Reg4
[63..16]	[15..8]	[7..0]	
	Ecb_N	Stat	

Table 13-12: Create Semaphore Info Read

Status: The status of the create semaphore request.

Data:

Ecb_N: The reserved event control block ID by the system for the semaphore. Any future reference to this semaphore must use this ID.

Semaphore Pend Request:

Semaphore pend request is used to wait for a semaphore for an optional timeout period in ticks. Table 13-13 shows the fields.

Arguments			**Command**	**Reg0**
[63..32]	[31..16]	[15..8]	[7..0]	
	Timeout	Event	0x08	

Table 13-13: Semaphore Pend Request

Command type: Request to wait for a semaphore (0x08).

Arguments:

Event: The semaphore ID that needs to wait for.

Timeout: An optional timeout in ticks. If the timeout is equal to zero, then that means the task should wait for the semaphore until it is available. If the timeout is not zero and expires before the semaphore become available, then an error will be generated.

Semaphore Pend Information Read:

Semaphore pend information read follows semaphore pend request to read the semaphore pend result. Table 13-14 shows the fields returned following a semaphore pend request.

Data					Status	Reg4
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	Prio_Cur	Ctx_Sw	Err	Stat	

Table 13-14: Semaphore Pend Info Read

Status: The status of the semaphore pend request.

Data:

Err: An error code. When it is equal to "0x00", it means there is no error and when it is equal to "0x01", it means the provided semaphore ID is not for semaphore type (type error).

Ctx_Sw: If the semaphore is not available then this field is equal to "0x01" which means the task should wait for the semaphore and there is a need for a context switch. Otherwise, this field is equal to "0x00" and the task is granted the semaphore.

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor.

Semaphore Post Request:

Semaphore post request is used to signal a semaphore and makes it available. Table 13-15 shows the fields.

Arguments		Command	
[63..16]	[15..8]	[7..0]	Reg0
	Event	0x0A	

Table 13-15: Semaphore Post Request

Command type: Request to signal a semaphore (0x0A).

Arguments:

Event: The semaphore ID to signal.

Semaphore Post Information Read:

Semaphore post information read follows semaphore post request to read the semaphore post result. Table 13-16 shows the fields returned following a semaphore post request.

Data					Status	
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	Reg4
	Prio_H	Prio_Cur	Ctx_Sw	Err	Stat	

Table 13-16: Semaphore Post Info Read

Status: The status of the semaphore post request.

Data:

Err: An error code. When it is equal to "0x00", there is no error. A value of "0x01" indicates that the provided semaphore ID is not for a valid semaphore. A value of "0x32" indicates that the post has resulted in semaphore count overflow.

Ctx_Sw: If releasing the semaphore causes a context switch, then this field is equal to "0x01". Otherwise, this field is equal to "0x00".

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor.

Create Mailbox Request:

Create mailbox request requires is used to create a mailbox to communicate with other tasks. Table 13-17 shows the fields.

Arguments		**Command**	**Reg0**
[63..32]	[31..8]	[7..0]	
Msg		0x0B	

Table 13-17: Create Mailbox Request

Command type: Request to create a mailbox for tasks communication (0x0B).

Arguments:

Msg: A pointer to the message that needs to be put in the mailbox. The message format is not important.

Create Mailbox Information Read:

Create mailbox information read follows create mailbox request to read the reserved ECB ID for the mailbox by the HWRTOS. Table 13-18 shows the fields returned following a create mailbox request.

Data		Status	
[63..16]	[15..8]	[7..0]	Reg4
	Ecb_N	Stat	

Table 13-18: Create Mailbox Info Read

Status: The status of the create mailbox request.

Data:

Ecb_N: The reserved event control block ID by the system for the mailbox. Any future reference to this mailbox should use this ID.

Message Mailbox Pend Request:

Message mailbox pend request is used to wait for a message to be received in a mailbox for an optional timeout period in ticks. Table 13-19 shows the fields.

Arguments			Command	
[63..32]	[31..16]	[15..8]	[7..0]	Reg0
	Timeout	Event	0x0C	

Table 13-19: Message Mailbox Pend Request

Command type: Request to wait for a message (0x0C).

Arguments:

Event: The mailbox ID that the message should be sent to.

Timeout: An optional timeout in ticks. If the timeout is equal to zero, then the task should wait until a message is available. If the timeout is not zero and expires before a message arrives at the mailbox, then an error will be generated.

Message Mailbox Pend Information Read:

Message mailbox pend information read follows message mailbox pend request to read the pend result. Table 13-20 shows the fields returned following a message mailbox pend request.

Data					Status	
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	Reg4
	Prio_H	Prio_Cur	Ctx_Sw	Err	Stat	
Msg						

Table 13-20: Message Mailbox Pend Info Read

Status: The status of the message mailbox pend request.

Data:

Err: An error code. When it is equal to "0x00", it means there is no error and when it is equal to "0x01", it means the provided mailbox ID is not for mailbox type (type error).

Ctx_Sw: If the message is not available then this field is equal to "0x01" which means the task should wait for the message and there is a need for a context switch. Otherwise, this field is equal to "0x00" and the task receives the message.

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H/Msg: If there is a context switch, then this field is the highest priority task that needs to preempt the current task and take control of the processor. If there is no context switch and the task received the message, then this field is a pointer to the message that is found in the mailbox.

Message Mailbox Post Request:

Message mailbox post request is used to send a message to a mailbox. Table 13-21 shows the fields.

Arguments			Command	Reg0
[63..32]	[31..16]	[15..8]	[7..0]	
Msg		Event	0x0E	

Table 13-21: Message Mailbox Post Request

Command type: Request to send a message (0x0E).

Arguments:

Event: The mailbox ID that the message should be sent to.

Msg: A pointer to the message that needs to be put in the mailbox. The message format is not important.

Message Mailbox Post Information Read:

Message mailbox post information read follows message mailbox post request to read the post result. Table 13-22 shows the fields returned following a message mailbox post request.

Data					Status	
[63..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]	Reg4
	Prio_H	Prio_Cur	Ctx_Sw	Err	Stat	

Table 13-22: Message Mailbox Post Info Read

Status: The status of the message mailbox post request.

Data:

Err: An error code. When it is equal to "0x00", it means there is no error, when it is equal to "0x01", it means the provided mailbox ID is not for mailbox type and when it is equal to "0x14", it means mailbox is full.

Ctx_Sw: If delivering the message should cause a context switch, then this field is equal to "0x01". Otherwise, this field is equal to "0x00".

Prio_Cur: Priority of the current task that needs to be switched.

Prio_H: The highest priority task that needs to preempt the current task and take control of the processor.

OS Start Request:

OS start request is used to start the multitasking process. Before calling this function, HWRTOS tick-timer should be enabled. Table 13-23 shows the fields.

Arguments	Command	
[63..8]	[7..0]	Reg0
	0x0F	

Table 13-23: OS Start Request

Command type: Request to start the multitasking process (0x0F).

Arguments: None.

OS Start Information Read:

OS start information read follows OS start request to get the information about the first task needs to run. Table 13-24 shows the fields returned following a OS start request.

Data			Status	Reg4
[63..24]	[23..16]	[15..8]	[7..0]	
	Prio_H	OS_Run	Stat	

Table 13-24: : OS Start Info Read

Status: The status of the OS start request.

Data:

OS_Run: When this field is equal to "0x00", it means the OS was not running and it needs to start the highest priority task. When it is equal to "0x01", it means the OS was running and no action is required.

Prio_H: The highest priority task that needs to take control of the processor.

HWRTOS Test Request:

HWRTOS test request is implemented as a test function where the application code can ask for system and task parameters from the HWRTOS. Table 13-25 shows the fields.

Arguments				Command	Reg0
[63..32]	[31..24]	[23..16]	[15..8]	[7..0]	
	Event	Task	Para	0x10	

Table 13-25: HWRTOS Test Request

Command type: Request to get some of the HWRTOS parameters (0x10).

Arguments:

Para: To choose which parameters to get from the HWRTOS as follows:

"0x00": TCB content.

"0x01": System parameters.

"0x02": ECB content (semaphore or message mailbox).

(Detailed description about each part covered in HWRTOS Test Information Read next).

Task: Task priority to get information about (0 to 15). If Task = "0xFF", then the current task is the desired task.

Event: Event ID for either semaphore or message mailbox.

HWRTOS Test Information Read:

HWRTOS test information read requires three read registers (Reg4, Reg5 and Reg6) and it follows HWRTOS test request to get information about the system. Table 13-26 shows the fields returned following a HWRTOS test request.

Data								**Para**	
[63..56]	[55..48]	[47..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]		
Msg				Task_Ctx				0x00	**Reg4**
Task_Cnt	Prio_Idle	Prio_H	Prio_Cur	OS_Time				0x01	
E_Name	Type	Cnt		Ptr				0x02	

Data								Para	
[63..56]	[55..48]	[47..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]		
Stat_P	Stat	Id		Dec_Period		Period		0x00	Reg5
Rdy_Tbl								0x01	
Wait_Tbl								0x02	

Data								Para	
[63..56]	[55..48]	[47..40]	[39..32]	[31..24]	[23..16]	[15..8]	[7..0]		
T_Free	ECB	Bity	Bitx	Y	X	T_Name	Prio	0x00	Reg6
					S_I_R	OS_Ctx	Rdy_Grp	0x01	
						E_Free	Wait_Grp	0x02	

Table 13-26: HWRTOS Test Info Read

Data: The content of these fields depends on the user request as follows:

When 'para' in HWRTOS test request is equal to "0x00", then read registers contain TCB of the selected task as follow:

Reg4:

Task_Ctx: Number of context switch occurred to the selected task.

Msg: Pointer to the message received from a mailbox by the selected task.

Reg5:

Period: The period of the selected task (if the task is not periodic then this field is equal to 0x00).

Dec_Period: The current value of the selected task period (decremented by one each tick-timer tick).

Id: Task ID.

Stat: Task status and it could be one of the following:

0x00: Ready to run or running.

0x01: Pend on semaphore.

0x02: Pend on mailbox.

Stat_P: Task pend status and it could be one of the following:

0x00: Pending status ok.

0x01: Pending timed out.

Reg6:

Prio: Priority of the selected task.

T_Name: Name of the selected task.

X, Y, Bitx and Bity: Pre-calculated values which are used in lists manipulation.

ECB: If the selected task is waiting for an event (semaphore or mailbox), then this is the ECB ID of that event. It is equal to 0x00 if the task is not waiting for any event.

T_Free: Bit indication to show if the TCB is free or used.

When 'para' in HWRTOS test request is equal to "0x01", then read registers contain some system parameters and ready list as follow:

Reg4:

OS_Time: The system time starting from OS_Start call and it increments by one with each tick-timer tick.

Prio_Cur: Priority of the current task.

Prio_H: Priority of the highest priority task.

Prio_Idle: Priority of the idle task.

Task_Cnt: Number of tasks in the system.

Reg5:

Rdy_Tbl: First variable of the system ready list, for more information see ready list discussion on Page 19.

Reg6:

Rdy_Grp: Second variable of the system ready list, for more information see ready list discussion on Page 19.

OS_Ctx: Number of the context switch occurred in the system.

S_I_R: The three least bits of this byte indicates the following:

R: Bit 0 of this byte and indicates the status of the OS (R=1 means OS is running, R=0 means OS is not running).

I: Bit 1 of this byte and indicates the status of the HWRTOS interrupt output (I=1 means interrupt is active, I=0 means interrupt is not active).

S: Bit 2 of this byte and indicates the status of the software interrupt test (S=1 means SW interrupt is active, S=0 means SW interrupt is not active).

When 'para' in HWRTOS test request is equal to "0x02", then read registers contain ECB of the selected event as follow:

Reg4:

Ptr: Pointer to the received message in case of mailbox.

Cnt: semaphore count in case of semaphore.

Type: ECB type (semaphore or mailbox).

E_Name: ECB name.

Reg5:

Wait_Tbl: First variable of the task waiting list.

Reg6:

Wait_Grp: Second variable of the task waiting list.

E_Free: Bit indication to show if the ECB is free or used.

Stop-Watch Timer Start Request:

Stop-watch timer start request is used to start the 32-bit stop-watch hardware timer. To read the timer value a stop-watch timer stop request should be issued. Table 13-27 shows the fields.

Arguments	Command	Reg0
[63..8]	[7..0]	
	0x11	

Table 13-27: Stop-Watch Timer Start Request

Command type: Request to start the stop-watch hardware timer (0x11).

Arguments: None.

Stop-Watch Timer Stop Request:

Stop-watch timer stop request is used to stop the 32-bit stop-watch hardware timer. Calling this function will stop the timer and copy the timer count to read register (Reg4) to be read by the application, and then resets the timer. Table 13-28 shows the fields.

Arguments	Command	
[63..8]	[7..0]	Reg0
	0x12	

Table 13-28: Stop-Watch Timer Stop Request

Command type: Request to stop the stop-watch hardware timer (0x12).

Arguments: None.

Stop-Watch Timer Stop Information Read:

Stop-watch timer stop information read follows stop-watch timer stop request to read the timer count. Table 13-29 shows the fields returned following a stop-watch timer stop request.

Data		
[63..32]	[31..0]	Reg4
	Timer_Cnt	

Table 13-29: Stop-watch Timer Stop Info Read

Data:

Timer_Cnt: Current value of the timer count.

Continuous Timer Start Request:

Continuous timer start request is used to start the 32-bit continuous hardware timer. This timer can be read dynamically. Table 13-30 shows the fields.

Arguments	Command	
[63..8]	[7..0]	Reg0
	0x13	

Table 13-30: Continuous Timer Start Request

Command type: Request to start the continuous hardware timer (0x13).

Arguments: None.

Continuous Timer Read Request:

Continuous timer read request is used to read the count of the 32-bit continuous hardware timer. This function takes place dynamically without the need to stop the timer, and copies the count to read register (Reg4) to be read by the application. Table 13-31 shows the fields.

Arguments	Command	Reg0
[63..8]	[7..0]	
	0x14	

Table 13-31: Continuous Timer Read Request

Command type: Request to read the count of the continuous hardware timer (0x14).

Arguments: None.

Continuous Timer Information Read:

Continuous timer information read follows continuous timer read request to read the timer count. Table 13-32 shows the fields returned following a continuous timer read request.

Data		Reg4
[63..32]	[31..0]	
	Timer_Cnt	

Table 13-32: Continuous Timer Info Read

Data:

Timer_Cnt: Current value of the timer count.

Continuous Timer Stop Request:

Continuous timer stop request is used to stop the 32-bit continuous hardware timer. Calling this function will stop and reset the timer. Table 13-33 shows the fields.

Arguments	Command	Reg0
[63..8]	[7..0]	
	0x15	

Table 13-33: Continuous Timer Stop Request

Command type: Request to stop the continuous hardware timer (0x15).

Arguments: None.

13.3 Appendix C: HWRTOS Flow Charts

Flowcharts for major services implemented in the HWRTOS are presented here:

- The main steps are shown in the flowcharts, not all details.
- Arguments check of each function and context switch are done in the modified SW RTOS.

Create Task:

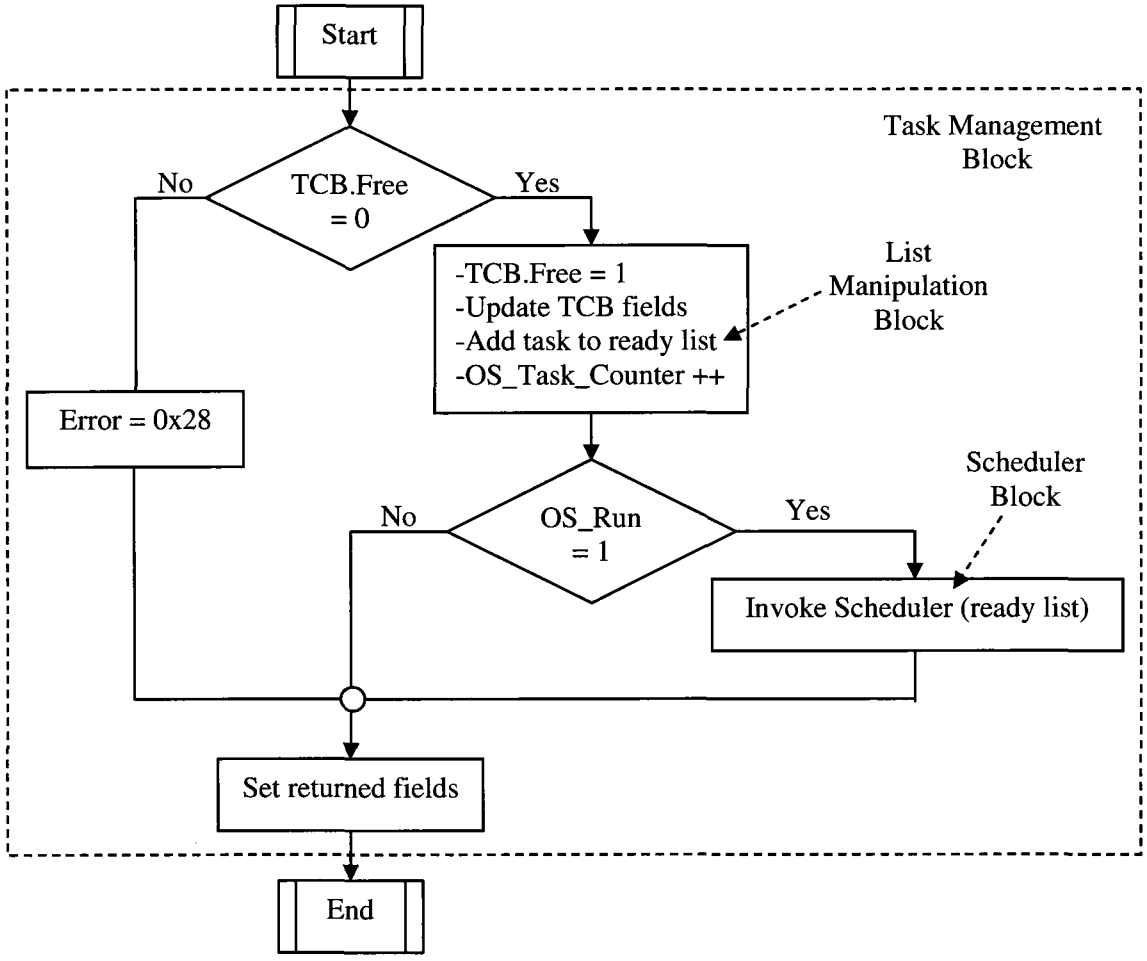


Figure 13-1: Create Task Flowchart

Tick-Timer:

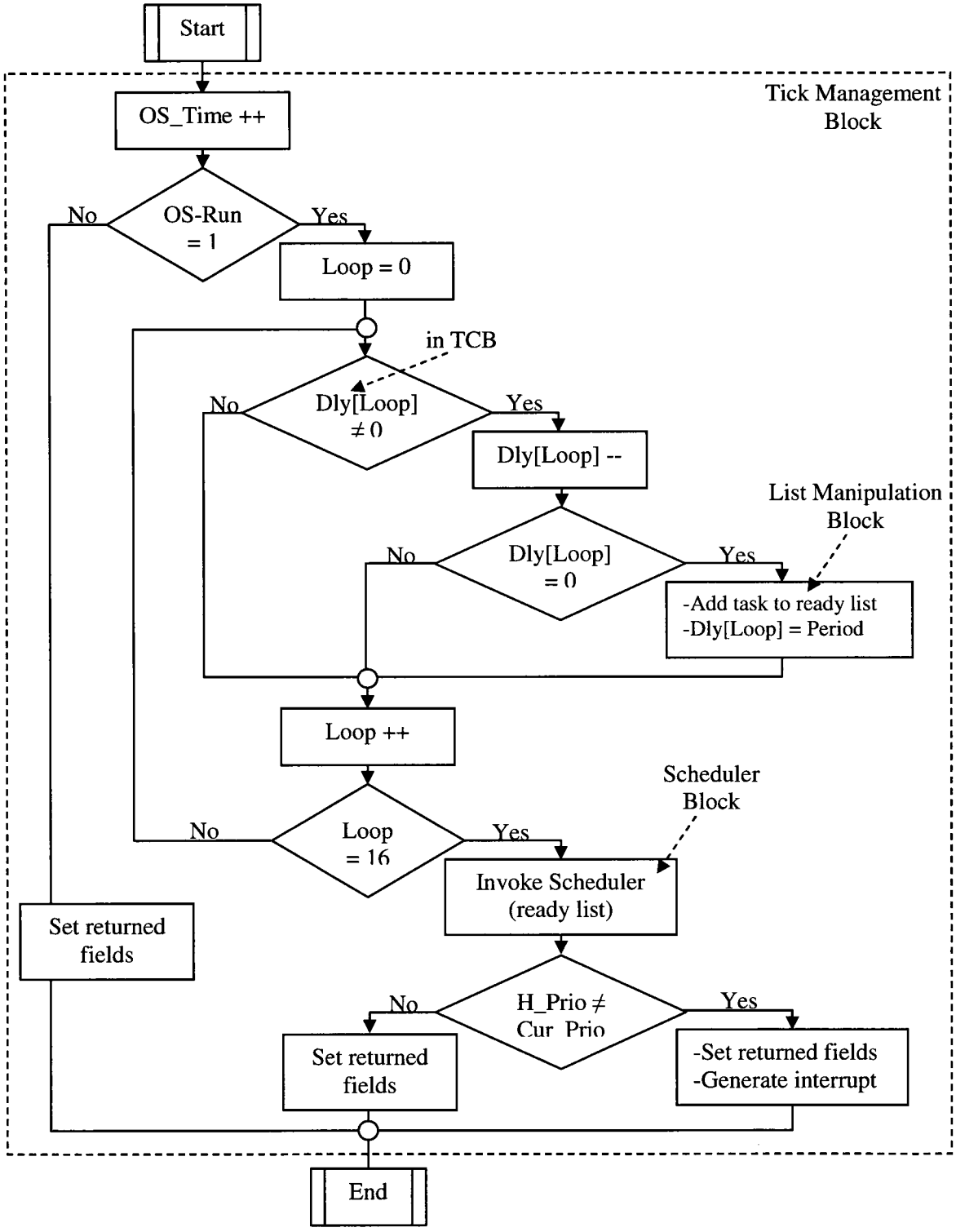


Figure 13-2: Tick-Timer Flowchart

OS Start:

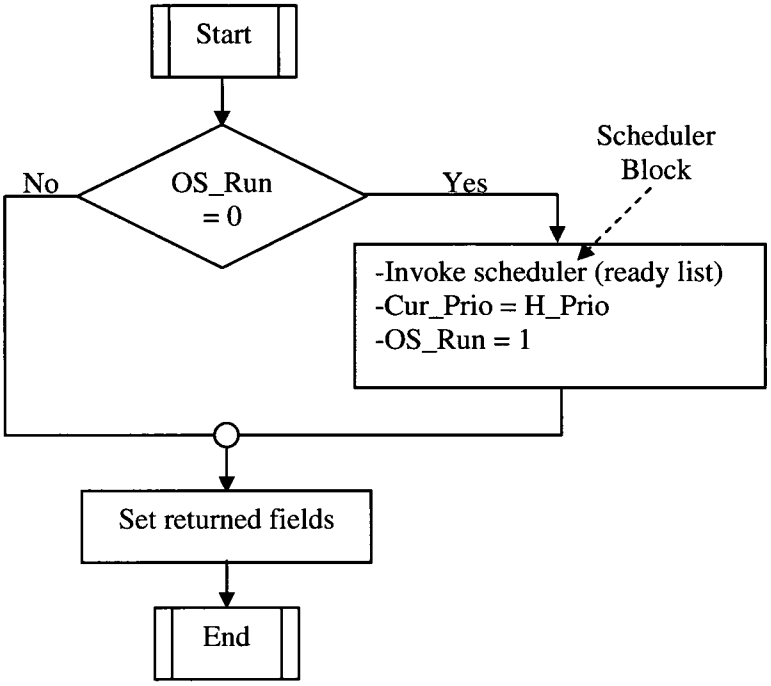


Figure 13-3: OS Start Flowchart

Task Done:

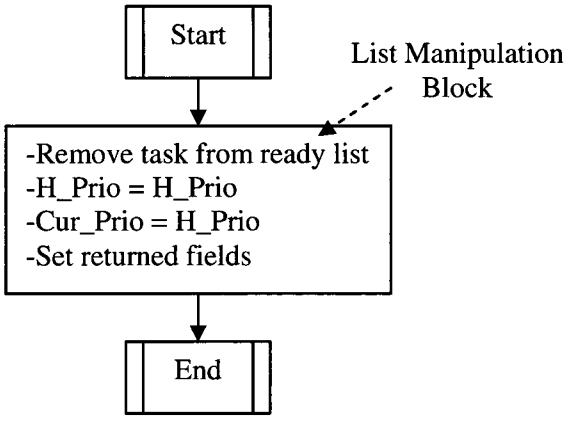


Figure 13-4: Task Done Flowchart

Create Semaphore Flowchart:

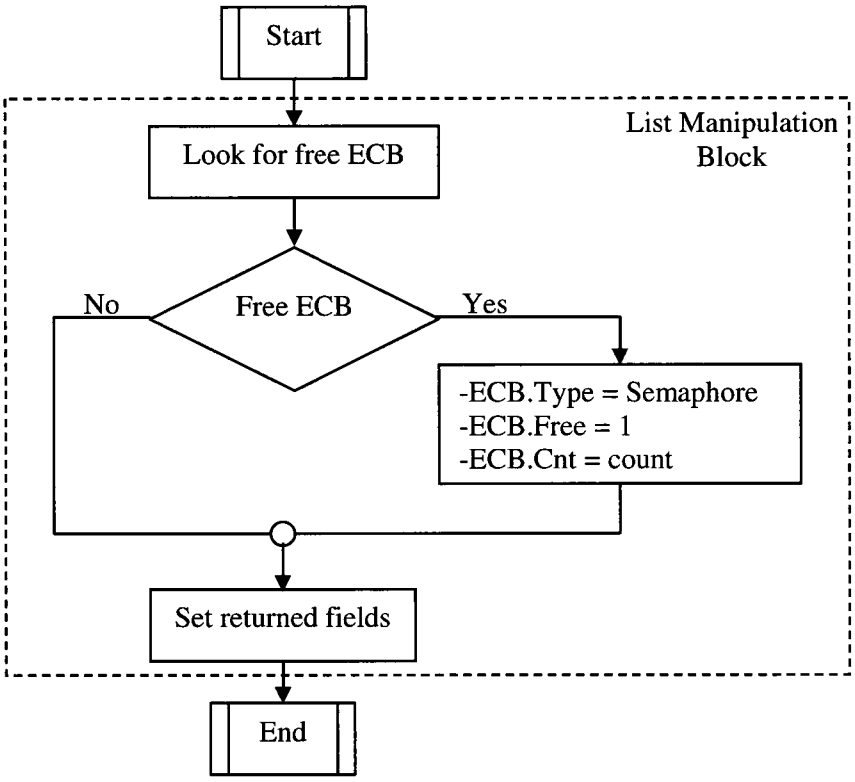


Figure 13-5: Create Semaphore Flowchart

Semaphore Pend Flowchart:

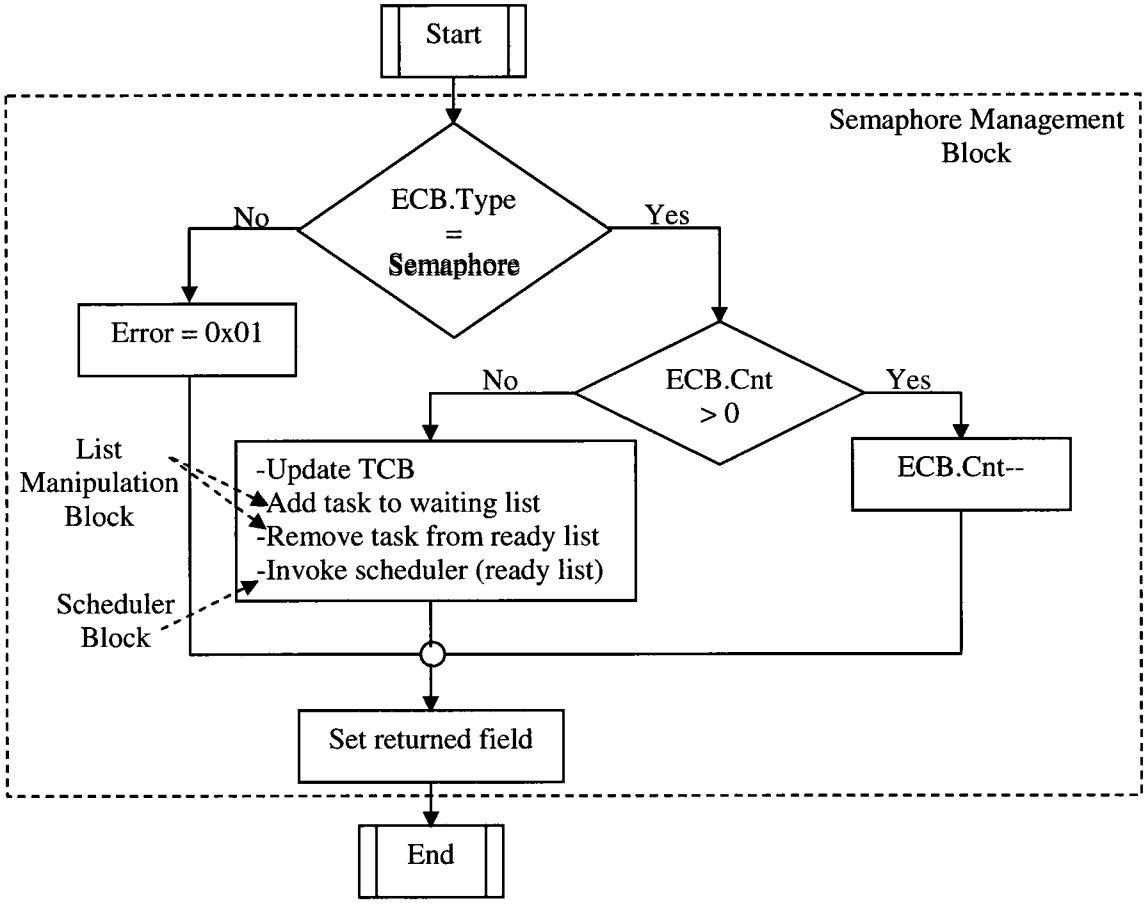


Figure 13-6: Semaphore Pend Flowchart

Semaphore Post Flowchart:

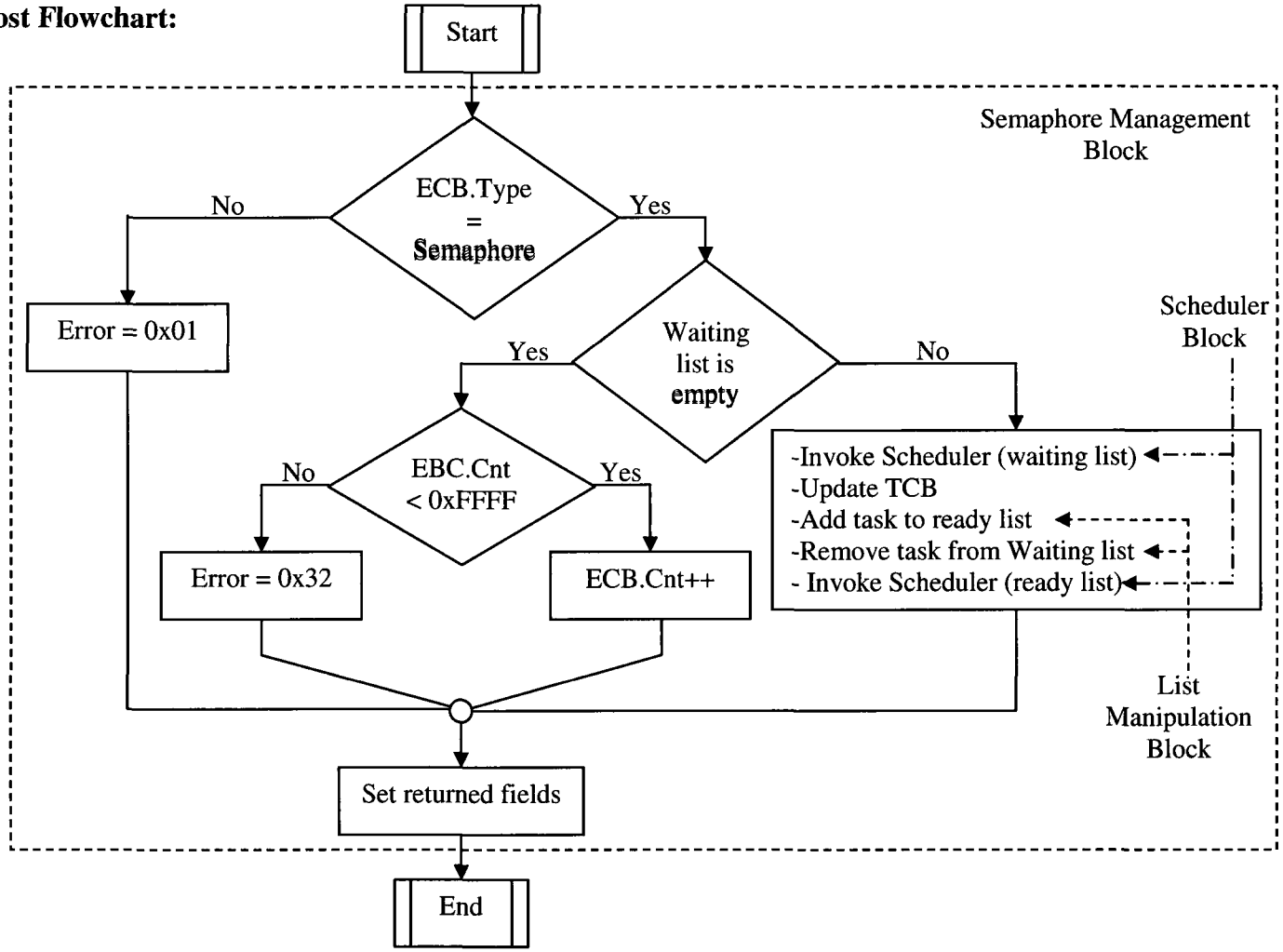


Figure 13-7: Semaphore Post Flowchart

Create Mailbox Flowchart:

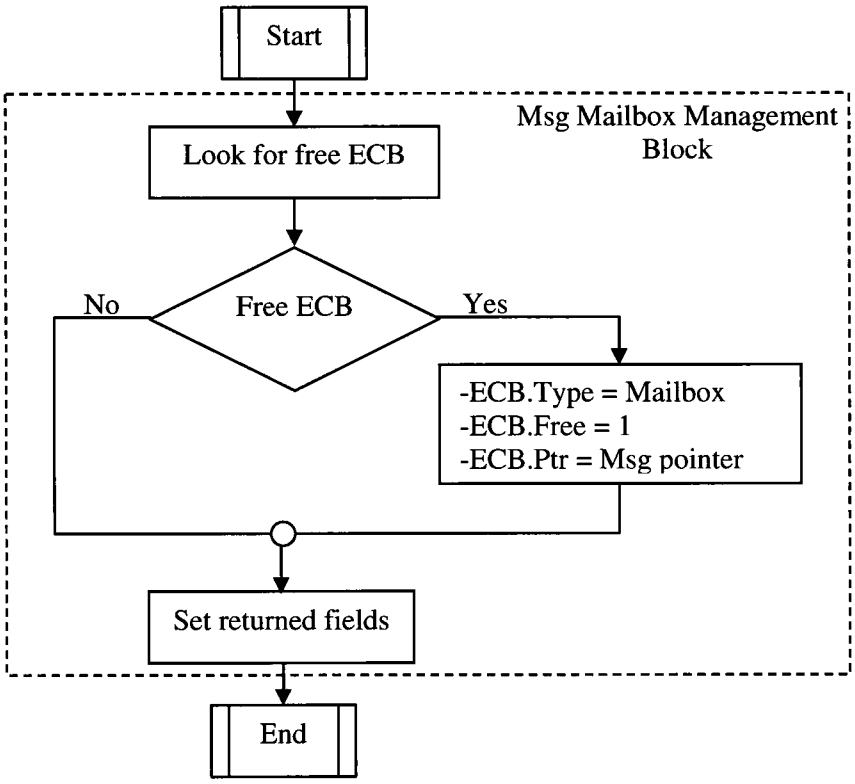


Figure 13-8: Create Mailbox Flowchart

Message Mailbox Pend Flowchart:

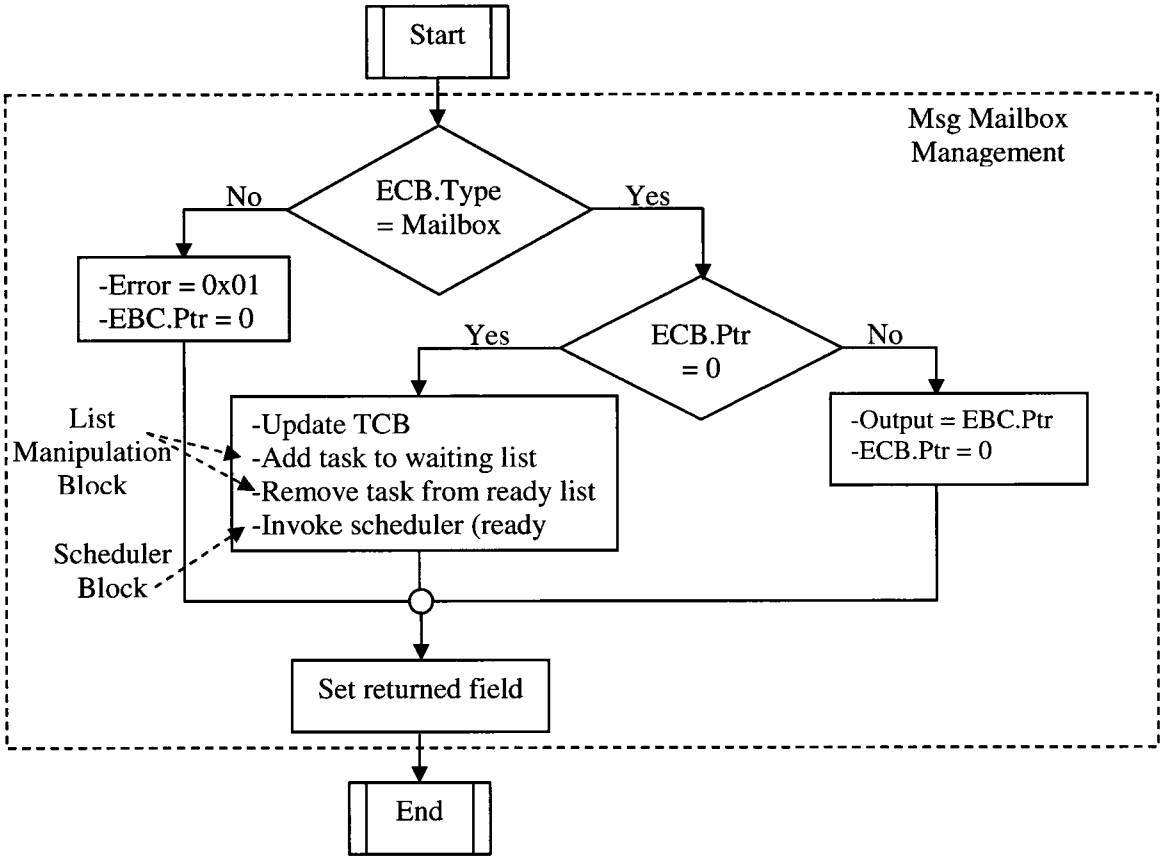


Figure 13-9: Message Mailbox Pend Flowchart

Message Mailbox Post Flowchart:

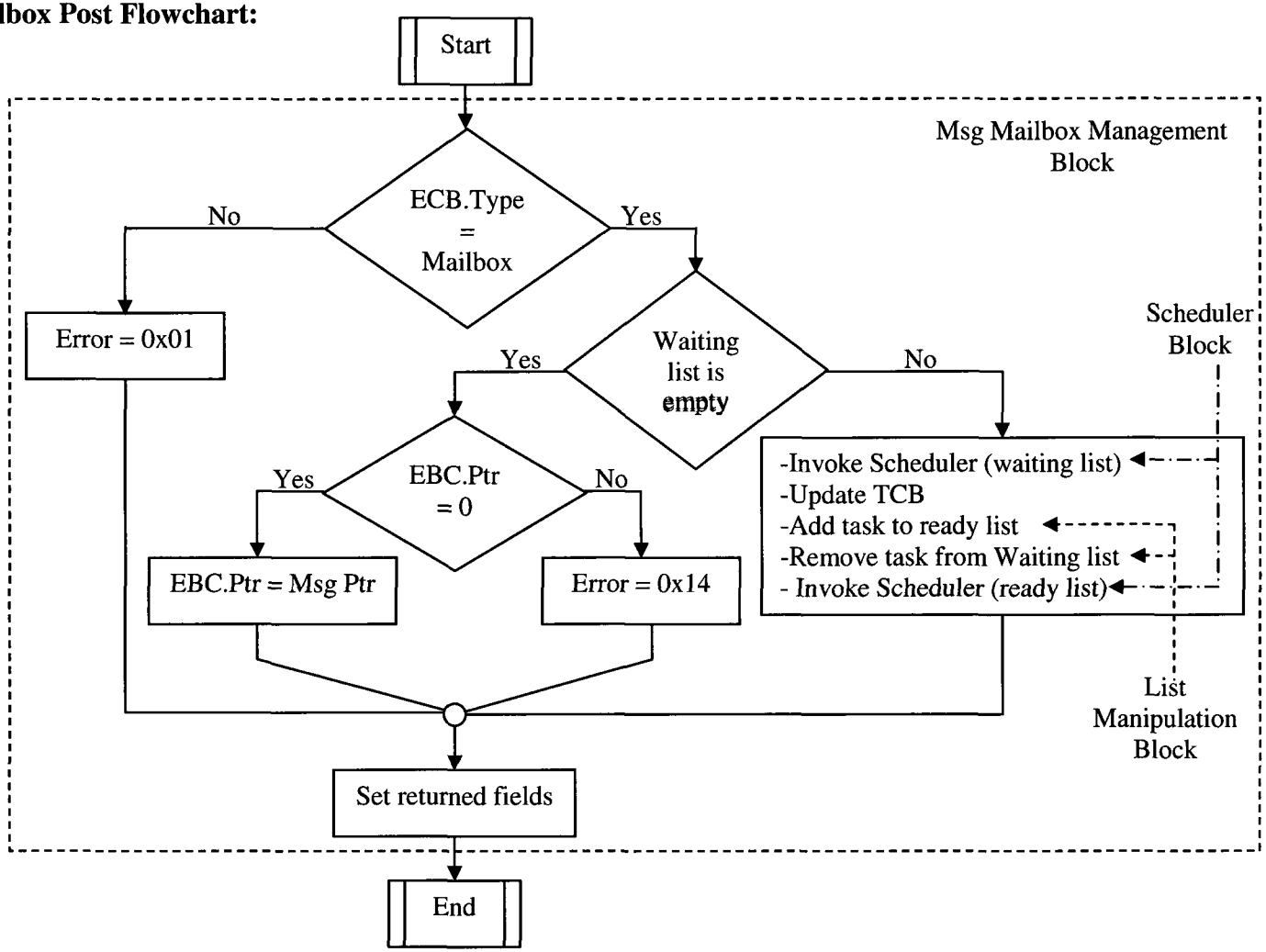


Figure 13-10: Message Mailbox Post Flowchart

13.4 Appendix D: Experiments Data

SW RTOS					
OS Tick-Timer (tick/sec)	**1 Tick Every (msec)**	**Number of Tasks (tasks)**	**Execution Time (ticks)**	**Execution Time (sec)**	**Jitter (µsec)**
100	10	2	83169340	1.03962	20.738
			83170999	1.03964	
		4	83201191	1.04001	00.413
			83201224	1.04002	
		8	83255149	1.04069	31.400
			83257661	1.04072	
		10	83283919	1.04105	34.338
			83286666	1.04108	
1000	1	2	84745212	1.05932	21.538
			84746935	1.05934	
		4	85060631	1.06326	24.7125
			85062608	1.06328	

SW RTOS					
OS Tick-Timer (tick/sec)	**1 Tick Every (msec)**	**Number of Tasks (tasks)**	**Execution Time (ticks)**	**Execution Time (sec)**	**Jitter (µsec)**
1000	1	8	85633722	1.07042	31.075
			85636208	1.07045	
		10	85928833	1.07411	34.550
			85931597	1.07414	
2000	0.500	2	86579118	1.08224	22.937
			86580953	1.08226	
		4	87216417	1.09021	24.625
			87218387	1.09023	
		8	88439721	1.10550	00.300
			88439745	1.10550	
		10	89086378	1.11358	34.963
			89089175	1.11361	

SW RTOS					
OS Tick-Timer (tick/sec)	**1 Tick Every (msec)**	**Number of Tasks (tasks)**	**Execution Time (ticks)**	**Execution Time (sec)**	**Jitter (µsec)**
4000	0.250	2	90414647	1.13018	20.700
			90416303	1.13020	
		4	91847658	1.14810	24.500
			91849618	1.14812	
		8	94785936	1.18482	32.000
			94788496	1.18486	
		10	96430366	1.20538	35.150
			96433178	1.20541	

Table 13-34: SW RTOS Tick-Timer Data

SW/HW RTOS					
OS Tick-Timer (tick/sec)	**1 Tick Every (msec)**	**Number of Tasks (tasks)**	**Execution Time (ticks)**	**Execution Time (sec)**	**Jitter (µsec)**
100	10	2	83000041	1.03750	0
		4	83000041	1.03750	0
		8	83000041	1.03750	0
		10	83000041	1.03750	0
1000	1	2	83000041	1.03750	0
		4	83000041	1.03750	0
		8	83000041	1.03750	0
		10	83000041	1.03750	0
2000	0.500	2	83000041	1.03750	0
		4	83000041	1.03750	0
		8	83000041	1.03750	0
		10	83000041	1.03750	0
4000	0.250	2	83000041	1.03750	0
		4	83000041	1.03750	0
		8	83000041	1.03750	0
		10	83000041	1.03750	0

Table 13-35: SW/HW RTOS Tick-Timer Data

OS Tick-Timer (tick/sec)	Number of Tasks (tasks)	Number of Ticks during Task Execution Period (ticks)	Overhead (SW RTOS – SW/HW RTOS) (msec)	Overhead per Tick (µsec)
100	2	103	2.11624	20.54600
	4		2.51437	24.41141
	8		3.18885	30.95971
	10		3.54847	34.45121
1000	2	1037	21.81464	21.03629
	4		25.75738	24.83836
	8		32.92101	31.74640
	10		36.60990	35.30366
2000	2	2075	44.73846	21.56070
	4		52.70470	25.39986
	8		67.99600	32.76916
	10		76.07921	36.66468
4000	2	4150	92.68258	22.33315
	4		110.59521	26.64945
	8		147.32369	35.49968
	10		167.87906	40.45279

Table 13-36: Tick-Timer Overhead

	SW RTOS Overhead (ticks)	SW RTOS Overhead (µsec)	SW/HW RTOS Overhead (ticks)	SW/HW RTOS Overhead (µsec)
Context switch @ critical instant	681	8.5125	926	11.575
Context switch due to preemption	1415	17.6875	1535	19.1875
Context switch due to task completes	1409	17.6125	1363	17.0375

Table 13-37: Overhead due to Context Switches

Task	SW RTOS		SW/HW RTOS	
	Begin of Task (µsec)	Calculate Delay (µsec)	Begin of Task (µsec)	Calculate Delay (µsec)
Task 1	32.8	7.7	33.4	-
Task 2	0.1	7.7	0.1	-
Task 3	0.1	7.7	0.1	-

Table 13-38: Overhead due to Codes at Beginning of Tasks and Calculating Delay for Case Study

13.5 Appendix E: Comparison Graphs

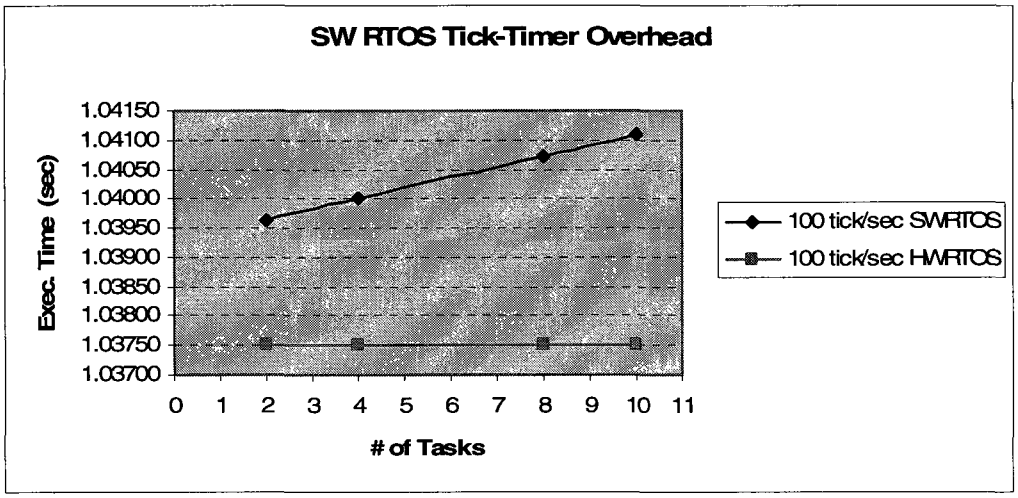


Figure 13-11: SW RTOS Tick-Timer Overhead (100 ticks/sec)

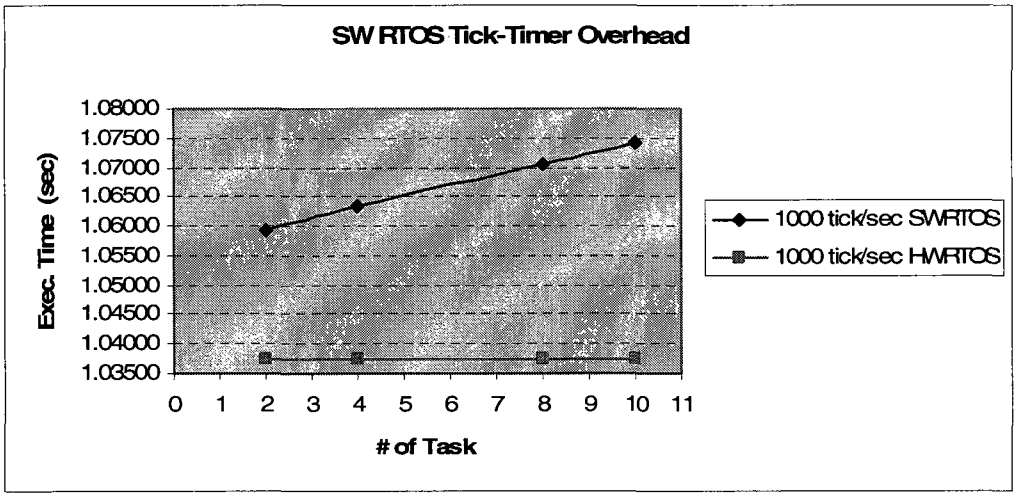


Figure 13-12: SW RTOS Tick-Timer Overhead (1000 ticks/sec)

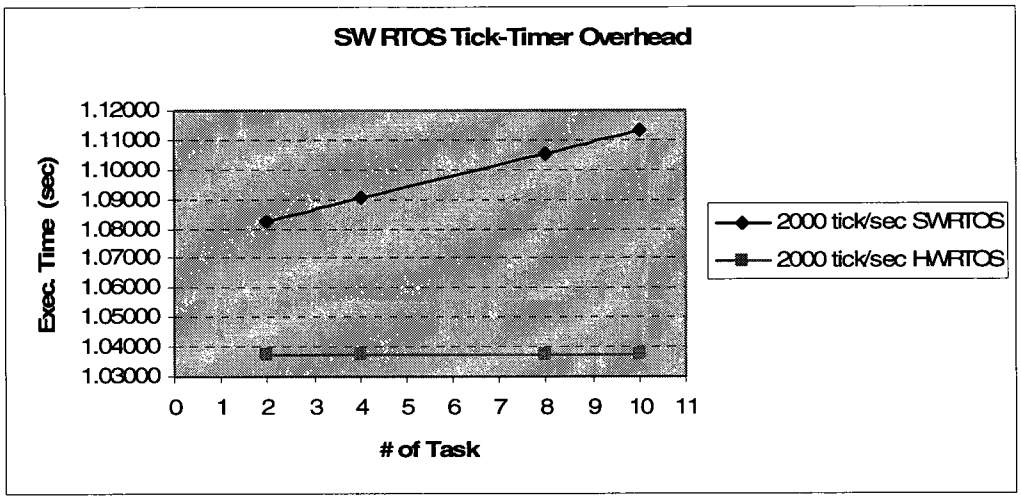


Figure 13-13: SW RTOS Tick-Timer Overhead (2000 ticks/sec)

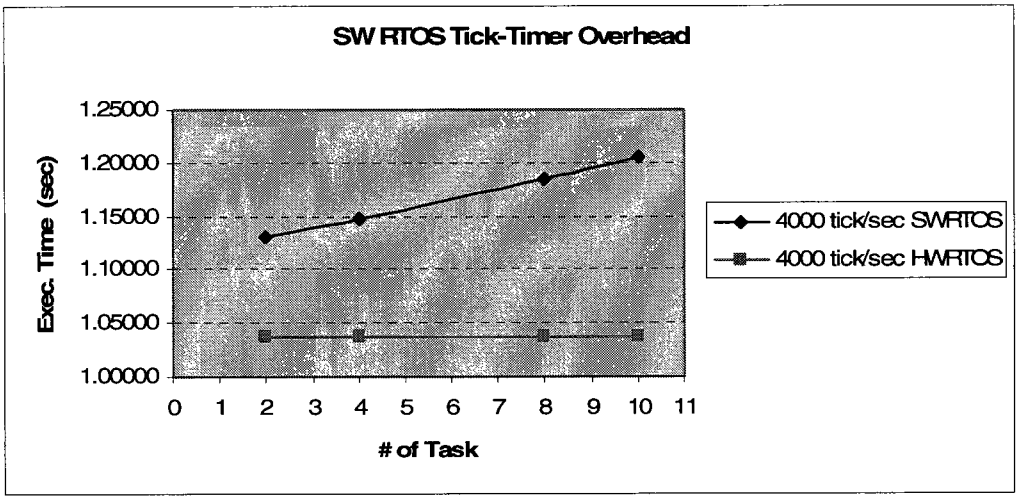


Figure 13-14: SW RTOS Tick-Timer Overhead (4000 ticks/sec)

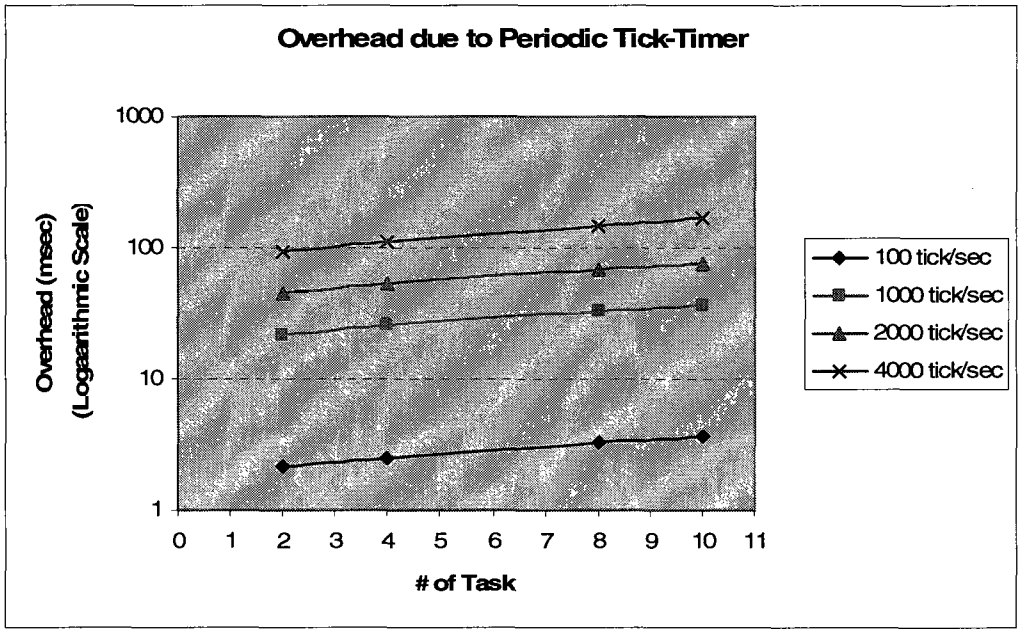


Figure 13-15: Overhead due to Periodic Tick-Timer

13.6 Appendix F: Calculations

Standard Response Time Analysis:

$$R_i = e_i + \sum_{j=0}^{i-1} \left\lceil \frac{R_i}{p_j} \right\rceil e_j$$

For T_1: $R_1^0 = e_1 = 2.5$ sec (this is the highest priority task so it is never preempted)

$(R_1 = 2.5) < (d_1 = 3.75)$ => Task 1 is ok

For T_2: $R_2^0 = e_2 = 1.25$ sec

$$R_2^1 = e_2 + \left\lceil \frac{R_2^0}{p_1} \right\rceil e_1 = 1.25 + \left\lceil \frac{1.25}{3.75} \right\rceil 2.5 = 3.75 \text{ sec}$$

$$R_2^2 = e_2 + \left\lceil \frac{R_2^1}{p_1} \right\rceil e_1 = 1.25 + \left\lceil \frac{3.75}{3.75} \right\rceil 2.5 = 3.75 \text{ sec}$$

$(R_2 = 3.75) < (d_2 = 7.5)$ => Task 2 is ok

For T_3 : $R_3^0 = e_3 = 1.25$ sec

$$R_3^1 = e_3 + \left\lceil \frac{R_3^0}{p_1} \right\rceil e_1 + \left\lceil \frac{R_3^0}{p_2} \right\rceil e_2 = 1.25 + \left\lceil \frac{1.25}{3.75} \right\rceil 2.5 + \left\lceil \frac{1.25}{7.5} \right\rceil 1.25 = 5 \text{ sec}$$

$$R_3^2 = e_3 + \left\lceil \frac{R_3^1}{p_1} \right\rceil e_1 + \left\lceil \frac{R_3^1}{p_2} \right\rceil e_2 = 1.25 + \left\lceil \frac{5}{3.75} \right\rceil 2.5 + \left\lceil \frac{5}{7.5} \right\rceil 1.25 = 7.5 \text{ sec}$$

$$R_3^3 = e_3 + \left\lceil \frac{R_3^2}{p_1} \right\rceil e_1 + \left\lceil \frac{R_3^2}{p_2} \right\rceil e_2 = 1.25 + \left\lceil \frac{7.5}{3.75} \right\rceil 2.5 + \left\lceil \frac{7.5}{7.5} \right\rceil 1.25 = 7.5 \text{ sec}$$

$(R_3 = 7.5) < (d_3 = 15)$ => Task 3 is ok

The final result is that all tasks are schedulable for the case study.

Response Time Analysis including System Overhead for SW RTOS:

For number of OS ticks = 100 tick/sec:

O_{tick} = 1.9 x 3 + 20 = 25.7 µsec

For T_1: $R_1^0 = e_1 + O\_ctx_{ci} + E\_start_1 + O\_cal_1 + O\_ctx_{te}$

$R_1^0 = 2500 + 0.0085 + 0.0328 + 0.0077 + 0.0176 = 2.50007$ sec

$$R_1^1 = e_1 + O\_ctx_{ci} + E\_start_1 + O\_cal_1 + O\_ctx_{te} + \left\lceil \frac{R_1^0}{p_{tick}} \right\rceil O_{tick}$$

$$R_1^1 = 2500 + 0.0085 + 0.0328 + 0.0077 + 0.0176 + \left\lceil \frac{2500.07}{10} \right\rceil 0.0257 = 2.50652 \text{ sec}$$

$$R_1^2 = e_1 + O\_ctx_{ci} + E\_start_1 + O\_cal_1 + O\_ctx_{te} + \left\lceil \frac{R_1^1}{p_{tick}} \right\rceil O_{tick}$$

$$R_1^2 = 2500 + 0.0085 + 0.0328 + 0.0077 + 0.0176 + \left\lceil \frac{2506.52}{10} \right\rceil 0.0257 = 2.50652 \text{ sec}$$

(R_1 = 2.50652) < (d_1 = 3.75) => Task 1 is ok

For T_2: $R_2^0 = e_2 + O\_ctx_{ci} + E\_start_2 + O\_cal_2 + O\_ctx_{te}$

$R_2^0 = 1250 + 0.0085 + 0.0001 + 0.0077 + 0.0176 = 1.25003$ sec

$$R_2^1 = e_2 + O\_ctx_{ci} + E\_start_2 + O\_cal_2 + O\_ctx_{te} + \left\lceil \frac{R_2^0}{p_{tick}} \right\rceil O_{tick} + \left\lceil \frac{R_2^0}{p_1} \right\rceil \left(e_1 + O\_ctx_{pre} + O\_cal_1 + O\_ctx_{te}\right) + E\_start_1$$

$= 3.75335$ sec

$R_2^2 = 6.25982$ sec, $R_2^3 = 6.26624$ sec, $R_2^4 = 6.26627$ sec, $R_2^5 = 6.26627$ sec

$(R_2 = 6.26627) < (d_2 = 7.5)$ => Task 2 is ok

For T_3: $R_3^0 = e_3 + O\_ctx_{ci} + E\_start_3 + O\_cal_3 + O\_ctx_{te}$

$R_3^0 = 1250 + 0.0085 + 0.0001 + 0.0077 + 0.0176 = 1.25003$ sec

$$R_3^1 = e_3 + O\_ctx_{ci} + E\_start_3 + O\_cal_3 + O\_ctx_{te} + \left\lceil \frac{R_3^0}{p_{tick}} \right\rceil O_{tick}$$

$$+ \left\lceil \frac{R_3^0}{p_1} \right\rceil \left(e_1 + O\_ctx_{pre} + O\_cal_1 + O\_ctx_{te}\right) + E\_start_1 \qquad = 5.00339 \text{ sec}$$

$$+ \left\lceil \frac{R_3^0}{p_2} \right\rceil \left(e_2 + O\_ctx_{pre} + O\_cal_2 + O\_ctx_{te}\right) + E\_start_2$$

$R_3^2 = 7.51307$ sec, $R_3^3 = 11.26961$ sec, $R_3^4 = 13.77929$ sec, $R_3^5 = 13.78574$ sec

$R_3^6 = 13.78576$ sec, $R_3^7 = 13.78576$ sec

$(R_3 = 13.78576) < (d_3 = 15)$ => Task 3 is ok => all tasks are schedulable

For number of OS ticks = 4000 tick/sec and following the same steps above:

For T_1: $(R_1 = 2.78655) < (d_1 = 3.5)$ => Task 1 is ok

For T_2 : $(R_2 = 6.96631) < (d_2 = 7.5)$ => Task 2 is ok

For T_3: $(R_3 = 22.2921) > (d_3 = 15)$ => Task 3 is not ok

So, the task set is not schedulable when number of OS ticks = 4000 tick/sec

Response Time Analysis including System Overhead for SW/HW RTOS:

For T_1: $R_1^0 = e_1 + O\_ctx_{ci} + E\_start_1 + O\_ctx_{te}$

$R_1^0 = 2500 + 0.0116 + 0.0334 + 0.0170 = 2.50006$ sec

$(R_1 = 2.50006) < (d_1 = 3.75) \Rightarrow$ Task 1 is ok

For T_2 : $R_2^0 = e_2 + O\_ctx_{ci} + E\_start_2 + O\_ctx_{te}$

$R_2^0 = 1250 + 0.0116 + 0.0001 + 0.0170 = 1.25003$ sec

$$R_2^1 = e_2 + O\_ctx_{ci} + E\_start_2 + O\_ctx_{te} + \left\lceil \frac{R_2^0}{p_1} \right\rceil \left(e_1 + O\_ctx_{pre} + O\_ctx_{te}\right) + E\_start_1 = 3.7501 \text{ sec}$$

$R_2^2 = 6.25013$ sec, $R_2^3 = 6.25013$ sec

$(R_2 = 6.25013) < (d_2 = 7.5) \Rightarrow$ Task 2 is ok

For T_3 : $R_3^0 = e_3 + O\_ctx_{ci} + E\_start_3 + O\_ctx_{te}$

$R_3^0 = 1250 + 0.0116 + 0.0001 + 0.0170 = 1.25003$ sec

$$R_3^1 = e_3 + O\_ctx_{ci} + E\_start_3 + O\_ctx_{te} + \left\lceil \frac{R_3^0}{p_1} \right\rceil \left(e_1 + O\_ctx_{pre} + O\_ctx_{te}\right) + E\_start_1 = 5.00013 \text{ sec}$$

$$+ \left\lceil \frac{R_3^0}{p_2} \right\rceil \left(e_2 + O\_ctx_{pre} + O\_ctx_{te}\right) + E\_start_2$$

$R_3^2 = 7.50017$ sec, $R_3^3 = 11.25024$ sec, $R_3^4 = 13.75028$ sec, $R_3^5 = 13.75028$ sec

$(R_3 = 13.75028) < (d_3 = 15) \Rightarrow$ Task 3 is ok $\Rightarrow$ all tasks are schedulable