Institutionen för systemteknik

Department of Electrical Engineering

Examensarbete

The Real-Time Multitask Threading Control

Master thesis performed in Computer Engineering *division*

by

Shuang Han

LiTH-ISY-EX--07/4132--SE

Linköping, 2007

The Real-Time Multitask Threading Control

Master thesis in Computer Engineering

at the Department of Electrical Engineering

Linköping Institute of Technology

by

Shuang Han

LiTH-ISY-EX--07/4132--SE

Supervisor: Professor Dake Liu

Examiner: Professor Dake Liu

Linköping December 5, 2007

Presentation Date

2007-12-05

Publishing Date (Electronic version)

Department and Division

Computer Engineering Division

Department of Electrical Engineering

Language	**Type of Publication**
x English	Licentiate thesis
___ Other (specify below)	X Degree thesis
	___ Thesis C-level
	___ Thesis D-level
	___ Report
	Other (specify below)

Number of Pages

63

ISBN (Licentiate thesis)

ISRN:

LiTH-ISY-EX--07/4132--SE

Title of series (Licentiate thesis) _______

URL, Electronic Version

http://www.ep.liu.se

Publication Title

The Real-Time Multitask Threading Control

Author(s)

Shuang Han

Abstract

In this master thesis, we design and implemented a super mode for multiple streaming signal processing applications, and got the timing budget based on Senior DSP processor. This work presented great opportunity to study the real-time system and firmware design knowledge on embedded system.

Keywords

Firmware, RTOS, RMA, Super Mode, Offline Optimizer

Abstract

In this master thesis, we design and implemented a super mode for multiple streaming signal processing applications, and got the timing budget based on Senior DSP processor. This work presented great opportunity to study the real-time system and firmware design knowledge on embedded system.

Acknowledgements

I would like to express my appreciation to my supervisor Professor Dake Liu for the helps he gave me during this thesis going and a lot of invaluable experience on how to conduct a project in industry work way. In addition, gratefully thanks for the Phd student Di Wu as he helped me to solve a lot of problems and gave me good suggestions in my study.

I also wish to thank all the friends in Linköping for all the good time we shared, wish the friendship would go on forever.

Last but not least, thanks to my parents for all the love they gave. I love you as all I have.

Shuang Han
Linköping, Dec 05

Contents

Glossary

ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuits
CODEC	Coder-Decoder
DSP	Digital signal Processing
EDF	Earliest Deadline First algorithm
ELF	Executable and Linkable Format
EOI	End of Interrupt
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
ISR	Interrupt Service Routine
MAC	Multiplication and Accumulation unit
MMU	Memory Management Unit
OO	Offline Optimizer
PCB	Process Control Block
RM	Rate Monotonic algorithm
RTP	Real-Time Process
RTOS	Real-Time Operating System
TC	Threading Controller
TCB	Thread Control Block
WCRT	Worst Case Run Time

Chapter 1

Introduction

1.1 Background

With the fast development of telecommunication technologies, numbers of functions such as multi-applications or multi-connections are integrated into a modern handset or a terminal, which includes radio baseband processing (download mode), voice codec, digital camera, audio mp3, video, web message, and so on. All these functions are implemented as application specific instruction set processors or ASIC modules. Furthermore, as the most important part in a modern handset, the communication modems and play CODEC systems are real-time subsystems. So a real-time multi-task control subsystem for the DSP subsystem in a modern handset or a terminal should be put more attention as same as the other function modules.

1.2 Purpose of this thesis

The purpose of this project is to design and implement a real-time multi-task threading control subsystem based on static scheduling for a DSP subsystem. Though the project, to better understand how to design and implement the real time applications on a DSP processor, furthermore, to master the skills of design the firmware of embedded systems from top to bottom.

1.3 Reading guidelines

In chapter 2 and 3, some basic topics around DSP theories and real-time system design are introduced. Meanwhile, the firmware design flow of embedded system where we discussed in detail is another emphasis of this project. For those who are familiar with DSP theories and real-time system design can skip to chapter 4.

Chapter 4 gave a brief survey of three commercial RTOS system that can be found in the market in 2006-2007. Study on the features of these RTOS was also carried out.

Chapter 5 is the implementation of real-time multitask threading control subsystem as a DSP super mode. The definition of supermode will be found in Chapter 5. From the function specification to detailed implementation of each function in the real-time multitask threading control subsystem, the complete subsystem implementation was explained step by step. The super mode was implemented using C++, and timing and functional critical machine dependent functions were implemented in assembly language.

In Chapter 6 the implementation of an Offline Optimizer was described. The offline optimizer is used for checking the timing performance of the super mode. By running the Offline Optimizer, the static scheduling of the tasks to be controlled by the super mode got better refined and timing performance is guaranteed by WCRT static scheduling.

Chapter 2

DSP firmware design flow

Firmware is a computer program that is embedded in a hardware device either via hard-coded logic, a microcontroller, a field-programmable gate array (FPGA), or even an on-board processor. So when we talk about firmware, we implicitly meant as firmware in embedded systems. There are two sorts of firmware---application firmware or system firmware. The former can be a voice codec in a mobile phone, which can be recognized by its user. On the other hand, firmware can also implicitly exist in a system and can not be discovered by its user, e.g. the firmware for radio baseband signal processing. The later kind of firmware is mainly charged with the management of the using of system hardware and running of application software. Real-time operating system (RTOS) is a kind of typical system firmware.

Deep understanding of DSP processor is the base for qualified DSP Firmware design. When we run a project, the time is always limited, so for the designer the good knowledge of hardware and enough understanding of algorithms and applications will be necessary. Here to understand application means to know three things: *what, why,* and *how.* “What” stand for understanding what is the algorithm and the application, including what are inputs, what are outputs, what is the function that the inputs be processed. “Why” stands for understanding the reason to choose such algorithms or applications, also including the market decision, the technology decision, and the product strategy. “How” means how to implement the project, which includes the algorithm design, the hardware design, the system design and optimizations, as well fabrication.

There are two main constraints when we design the DSP firmware, the real-time constraints of the application and the hardware constraints, for example, the timing period of the input data packet, the performance and the hardware precision. Correct implementation of DSP firmware is not only including the correctness of functionality, also including the correctness of the runtime behavior, and correct handling of precision and dynamic range for finite-length computing and storage hardware. Following figure give an intuitive view of DSP firmware design.

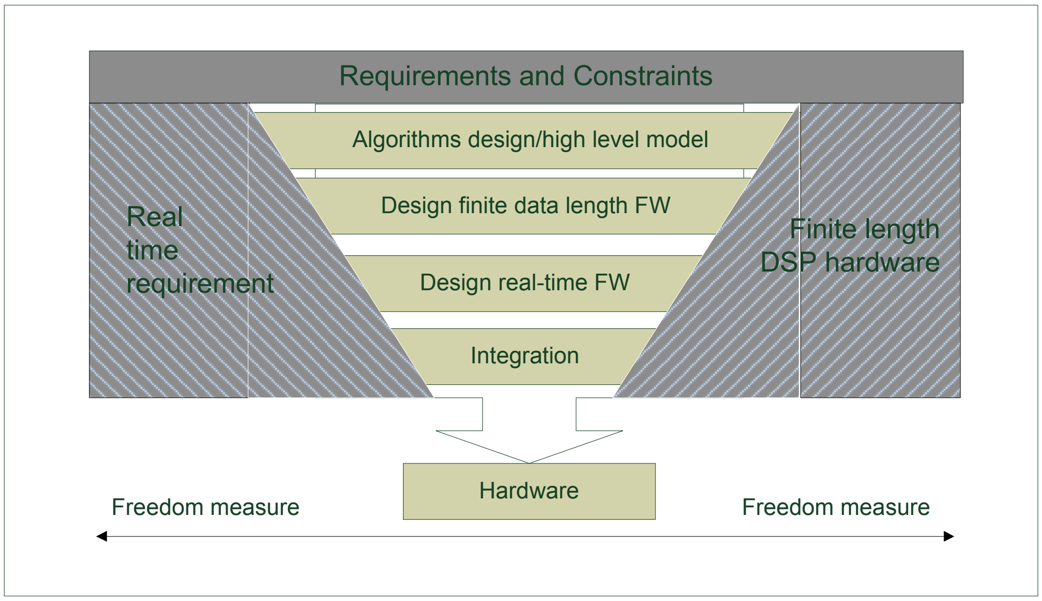

Requirements and Constraints
Algorithms design/high level model
Design finite data length FW
Design real-time FW
Integration
Real time requirement
Finite length DSP hardware
Hardware
Freedom measure
Freedom measure

Figure 2-1 Requirements and constraints on firmware design

2.1 Real-time Firmware

A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness [1]. So comparing with offline signal processing, real-time application need more attention on time constraints, that means if we want to design a successful real-time firmware, we must make sure the speed of signal processing is higher than the arriving speed of the signal, so the system can finish processing all the tasks before the deadline.

Figure 2-2 (a) gives a simplified example of the parameters of real-time tasks. After the input data packet arriving at time (1), the signal processing will be started since time (2). The computing time is (3) and the processing will be finished at time (4). Time (5) is the deadline at which all the tasks execution should be finished. The time interval between (4) and (5) is reserved for possible data dependant uncertainty and asynchronous events such as interrupts. Figure 2-2 (b) gives a more practical scheduling example. It minimized the computing latency which is the time interval between (1) and (5) by scheduling the input data packet reception and task execution in parallel.

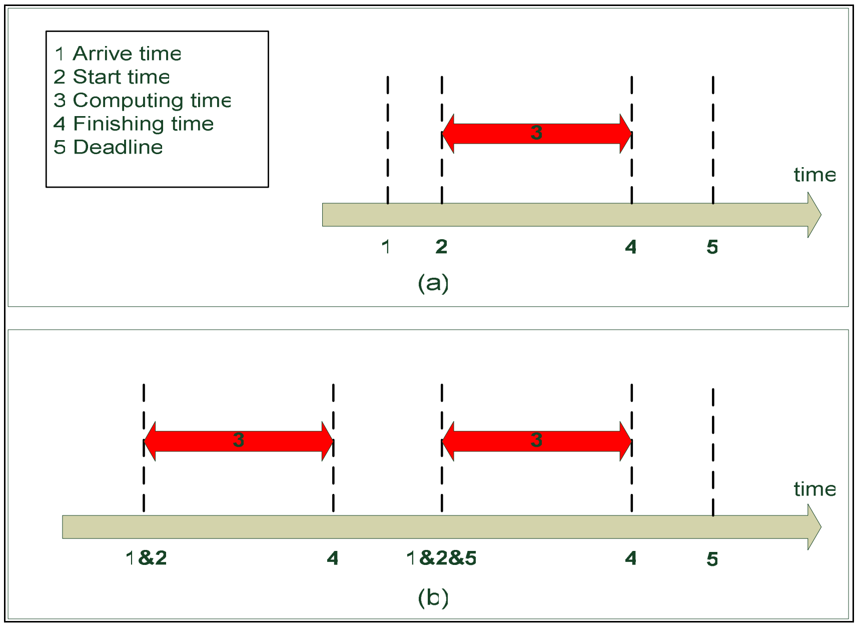


Figure 2-2 An example of the parameters of real-time tasks

2.2 Firmware Design Flow for multiple applications

If there are two or more tasks running in the DSP core, in fact, only one application can be executed at certain time, the hardware resource allocation should be decided by running a program above the application programs, which is called "super mode" in DSP application[2]. A scheduler should be implemented to decide which task can be executed at any particular time. The firmware can suspend and later continue executing a task before its next coming period.

When running multiple applications, the task with shorter coming period holds higher priority than the task with longer coming period in order to finish executing before the deadline. The higher priority task can interrupt the task with lower priority when it is running. The application with longest coming period or without steaming timing feature will holds the lowest priority which is call "background task". The computing capacity of the processor should afford all the computing load of multiple applications plus the super mode computing load.

The most basic example of this is a real time system that incorporates a keypad and LCD. A user must get visual feedback of each key press within a reasonable period. If the longest acceptable period was 100ms - any response between 0 and 100ms would be acceptable. Now assume the real time system is also performing a control function that relies on a digitally filtered input. The input must be sampled, filtered and the control cycle executed every 2ms. For correct operation of the filter the temporal regularity of the sample must be accurate to 0.5ms. The RTOS has itself created a task - the **background** task - which will execute only when there are no other tasks able to do.

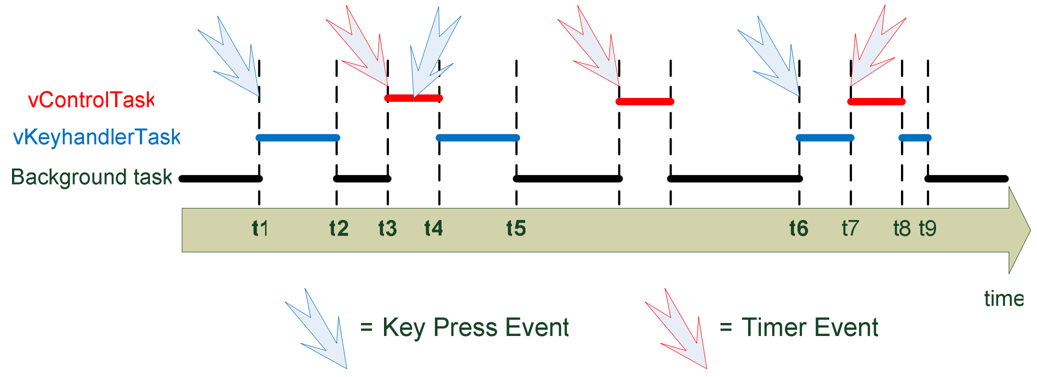


Figure 2-3 An example of multi-task scheduling

- At the beginning, no task is ready to run except background task, vControltask is waiting for the the correct time to start a new control cycle and vKeyHandlerTask is waiting for a key to be pressed. The processor is given to the background task.
- At time t1, a key press occurs. vKeyHandlerTask is ready to run. Because it has a higher priority than the background task, so is given processor time.
- At time t2 vKeyHandlerTask has completed processing the key and updating the LCD. It cannot continue until another key has been pressed so suspends itself and the background task is again resumed.
- At time t3 a timer event indicates that it is time to perform the next control cycle. vControlTask can now execute and as the highest priority task is scheduled processor time immediately.
- Between time t3 and t4, while vControlTask is still executing, a key press occurs. vKeyHandlerTask is now able to execute, but as it has a lower priority than vControlTask it is not scheduled any processor time.
- At t4 vControlTask completes processing the control cycle and cannot restart until the next timer event - it suspends itself. vKeyHandlerTask is now the task with the highest priority that is able to run so is scheduled processor time in order to process the previous key press.
- At t5 the key press has been processed, and vKeyHandlerTask suspends itself to wait for the next key event. Again neither of our tasks is able to execute and the background task is scheduled processor time.
- Between t5 and t6 a timer event is processed, but no further key press occurs.
- The next key press occurs at time t6, but before vKeyHandlerTask has completed processing the key a timer event occurs. Now both tasks are able to execute. As vControlTask has the higher priority, vKeyHandlerTask is suspended before it has completed processing the key, and vControlTask is scheduled processor time.
- At t8 vControlTask completes processing the control cycle and suspends itself to wait for the next. vKeyHandlerTask is again the highest priority task that is able to run so is scheduled processor time so the key press processing can be completed. [6]

Chapter 3

Challenges of Real-time signal processing

When design a real-time operating system, there are a lot of things need to be aware of for a software engineer. Such as the real-time kernel which must provide three specific functions with respect to a task: scheduling, dispatching, and intercommunication and synchronization [1]. A scheduler determines which task will be executed next in a multi-tasking system, while a dispatcher performs the resources allocation and other necessary jobs for starting a new task. Intercommunication and synchronization let the tasks can cooperate with each other. This chapter is presented for introducing some problems exist in these issues and giving the reader basic knowledge for understanding the design principles of this project.

3.1 Features of streaming signal and its processing

In real-time systems, all tasks are defined by their timing specifications, such as their deadline, type (periodic or sporadic), and required resources. Sporadic tasks are tasks associated with event-driven processing, such as responses to user inputs. Periodic tasks are tasks coming at regular intervals [5], also called “streaming signal”.

When running multiple streaming signals, the signal has shorter streaming period holds the higher priority than the priorities of the signals with longer streaming period. Since shorter streaming period also means the deadline of each coming signal is shorter than the other signals, we must finish executing the signal processing before their deadlines. The signal with higher priority can interrupt signals with lower priority. The signal processing task without streaming timing feature will hold the lowest priority and can be interrupted by any other signals.

3.2 Processes versus Threads

A process is an abstraction of a running program and is the logical unit of work scheduled by the operating system [1]. A process is not only containing the program code, which is known as ***text section***. It also includes the current activity of the running task, as represented by the value of the ***program counter*** and the contents of the processor's registers. In addition, a process should also include the process ***stack***, which contains temporary data (such as method parameters, return addresses, and local variables), and a ***data section***, which contains global variables [3].

Each process is represented in the operating system by a ***process control block*** (PCB) --- also called a task control block [3]. A PCB example is shown in Figure 3-1. All the necessary information of a specific running task can be found in the PCB, which including:

Process state: The process state is the current activity of a running task, which can be one of these states: new, ready, running, waiting, terminated, and so on.

Program counter: A register in a computer processor that indicates the address of the next instruction to be executed in program sequence.

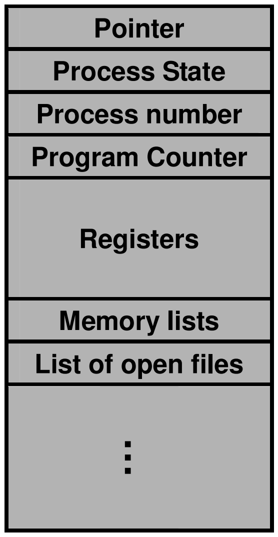


Figure 3-1 Process Control Block (PCB)

CPU registers: They are a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere.The number and type of registers vary from the computer architecture. They normally contain accumulators, index registers, stack pointer, and general-purpose registers.

CPU scheduling information: This includes the priority of the process, pointers to scheduling queues, and any other scheduling information.

Memory management information: Depending on memory system used by the operating system, it may contain such things: the value of the base and limit register, the page tables or segment tables.

Accounting information: This information is used to record the number of CPU, the time limit within a real-time system, the number of job or process, and so on.

I/O status information: This includes the list of I/O devices associated with this process, the I/O devices status, a list of open files, and so on.

The PCB saves all the significant information of a process, the information will be saved and restores when a context switch needed.

A **thread** is a **lightweight process** that shares resources with other processes or threads. A thread comprises a **thread ID**, a **program counter**, a **register set**, and a **stack.** It can share with other threads belonging to the same process its **code section**, **data section**, **address space** and other operating system resources, such as open files and signals. Although threads share many objects with other threads of that process, threads have their own *private* local **stack**. Normally, **Context Switching** between threads in the same process is faster than between processes.

One big difference between threads and processes is that global variables are shared among all threads. Because threads execute concurrently with other threads, they must worry about synchronization and mutual exclusion when accessing shared memory.

An application can be implemented as a single process with several threads of control, for example, a word processor may have one thread displaying words to the screen, another thread reading the keys pressed by user, and a third thread performing spelling and grammar checking in the background. Figure 3-2 illustrated the example above. The code, data and open files are shared during three threads for the same word processing application, however, each thread has its own Thread ID, PC, registers, and stack.

Nowadays, multithreading is extensively implemented in either software packets running on PCs or firmware controlling the embedded system. We can conclude four major benefits with multithreading:

1. **Responsiveness:** In a traditional process control system, when a process is blocked on a resource (such as a file, a semaphore, or a device), it can not continue executing until another process which running a different program release that resource. But when we use multithreading, a program can be allowed to continue running even if some part of it is blocked, thereby increasing the responsiveness to the user. For instance, a multithreaded web browser can allow the user interaction in a thread while an image is being loaded in another thread.
2. **Resource sharing:** As mentioned above, multiple threads can share memory and the resources of the same process to which they belong. The benefit of sharing

code section and data section is that it allows an application to have several threads of different function all within the same address space.

3. **Decreased overhead:** Due to memory and resources sharing with multiple threads, it is generally much more time consuming to create and manage processes than threads. The context switching is also less overhead and faster with multithreading.
4. **Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency. In a single-processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time [3].

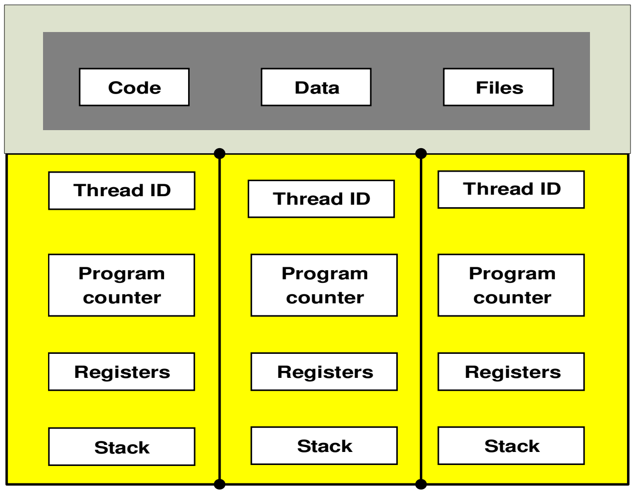


Figure 3-2 Multithreading in word processor application

3.3 Scheduling Algorithm selection

Various scheduling algorithms exist for real-time systems, but most commercial RTOSs use the priority-driven scheduling scheme, in which a static or dynamic priority is assigned to each task based on a given algorithm. Popular existing algorithms for the priority-driven scheduling scheme include the rate monotonic (RM) algorithm, and the earliest deadline first (EDF) scheduling algorithm.

3.3.1 Rate Monotonic (RM) algorithm

The rate monotonic algorithm is mainly used in the fixed-priority scheduling. In a fixed-priority scheduling system, the priority of each periodic task is fixed relative to other tasks due to their different task periods. The theorem, known as the rate monotonic theorem, is the most important (and useful) result of real-time systems theory. It can be stated as follows.

Theorem (Rate-monotonic) *Given a set of periodic tasks and preemptive priority scheduling, then assigning priorities such that the tasks with shorter periods have higher priorities (rate-monotonic), yields an optimal scheduling algorithm.[1]*

To illustrate rate-monotonic scheduling, consider the task set shown in Table 3.1.

Event Name	Handler Run Time *e*	Event Period *p*	Utilization *U =e/p*
A	1 μs	4 μs	0.25
B	2 μs	5 μs	0.4
C	5 μs	20 μs	0.25

Table 3-1 Sample task set for RM scheduling

Figure 3-3 illustrates the RM scheduling for tasks set in Table 3-1. All tasks are released at time 0. Since task A has the smallest period, it is assigned with the highest priority and scheduled first. Note that at time 4, the second instance of task A is coming and it preempts the current running task C, which owns the lowest priority.

The processor utilization *U* here is equal to the event execution time *e* divided by task period *p.*

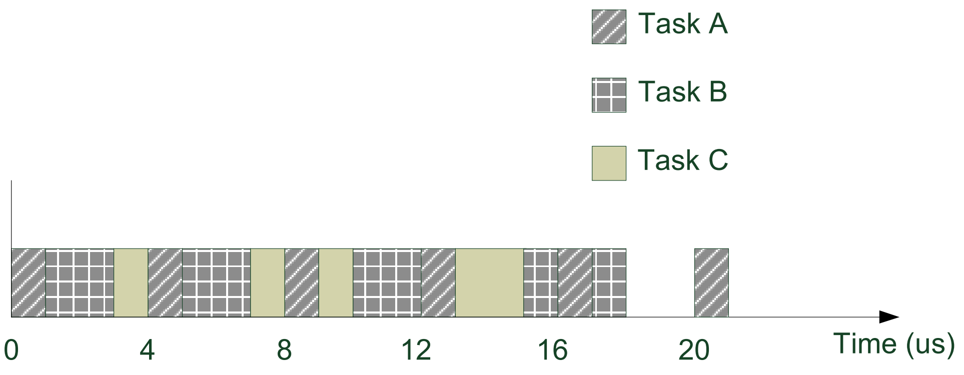


Figure 3-3 Rate-monotonic scheduling

3.3.2 Earliest Deadline First (EDF) Algorithm

In contrast to fixed priority algorithm, the priority of each task changes with respect to the other tasks released or completed in a dynamic priority scheme. A well-known dynamic algorithm, earliest deadline first, is one of them as dealing with deadline rather than execution time. The ready task with the earliest deadline has the highest priority at any point of time.

Figure 3- 4 illustrates the EDF scheduling policy with the task set shown in Table 3-2.

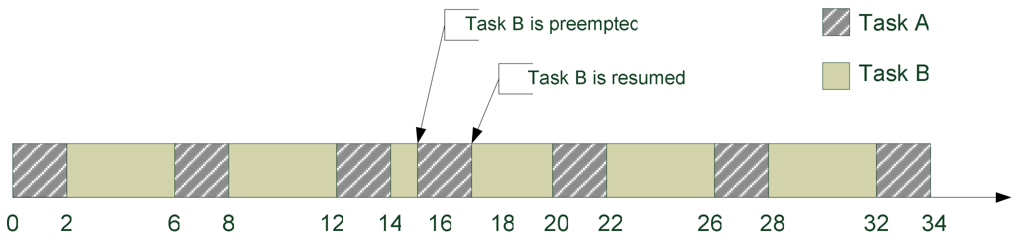


Figure 3-4 Earliest Deadline First Scheduling

Event Name	Handler Run Time *e*	Event Period *p*
A	$2\mu s$	$5\mu s$
B	$4\mu s$	$7\mu s$

Table 3-2 Sample task set for EDF scheduling

Although task A and task B release simultaneously, task A executes first according to its deadline is earliest. At time 2, task B can execute. Even though task A coming again at time 5, its deadline is not earlier than task B, so task B continue executing. This is going on until time 15 when task B is preempted by task A, since its deadline is later (t =21) than task A (t=20). Then task B resumes when task A completes.

3.3.3 Comparison of RM and EDF algorithms

Compared these two algorithms, the timing behavior of a system with a fixed priority scheduling is more predictable than that of a system with a dynamic priority algorithm. In case of overloads, RM is stable in the way of missed deadlines. In contrast, tasks scheduled with EDF are more difficult to predict which one will miss their deadlines during overloads. In addition, a late task that has already missed its deadline has a higher priority than a task whose deadline is still in the feature. That will lead to other tasks to be late if the late task is allowed to continue executing. So a good overrun management scheme is needed for such a system where overload conditions can not be avoided.

This project is aim to schedule some applications with fixed timing period and external interrupts occurring during the processing. So the RM algorithm is more fitted than EDF. We will provide an interface for user to configure the priority of each task and interrupt.

3.4 Interrupt handler and Context Switch

A processor has only one PC FSM and it is designed to do one thing at a time. However, users want the DSP processor can execute multiple tasks, including I/O handling. That makes us introduce a mechanism called “interrupt” to achieve this goal.

An interrupt can be hardware interrupt or software interrupt, both of them may cause the DSP core suspend current running task and execute interrupt service routine (ISR). A hardware interrupt is an asynchronous signal generated by a hardware device outside the core. On the other hand, a software interrupt is a synchronous event caused by an instruction asking supports from RTOS, e.g. access not reachable data. In most cases, an interrupt implicitly means hardware interrupt.

A complete interrupt handling process is distributed to the interrupt controller, the core hardware, and the ISR. Figure 3-5 illustrates the interrupt handling process step by step.

First, an interrupt is initialized by an interrupt request from a hardware pin (step1). Then the interrupt priority will be checked and compared with the priority of current running task in the core. If the interrupt owns higher priority than current running task, the interrupt will be accepted, otherwise, the interrupt request will be waiting until current running task finishes. Either an interrupt is accepted or rejected, an acknowledgement will be sent back to the hardware that required the interrupt. (step2).

As long as the interrupt is accepted, contexts (typically the next PC and flag values) of current running task will be saving into the stack in order to resume the task later (step3). Other state information does not need to be stored if the interrupt is very simple and register usage is restricted and predictable. However, if the interrupt is a normal one, all the contexts must be saved before serving the interrupt (step4). The DSP core then will start to serve the interrupt subroutine, which is the function requested by the interrupt (step5). After executing the ISR, the contexts saved in the stack will be restored (step6). Finally, saved PC and flag values of the interrupted program will be restored so that it can continue to execute, in addition, an End of Interrupt (EOI) signal is sent to the interrupt controller to let it check for next interrupt.

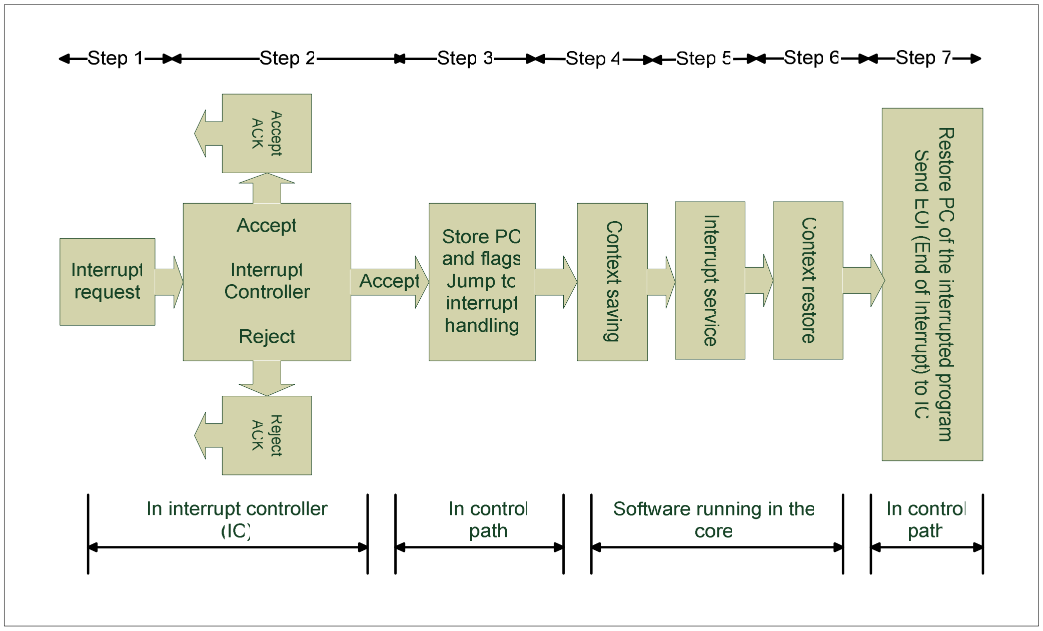


Figure 3-5 Interrupt handling steps [2]

Context Switching is the process of saving and restoring sufficient information of a task so that it can be resumed after being interrupted. The contexts of DSP applications include only register data, special and general register values. With context switching, multiple tasks can be executed in the same hardware resources. A context switching in a DSP processor is usually register context switching for a thread. Memory segments used by different threads are isolated instead of shared. A thread here stands for an interrupt service task that can be an application program, an I/O subroutine, or a system call. Context switching is inherent machine dependent, so it is normally implemented in assembly language related to the running machine. The detailed context switching implementation of this project is discussed in chapter 5.

3.5 Intertask communication and synchronization

From a practical point of view, tasks can not be totally independent or be preempted at any point of their execution, because task interaction is needed in most common applications. The main concern of this part is how to minimize blocking that may arise in a uniprocessor system when concurrent task use shared resources. Related to these issues is the problem of sharing certain resources that can only be used by one task at a time.

There are some strictly controlled mechanisms that allow tasks to communicate, share resources, and synchronize activity.

3.5.1 Buffering Data

The simplest and fastest way to pass data between tasks in a multitasking system is the use of global variables. Although it is considered as not good software engineering practices, are often used in high-speed operations.

One of the problems related to using global variables is that tasks of higher priority can preempt lower- priority routines at inopportune times, corrupting the global data. For example, one task may produce data at speed of 100 units per second, whereas another may consume these data at a rate less than 100 units per second. Assuming that the production interval is finite (and relative short), the slower consumption rate can be accommodated if the producer put the data into a storage buffer. The buffer holds the excess data until the consumer task can catch up. The buffer can be a queue or other data structure. On the other hand, if the consumer task consumes the data faster than it can be produced, or if the consumer cannot keep up with the producer, problems occur. Selection of appropriate size buffer is critical in reducing or eliminating these problems.

3.5.2 Time-Relative Buffering

A common use of global variables is in double buffering or Ping-Pong buffering. this technique is used when time-relative (correlated) data need to be transferred between cycles of different rates, or when a full set of data is needed by one process, but can only be supplied slowly by another process [1]. The classic bounded-buffer problem is a good example. In a bounded-buffer, a block of memory is used as temporal repository for data produced by "writer" and consumed by "reader". A further case in readers and writers problem is there are multiple readers and writers sharing a common resource, as shown in Figure 3-6. The buffer can only be written to read from by one reader or writer at a time.

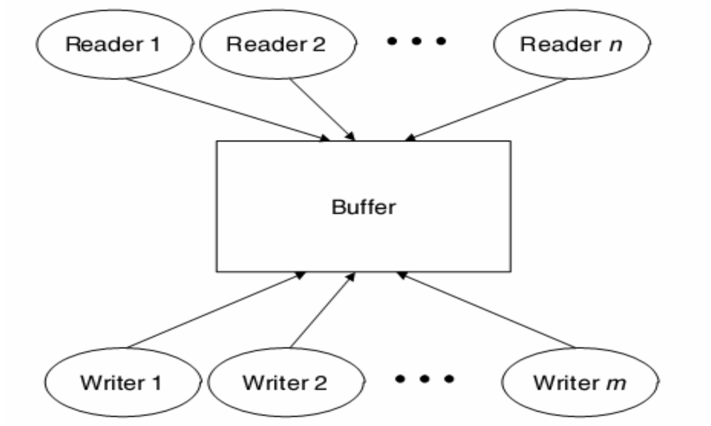


Figure 3-6 Readers and Writers problem, with *n* readers and *m* writers.

3.5.3 Mailboxes and Queues

Mailboxes or message exchanges are an extensive used intertask communication device in many commercial, full-featured operating systems. The tasks can write to the location via a post operation and to read from it via a pend operation. The interfaces of these two operations are:

```
void pend (int data, s);        void post (int data, s);
```

The difference between the pend operation and polling the mailbox is that the pending task is suspended while waiting for data to appear, therefore, no time wasted on checking the mailbox. The data passed by a message can be a flag used to protect a critical resource, a single piece of data, or a pointer to a data structure. In most implementations, when the data is taken from the mailbox, the mailbox will be empty. Then other tasks that wanted to pend on that mailbox can not receive the data.

Some operating systems support a type of mailbox that can queue multiple pending requests. In this case, the queue can be regarded as any array of mailboxes. Queues should not be used to pass arrays of data, instead of it, pointers should be used.

3.5.4 Critical regions and Semaphores

Multitasking systems are concerned with resources sharing, such as certain peripherals, shared memory, etc. The code that interacts with these resources can only be accessed by one task a time and not be interrupted. Such code is called a critical region. If there are two tasks enter the same critical region simultaneously, a catastrophic error can occur. For example, consider two C programs, Task_A and Task_B, which are running in a round-robin system. Task_B outputs the message "I am task_B" and Task_A outputs the message "I am Task_A." In the midst of printing, Task_B is interrupted by Task_A, which begins printing. The result is the incorrect output:

```
Iam I am Task_A Task_B
```

The most common method to protect a critical region is involving a special variable called a semaphore. A semaphore *S* is a memory location that acts as a lock to protect critical regions. There are two operations within a semaphore, **wait** and **signal** that are used either to set or to reset the semaphore. Traditionally, one denotes the **wait** operation as **P(S)** and the **signal** operations **V(S)**. The primitive operations are defined by the following C code:

```
void P(int S)
{
    while (S = = TRUE);
    S=TRUE; }
```

```
void V(int S)
{
      S=FALSE;
}
```

The wait operation will suspend any program calls until the semaphore **S** is **FALSE**, whereas the signal operation sets the semaphore **S** to **FALSE**.

Now, consider the Task_A and Task_B problem again. The problem can be solved by put the output statement into critical region with semaphore operations as follows:

```
void Task_A (void)
{
      P(S);
      Printf("I am Task_A");
      V(S):
}
```

```
void Task_B (void)
{
      P(S);
      Printf("I am Task_B");
      V(S):
}
```

Assume that **S** is within the scope of both **Task_A** and **Task_B** and that it is initialized to **FALSE** by the system.

3.5.5 Deadlock and livelock

Deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing when many processes share a specific type of mutually exclusive resource known as a *software,* or *soft,* lock [4].

An example is shown in the following. Task_A requires resources 1 and 2, as does Task_B. Task_A is holding resource 1 but is waiting on resource 2. Meanwhile, Task_B is holding resource 2 but is waiting on resource 1. Neither Task_A nor Task_B will relinquish the resource until its other request is satisfied. The situation is illustrated as follows where two semaphores, S and R, are used to protect resource 1 and resource 2, respectively, by the side-by-side pseudo code:

```
Task_A
…
P(S)
use resource 1
…
P(R)
stuck here
use resource 2
V(R)
V(S)
…
```

```
Task_B
…
P(R)
use resource 2
…
P(S)
stuck here
use resource 1
V(S)
V(R)
…
```

If semaphore S guard device1 and semaphore R guard device2, then the realization of the two might appear as the resource diagram in Figure 3-7.

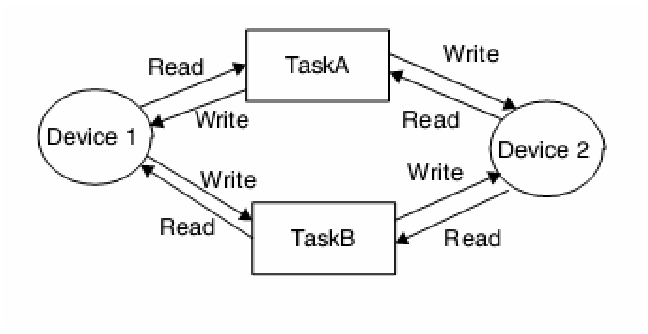


Figure 3-7 Deadlock realization in a resource diagram

A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none processing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can repeatedly trigger. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action [4].

Chapter 4

Commercial RTOS survey

Three mainstream RTOSs have been investigated in the following. The purpose of this survey is to discover the good features of each RTOS, which including the scheduling, inter-process communication, real-time performance and scalability. We used the technical documents offered by these three RTOS companies to compare the parameters qualitatively. These features are also important for this thesis project, so for other features of these RTOS will not be included in the scope of this survey.

4.1 OSE

4.1.1 Direct message passing

OSE is a modular, high-performance, full-featured real-time operating system from ENEA, optimized for complex distributed systems that require the utmost in availability and reliability. Differs from traditional interprocess communication way, OSE provides another approach called direct message passing or signaling. The mechanism is built into the operating system and means messages are sent directly from one process to another. A message, also referred to as a signal, contains an ID and both addresses of sender and receiver, as well as data. Once a message is sent to another process, the sending process can not access it any more. The ownership of a message is never shared. This important feature eliminates the common error such as memory access conflicts. The receiving process may specify which message types it wants to receive at any particular moment. The process can also specify a time limit for a signal before timing out. Figure 4-1 gives an example of direct message passing.

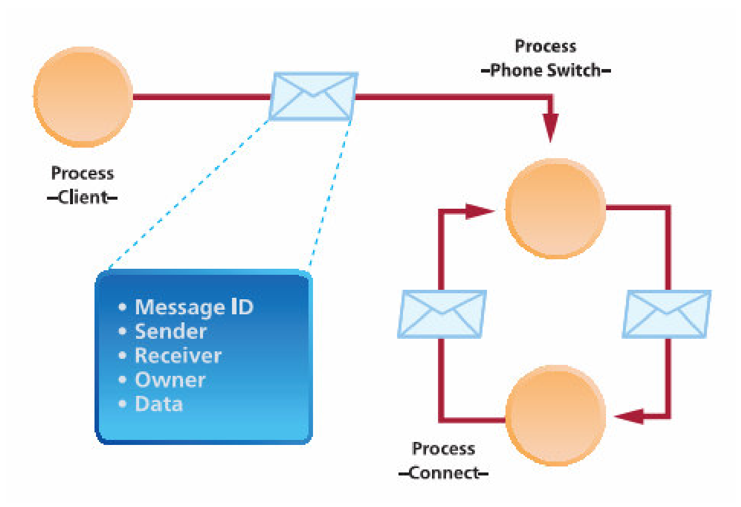


Figure 4-1 An example of direct signal sending [7]

4.1.2 Processes

Interrupt processes: Interrupt processes are scheduled in response to a hardware interrupt or using a unique OSE capability in response to a software interrupt. They run throughout the whole system processing. Interrupt processes have context just as other processes and always have the highest priority among the process types. Timer interrupt processes act in the same way as interrupt processes, except that they are scheduled periodically according to a specified time period.

Prioritized processes: A prioritized process is the most common process type, usually does the major part of processing and is designed as an infinite loop that run as long as no interrupt process or higher priority process becomes ready to run.

Background processes: Background processes run in a strict way with round robin time slicing mode at a prioritized level below all prioritized processes. A background process can be preempted by prioritized processes or interrupt processes at any time.

Blocks: The OSE real-time kernel has, in addition to processes, higher-level objects for organizing programs and their resources. The additional kernel objects are blocks. OSE allows a number of processes to be grouped together into a block. A block can have its own local and stack memory pool and blocks can be treated like a process.

4.1.3 Product Features

Dynamic runtime configuration

Environment variables, dynamic memory management, and automatic supervision make the system readily reconfigurable, even during runtime. Environment variables are named strings connected to either a process or to a block. The variables can be created and modified at runtime and are used by the application to store status configuration information. By using environment variables, processes and blocks can be reconfigured at runtime and information can easily be distributed to all processes in a system.

Real-Time Performance

The OSE real-time kernel provides **excellent deterministic performances**. All time-critical parts of the kernel are highly optimized. In particular, message passing in the OSE real-time kernel is extremely efficient with message pointers "handed off" rather than message data contents copied between processes, resulting in high data throughput rates. All execution times in the OSE Real-Time kernel are deterministic and are not affected by the size of the application, memory consumption, or number of processes.

The OSE Real-Time kernel is a **fully pre-emptive Real-Time kernel**. An interrupt can be served at any time, even during execution of a system call. The interrupt processing in OSE is optimized for low latency assuring quick interrupt response times and general real-time behavior.

CPU Transparency

OSE message passing is fully transparent between different CPUs. A message is sent in the same manner regardless of whether the receiving process is running in the same or a remote CPU. OSE does not require a master CPU in a distributed system. Each CPU is the same and has the same capabilities. There is no difference in the way a globally visible or a local private process is created. A local process can become globally visible to the whole system without re-creation.

Performance and flexibility

Processes can either be static or dynamic in OSE. Static processes are created at system start by the kernel. Static processes exist all the time that a system exists. Dynamic processes can be created and killed at run-time enabling instances of the same code and preserving system resources. By offering both types of processes, OSE enables applications to be optimized for the best performance and flexibility.

4.2 VxWorks

4.2.1 A Real-Time Process Model

VxWorks 6 is commercial RTOS for device software applications from Wind River. Modern CPUs implement a strong partition between a protected user mode, in which application executes, and an unprotected kernel mode, in which the OS kernel and associated drivers execute. Characteristic of user mode typically provides an easy-to-understand, procedural call programming model to isolate the applications from hardware platform and the physical provision of OS services. In contrast, kernel mode trades off abstraction and protection of kernel components for the requirement of having direct access to hardware, tightly bound interaction with the kernel. The real-time process (RTP) model introduces the user and kernel mode partition for VxWorks. A user mode application is an ELF executable and executes as an RTP. RTPs are isolated from the kernel that contains the VxWorks OS, vice versa. This isolation allows application to execute independently and provides code, data, and symbol namespace separation. Memory protection is provided by a CPU’s MMU, such that a fault occurring within an application will not affect the kernel or other applications.

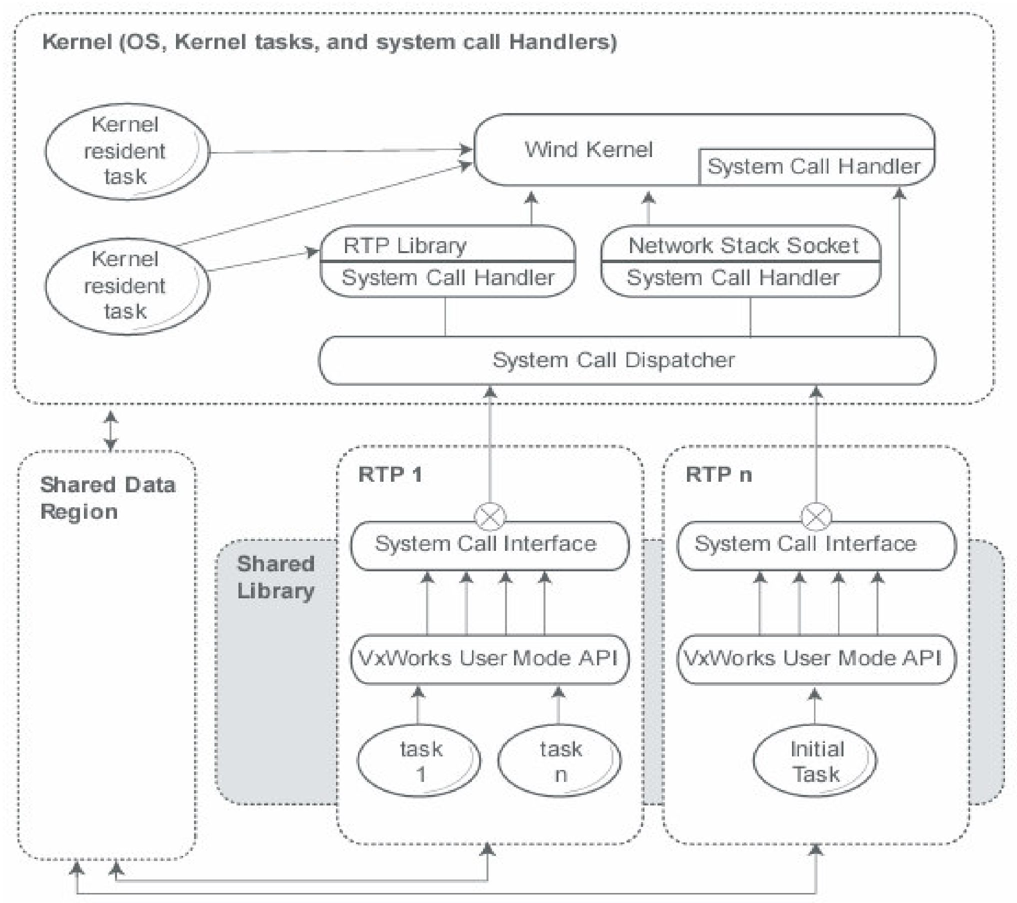


Figure 4-2 RTP Architecture block diagram [8]

The execution unit of an RTP is a VxWorks task, and there may be multiple tasks executing within an RTP. Tasks in the same RTP share its address space and memory context, and can not exist beyond the life time of the RTP. All the resources allocated to the RTP will be reclaimed when the last task exits.

4.2.2 Process Model: Real-Time Characteristic

Process creation completes in two phases: The creation of the RTP is separate from the loading of the application. The creation phase of an RTP is a minimal activity to verify that the application is a good ELF image, create the objects associated with the RTP, create the initial memory context, and create the initial task running within the RTP. This first stage runs in the caller's context. Then the second phase (loading and instantiating the RTP) will execute in the newly created task, within the new process's memory context, and run with the priority specified by the user.

Processes load entire applications: When the processes are created, the application will be fully loaded. There is a latency to load and start an RTP that may be relatively long. If this is a problem, the RTP can be loaded ahead of creation, and its initial task left suspended. Without demand paging, pages are never loaded from disk, so there is **never a non-deterministic delay during execution**.

RTPs are not scheduled: Only tasks are schedulable in a VxWorks system, allowing the system designer to implement interaction between tasks with their priority throughout the system. In addition, **the highest priority task in the system will always be guaranteed to run**. A task can be run is not simply because it is the RTP's turn, but the task is designed to be executed at that time. Therefore, the system remains deterministic.

4.2.3 Interaction with the kernel and other RTPs

Applications running in the RTP are not directly linked against the kernel. A system call interface is used to access kernel, which perform work in kernel mode on behalf of the application in the application's context.

A User Mode API of VxWorks is available when developing an RTP. This API primarily differs from the VxWorks Kernel Mode API in that it doesn't provide direct access to hardware, processor, or BSP-level features. Meanwhile, it provides **excellent compatibility with the traditional VxWorks API, and provides a greater degree of alignment with the POSIX specification**.

Although tasks in different RTPs are isolated, they may interact using dedicated mechanisms (such as shared data regions), and various inter-process communication mechanisms (such as semaphores, message queues, message channels, signals, etc.). Such interaction is achieved via system calls.

Named shared data regions can be created and shared between the kernel and multiple RTPs. A shared data region appears at a common location in each RTP's virtual memory map and allows easy data sharing as pointer addresses are common between RTPs.

4.2.4 Product Features

Global task scheduling
The global scheduler schedules tasks across all RTPs. This allows the designer to easily manage the interaction between task priorities on a system-wide basis, which is often highly important in a real-time device. It is not necessary to place high-priority tasks within the kernel to ensure they get adequate processor priority.

Scalability
RTP support is a scalable component in VxWorks. Only systems that require RTP support must include the RTP component. Similarly, ancillary RTP services, such as shared libraries and shared data, need only be included if they are required by the RTPs being executed. Even system calls that are not required may be scaled out of the system, and additional system calls can be dynamically added to a system.

4.3 DSP/BIOS

4.3.1 Modular OS

DSP/BIOS kernel is a **scalable real-time multi-tasking kernel**, designed specifically for the TMS320C6000™, TMS320C5000™, and TMS320C28x™ DSP platforms from Texas Instruments. It is a modular based operating system. The user has the ability to choose the parts of the operating system that are needed by application. Every module has a unique name and special three-letter prefix that is used as an identifier for system calls and header files. All identifier for the system calls are composed of three uppercase letters followed by an underscore e.g. HWI_* and IDL_*.

4.3.2 Processes

DSP/BIOS has four major types of process: Hardware Interrupt process (HWI), Software Interrupt process (SWI), Task process (TSK) and Background thread (IDL), associated with two extra types: Clock functions (CLK) and Periodic functions (PRD). Clock function is a special kind of hardware interrupt and Periodic function is belonged to software interrupt. The priority level of four major process types from high to low is HWI, SWI, TSK and IDL [9].

Hardware Interrupts and Clock functions: The hardware interrupts that DSP supports are triggered either by on chip devices or external devices. A user has the ability to enable or disable the hardware interrupt. However，the HWIs are divided into two groups, maskable and nonmaskable hardware interrupts. Maskable hardware interrupts can be enabled and disabled either individually or globally by software. Nonmaskable hardware interrupts cannot be enabled nor disabled. The API for the HWI module contains the functions to enable or disable all maskable hardware interrupts and do context switching when executing its ISR.

DSP/BIOS provide two timing methods, the high- and low- resolution time and the system clock. By default, the low- resolution time is the same as the system clock. It is possible to make CLK objects execute user defined functions at every system clock tick. One CLK object is throughout the processing called PRD_clock. This object drives the system clock and triggers the periodic functions. The CLK objects are triggered as timer interrupt, therefore, the functions executed by CLK objects must be short.

Software Interrupt and Periodic functions: Instead of being triggered by hardware, software interrupt are triggered by calling SWI system call from an application. The software interrupt can also be preempted by hardware interrupts and software interrupts with higher priority. There are 15 different priority levels among the software interrupt, and lowest one is reserved for a SWI object named KNL_swi that executes the TSK scheduler. Each software interrupt comes with a mailbox.

Periodic functions are objects that execute periodically and are triggered by the PRD_clock object. When a PRD object is created a period will be assigned to a counter belonging to this object. The counter value is an integer that will be decremented every system tick. When the counter's value reaches zero the PRD object executes and the counter is set to its initial value. An SWI object called PRD_swi is triggered every system tick and manages the scheduling of the periodic functions.

Task Processes: Task processes are the only process type that can be blocked during executing, therefore, it is used when waiting for common resources. In contrast to Hardware and Software interrupts use the system stack, task processes are assigned their own stacks because they are able to be blocked. The stack size can be specified separately. There are 16 different priority levels can be assigned to task process. The lowest one is reserved for a task process name TSK_idle. This task process executes the functions defined by IDL objects.

Idle Processes: The idle loop or background process that executes when no other process is executing. It will call the user defined functions assigned to the IDL objects in a round robin scheme.

4.3.3 Developing DSP/BIOS Applications

DSP/BIOS is a scalable set of run-time services that provide the essential foundation upon which developers build their applications. DSP/BIOS provides these run-time services in the form of a scalable run-time library, also referred to as the DSP/BIOS kernel. Included in this library of services is:

- A small, preemptive scheduler for real-time program threads with optional multitasking support
- A hardware abstraction of on-chip timers and hardware interrupts

- Device-independent I/O modules for managing real-time data streams
- A series of functions that perform real-time capture of information generated by the DSP application program during its course of execution.

Figure 4-3 illustrates the components involved in building DSP/BIOS-based DSP applications.

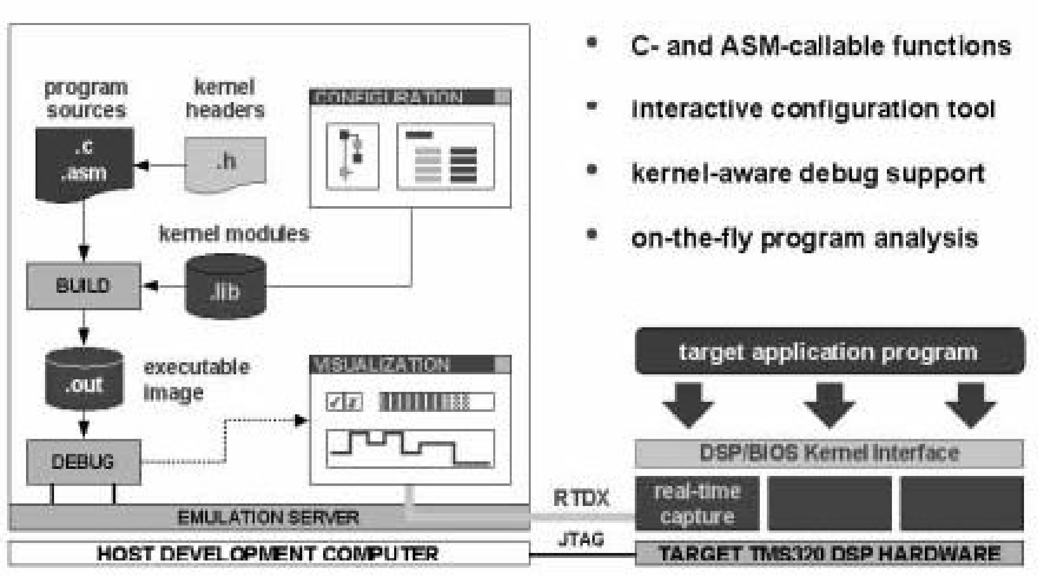

program sources
kernel headers
CONFIGURATION
.c
.asm
.h
kernel modules
BUILD
.lib
.out
executable image
VISUALIZATION
DEBUG
EMULATION SERVER
HOST DEVELOPMENT COMPUTER
RTDX
JTAG
C- and ASM-callable functions
interactive configuration tool
kernel-aware debug support
on-the-fly program analysis
target application program
DSP/BIOS Kernel Interface
real-time capture
TARGET TMS320 DSP HARDWARE

Figure 4-3 Building DSP/BIOS-Based DSP Applications

4.4 Survey Conclusion

After the investigation for three widely used RTOS, we approximately know how a good RTOS should be. These three RTOS all have some good features like deterministic execution time, fully pre-emptive. However, they also have some special features differentiating from each other. Following table gave a collection of these features.

	OSE	**VxWorks 6**	**DSP/BIOS**
Scheduling			
Global task scheduling	×	×	×
Static scheduling	×	×	×
Dynamic scheduling	×		
Inter-process communication			
Semaphores, message queues, message channels, signals		×	×
Direct message passing	×		
shared data regions		×	
Real-time performance			
Deterministic execution time	×	×	×
Time-critical parts optimized	×	×	×
Fully pre-emptive	×	×	×
CPU transparency	×		
Scalability		×	×

Table 4-1 Features comparison of three widely used RTOS

Chapter 5

Implementing super mode for multiple streaming DSP applications

As mentioned in chapter 2, running two or more streaming signal processing applications in one processor, the usage of the hardware resource shall be decided by running a program above the application programs. This program is called "super mode" in DSP application [2]. It is actually part of a simplified RTOS, whereas differing from RTOS. Rather than RTOS dealing with all the complex problems from software to hardware, which including interface to human and interface to machine, super mode is more concentrated on dealing with pre-defined streaming tasks, where only thread and resource management need to care. For this project, we want to design and implement a real-time multi-task threading control subsystem. Apparently RTOS is too big for this purpose either in functionalities or in developing time. So design and implement a super mode should be a better choice.

The flow chart of a super mode for multiple DSP applications is shown in figure 5-1. The super mode includes a **background task identification** subroutine, a **priority setting** subroutine, an **interrupt handling** subroutine, a **context switching** subroutine, and an **interrupt enable/disable** subroutine.

The mainly functions covered by super mode are:

1. When a task is ready to run, it will send a request from an interrupt request pin to interrupt controller, which is a peripheral hardware module outside the DSP core. The interrupt controller accepts a request of an interrupt and asks the super mode to give a service. Then the super mode will decide either serving the interrupt request or continue executing current running task according to the priority they own. If the running task holds the higher priority, the interrupt request will be rejected and put into ready list (*schqueue*). If the task of interrupt service had higher priority, the running task will be interrupted, giving hardware resources to the interrupt service.

2. The priority of each task is configured by the super mode during the starting phase of an application according to its streaming period. The application has shorter streaming period holds higher priority than the priorities of the applications with longer streaming periods. The signal processing task without streaming timing feature has the lowest priority and will be configured as the background task.

3. As long as an interrupt request is accepted, super mode will check if context saving is required. Context saving will be executed before serving the custom ISR (Interrupt Service Routine).

4. After serving an interrupt, super mode should handle two things: do context restoring if the interrupted task has not finished and context has been saved, the other one is to enable interrupt if it was disabled. Super mode has a complement subroutine which can be called by any task to enable or disable interrupts at any time.

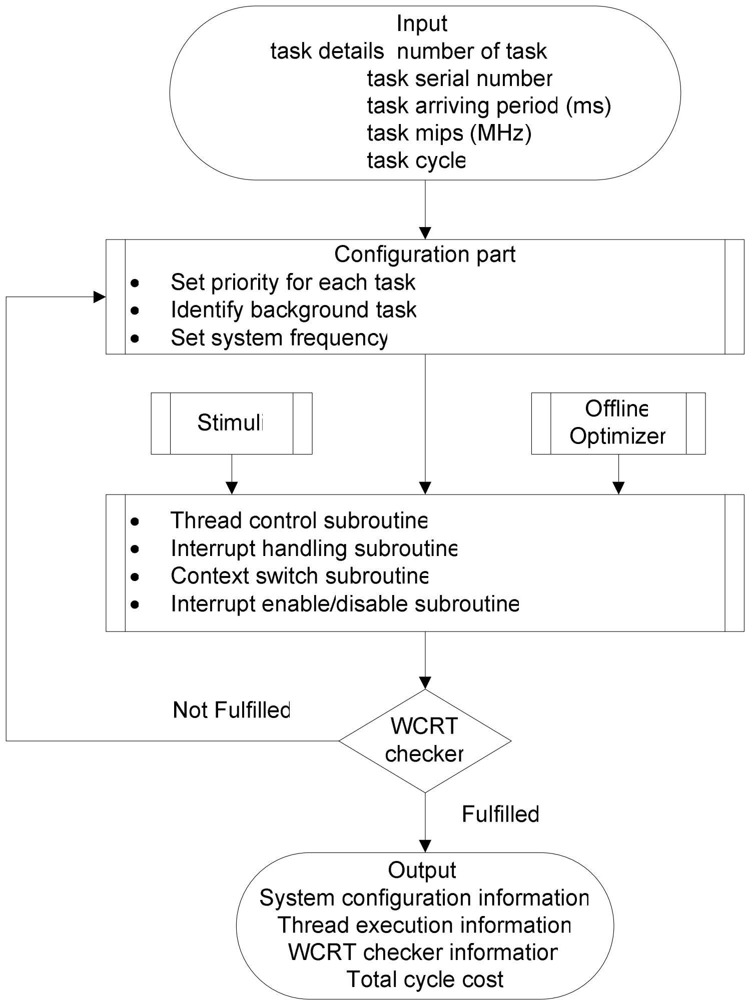

Input
task details number of task
task serial number
task arriving period (ms)
task mips (MHz)
task cycle
Configuration part
Set priority for each task
Identify background task
Set system frequency
Stimuli
Offline Optimizer
Thread control subroutine
Interrupt handling subroutine
Context switch subroutine
Interrupt enable/disable subroutine
Not Fulfilled
WCRT checker
Fulfilled
Output
System configuration information
Thread execution information
WCRT checker information
Total cycle cost

Figure 5-1 Macro view of behavior model for super mode

For better encapsulation and future system updated or reuse, the project uses the object-oriented programming language C++ for behavior modeling implementation. The data structure is abstracted in different classes defined by these own features. This chapter will introduce the details of different function parts within super mode, and how they implement the functions mentioned above. In addition, the **thread** component will be briefly introduced to better understand how super mode works.

5.1 Thread component

In this project, we defined a data structure for managing threads, which represent sequential execution of code within a program. As the same in the other operating system, a *thread* object contains the thread name, the program counter, the processor registers, and the execution stack. Each thread also has an associated *state* describing what the thread is currently doing. There are three states:

Input packet ready: When a task is ready to run, it will send a request from an interrupt request pin to interrupt controller, which is a peripheral hardware module outside the DSP core. The interrupt controller accepts a request of an interrupt and asks the super mode to give a service. Then the function *Thread::Create()* turns the input packet into the threads that can be run in the system.

Ready: The thread is eligible to use the processor, but another thread is currently running. When the scheduler (super mode) selects a thread for execution, it simply removes the thread from the ready list (*schqueue*) and changes its state from **Ready** to **Running**.

Running: The thread is currently running in the processor. A global variable *currentThread* always points to the currently running thread.

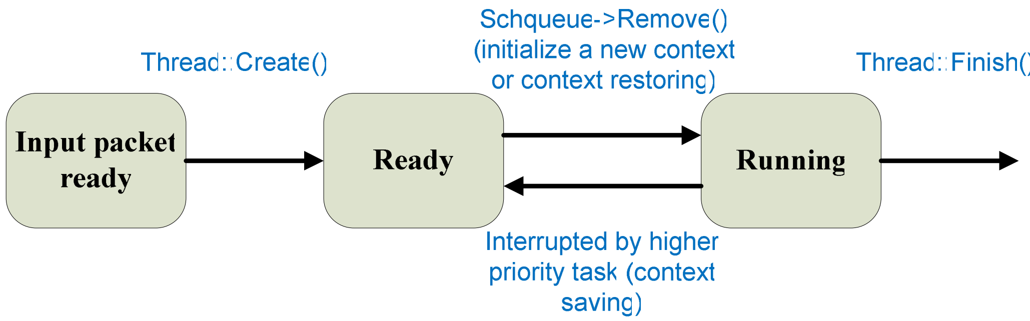


Figure 5-2 Thread state transition.

Normally, operating system maintains a data structure called TCB (Thread Control Block) which contains all the information associated with a thread. In contrast, we define a class to manage thread in this project, so all the information associated with thread is maintained as private data of a *Thread* object instance. To get specific thread information, a pointer to the thread instance is needed. The following operations are supported by *Thread* object:

Thread *Thread (char *debugName) The *Thread* constructor does only minimal initialization. The thread's status is set to **Input packet ready**, its stack is initialized to NULL, its given name is *debugName*, etc.

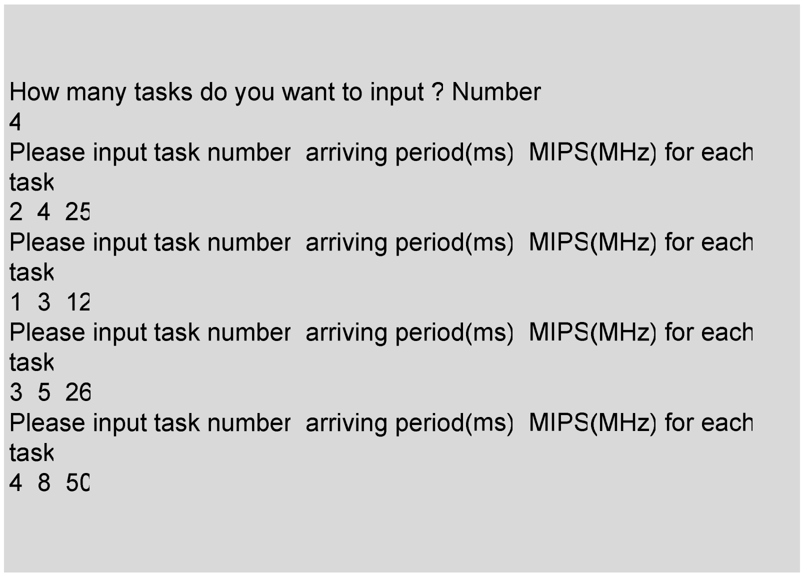
Create (VoidFunctionPtr func, int arg) The *Create* function does the concrete work for thread creation: turning a thread into one that processor can schedule and execute. *Create* allocates stack space for the new thread, initializes the registers, and so on. Argument *func* is the address of a procedure where execution is to begin. Argument *arg* is an argument that should be passed to the new thread – here is the task identity number in task Array which keeps all the tasks information configured by user.

void Sleep( ) Interrupt the current thread, execute other task by running context switching. This function is called when we need to interrupt the *currentThread* by a higher priority task or interrupt.

void Finish( ) Terminate the current running thread.

5.2 Task configuration

A **Task** class defined the configurable parameters of each application, and can be configured by user through a user interface, an example of user interface is shown in Figure 5-3.



```
How many tasks do you want to input ? Number
4
Please input task number  arriving period(ms)  MIPS(MHz) for each
task
2  4  25
Please input task number  arriving period(ms)  MIPS(MHz) for each
task
1  3  12
Please input task number  arriving period(ms)  MIPS(MHz) for each
task
3  5  26
Please input task number  arriving period(ms)  MIPS(MHz) for each
task
4  8  50
```

Figure 5-3 Interface of configuration part.

The data members included in **Task** class are:

Task[i]->num: task serial number

Task[i]->time: fixed task arriving period

Task[i]->mips: the frequency of each task (MHz)

Task[i]->total_c: a fixed cycle cost, which can be calculated by arriving period multiply task frequency

In addition, two more data are important in this configuration part. One is **nbr_of_task** represents the number of tasks that user want to schedule by TC. Another one is **system_mips** which will be calculated out after user input all the tasks details, represents the frequency of system.

After the user input all the parameters of a task, we will define a **Task** pointer as an object instance of each task and save it into a **Task** array. This *Task* array will further be used for task priority setting subroutine and background task identification subroutine.

The **task priority setting subroutine** is actually based on the streaming period of each task. In a real-time system, when running multiple applications, the application with shorter streaming period holds higher priority. So according to the time feature entered by user, we can use the bubble sort algorithm to set the priority to each task and save it in *Task* array with this priority order. Then the **background task** is the task with lowest priority in *Task* array, and will be executed as first task in processor.

The system frequency which is represented by **system_mips** is calculated out corroding to the overheads of the super mode, which including the computing load for enable/disable interrupt, for context saving and context restoring, etc. Table 5-1 gives the cycle cost of the super mode for one thread switching in detail.

Description	**Cycle cost**
Disable interrupt	5 cc
Context Saving	
32 General registers	32 cc
32 Special registers (first move data to general register, then save to stack)	64 cc
4 Accumulation registers (guard bits, high part 16 bits, low part 16bits)	4×6 cc
Context Restoring	
32 General registers	32 cc
32 Special registers (first move data to general register, then save to stack)	64 cc
4 Accumulation registers (guard bits, high part 16 bits, low part 16bits)	4×6 cc
Enable interrupt	5 cc
Total	250 **cc**

Table 5-1 super mode cycle cost for one thread switching

As same as the other real time system, the real time kernel of this project also has a variable to measure the time – the *cycle* cost which represented by **stats->userTicks.** A cycle increments one userTicks with strict temporal accuracy-- allowing the real time kernel to measure time to a resolution of the system frequency. The system divided the whole processing into many small executing iteration periods with same interval. The interval of each executing iteration period is decided based on the smallest time unit of all the tasks. For example, if all the tasks' arriving period time units are 1ms, then the time unit 1ms is the interval of each executing iteration period. Within this executing iteration period, the real time kernel will process each task as many times as system frequency. In this way, we can count the cycle cost as well the real time cost. We can also accurately detect the coming of each task through calling stimuli once an executing iteration period.

According to the cycle cost listed above, we can calculate the proportion of super mode computing load in an executing iteration period. If the proportion is relatively lower, we can ignore it. Whereas the computing load of super mode is relatively higher, we must increase the system frequency in order to finish executing task before their deadline.

$$\frac{\text{cycle cost of super mode}}{\text{An executing iteration period} \times \text{machine clock}} \tag{5.1}$$

The cycle cost of super mode for one thread switching is 250 cycles, and machine clock for *Senior* DSP is 100MHz, and smallest executing iteration period is decided by the time units of the task streaming periods. So the primary element in this equation is the time unit of the task streaming periods. Following we collected some samples of different executing iteration periods and the super mode proportion.

Task streaming period	**1ms**	**0.1ms**	**0.01ms**	**1 μs**
proportion	**0.25%**	**2.5%**	**25%**	**250%**

Table 5-2 time unit samples and corresponding super mode proportion

From the table above, we can see when time unit of the task streaming period is 1ms, the computing load of super mode can almost be ignored, however, with the decreasing of time unit, the proportion is increasing dramatically. When the time unit is down to 1 μs, the machine clock of Senior DSP can not fulfill the requirement. So we set the system frequency as 1.25 times higher than the total computing load according to the data we got above.

$$system\_mips = 1.25 \times \sum task[i] -> mips \tag{5.2}$$

5.3 Task management process

As mentioned above, the whole processing is divided into many small executing iteration periods. To make data packet reception and task execution in parallel, we need to mask all the interrupts in the end of each executing iteration period. So when the cycle is up near to the end of full times of executing iteration period, we will turn off the interrupt by invoking the function **interrupt->SetLevel (IntOff)**, and of course make it available by the next period start. In practice, we reserve the last 5 cycles for this purpose.

Each time a new period coming, super mode will automatically call the function **Stimuli()** in order to check if there is any new task arrives. If there is any, the new coming task is inserted in sequence to the ready queue (**schqueue**) by its priority. The **Stimuli()** will return a integer number represents how many new coming tasks arrives concurrently. And then super mode will schedule the *currentThread* into the ready queue (**schqueue**), too. Since the tasks are sorted in the order of its priority in the ready queue, so when the *currentThread* is put into the ready queue, all the tasks that are currently ready to run are sorted in sequence of its priority, the first task in this queue will own the highest priority and be removed from this queue – in this way, the *currentThread* will be always the task with highest priority. Super mode will check if the removed task is the same task super mode scheduled into the ready queue (*currentThread*). If so, there will be no change, the *currentThread* will continue to execute. If the removed task is a new one, super mode will initialize the new thread step by step as described in the following part *Thread Initialization* and then do Context Switching to make new thread ready to run.

As a task finishes, super mode will check if there are any tasks have not finished in the ready queue. The one with highest priority at this time will be removed from the ready queue and be done Context Switching so that it can resume its execution in the next cycle.

When all the tasks finishes their execution, including the *background task*, there may be an exception that some tasks has not yet come due to really late arriving period. If so super mode will initialize the new thread and then do Context Switching to make new thread ready to run.

Figure 5-4 shows the detailed super mode processing steps.

5.3.1 Thread Initialization

In the beginning, super mode will schedule the **background task** as the first thread running in the real time kernel. This will be done by initializing the thread name as "**Task**" plus "**task[i]->num**" of the background task and using thread constructor function **Thread *Thread (char *debugName)**. Then using **Create**

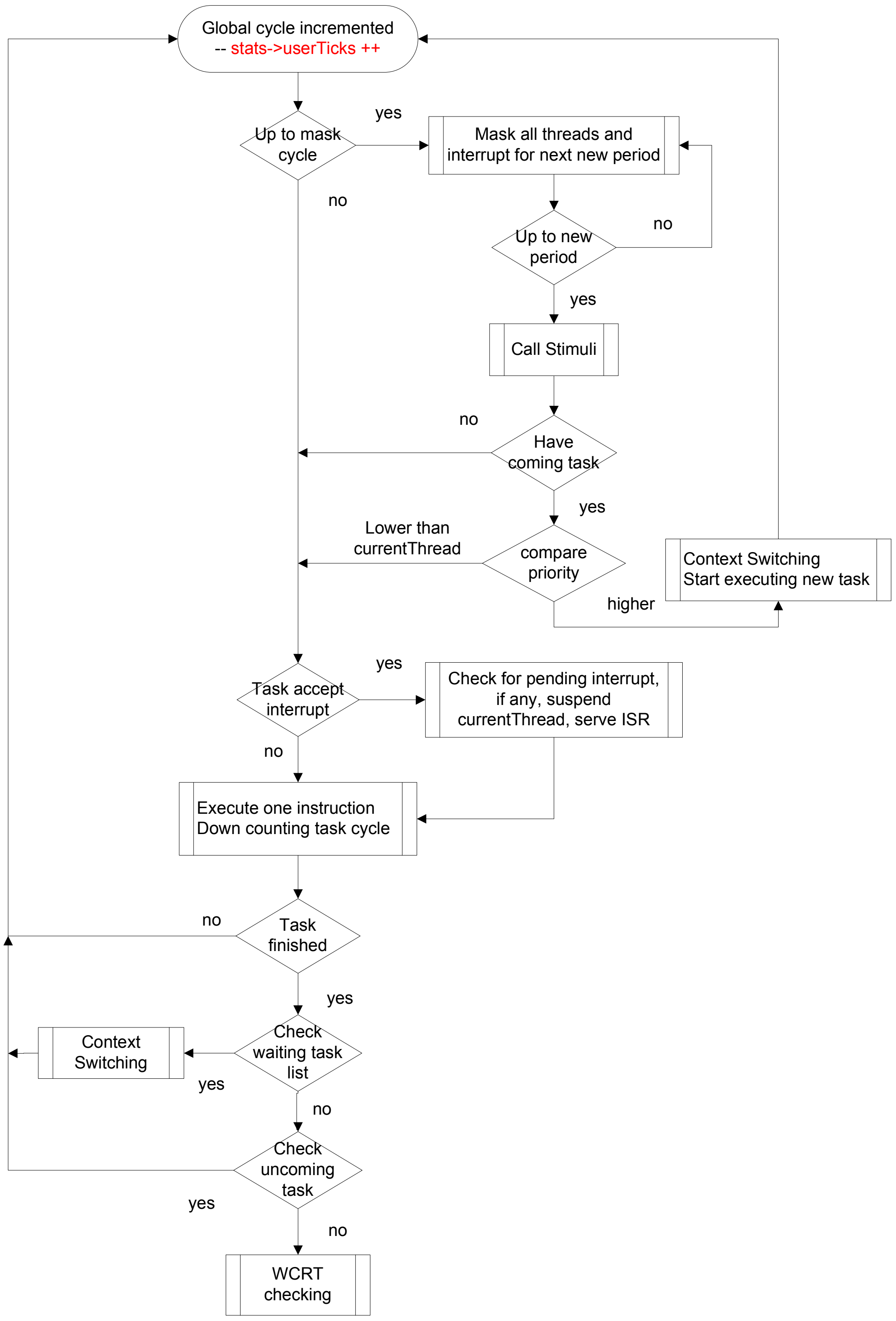


Figure 5-4 Super mode processing steps

(VoidFunctionPtr func, int arg) function to turn the thread to be ready to run in the processor, **func** here is common code section of program Offline Optimizer, parameter **arg** is the number '**i**' in task array that also represents the task priority.

5.3.2 Hardware interrupts handling

The hardware interrupts have even higher priorities than the ready tasks. When cycle increments, the real time kernel will check the hardware interrupt controller if there is any pending interrupt signal generated by a peripheral device, which is done by **interrupt->OneTick**(). If any, the real time kernel will turned off the interrupt by function **interrupt->SetLevel (IntStatus level)**, suspend *currentThread*, and then invoke the ISR. During serving the ISR, the *cycle* is still incremented with time passed. After serving an ISR, the interrupt will be turned on and *currentThread* will be restored and resumed to execute next instruction. **Interrupt->SetLevel (IntStatus level)** is a global function that can be called anywhere to turn the interrupt state to either **IntOff** or **IntOn**.

5.3.3 Context Switching

Before serving an ISR, the context saving will be checked if it is needed. Context switching is inherently machine dependent, since the registers to be saved, how to set up an initial call frame, etc, are all specific to processor architecture. This project is supposed to run and be simulated on *Senior* DSP processor, so all the instruction sets of context switching are implemented using assembly language with constraints of *Senior* processor.

The *Senior* processor is a single issue DSP processor with 32 general registers (16 bits wide), r0 to r31, used as computing buffers. Several special register such as processor status register and core control register together with 4 accumulator register must be saved during context switching. Detailed processing steps and information need to be saved or restored in context switching can be found in Figure 5-5.

The real time kernel will hold a software stack pointer table. Each new thread will be allocated a piece of memory area as software stack and a stack pointer pointed to that memory. When context saving is needed, the processor will first save the value of program counter into hardware stack of thread in order to put it back and continue executing the next instruction when the thread is ready to resume, then check the stack pointer table to find out the related stack pointer and move the address to address register **ar0**, set the step size of AR0. The contents in processor status register (**fl0**) and core control vector register (**fl1**) are important as they keep the recent result flags from ALU and MAC, and other core state information like Hardware stack pointer and global interrupt enable, so both these two registers must be saved after saving PC. There are 5 general registers need to be used as buffers during context switching, the rest 27 general registers will be saved sequentially. Next information need to be saved is Accumulator registers with 32 bits wide, so they will be saved in sequence of higher part 16bits and lower part 16bits. When saving hardware stack, we first need to know the value of hardware stack, which we can get from core control register (**fl1**), then blank hardware stack one by one. The hardware stack value is then saved for later restoring the contents in hardware stack back. The Context Saving is to be done with saving the new software stack pointer into stack pointer table.

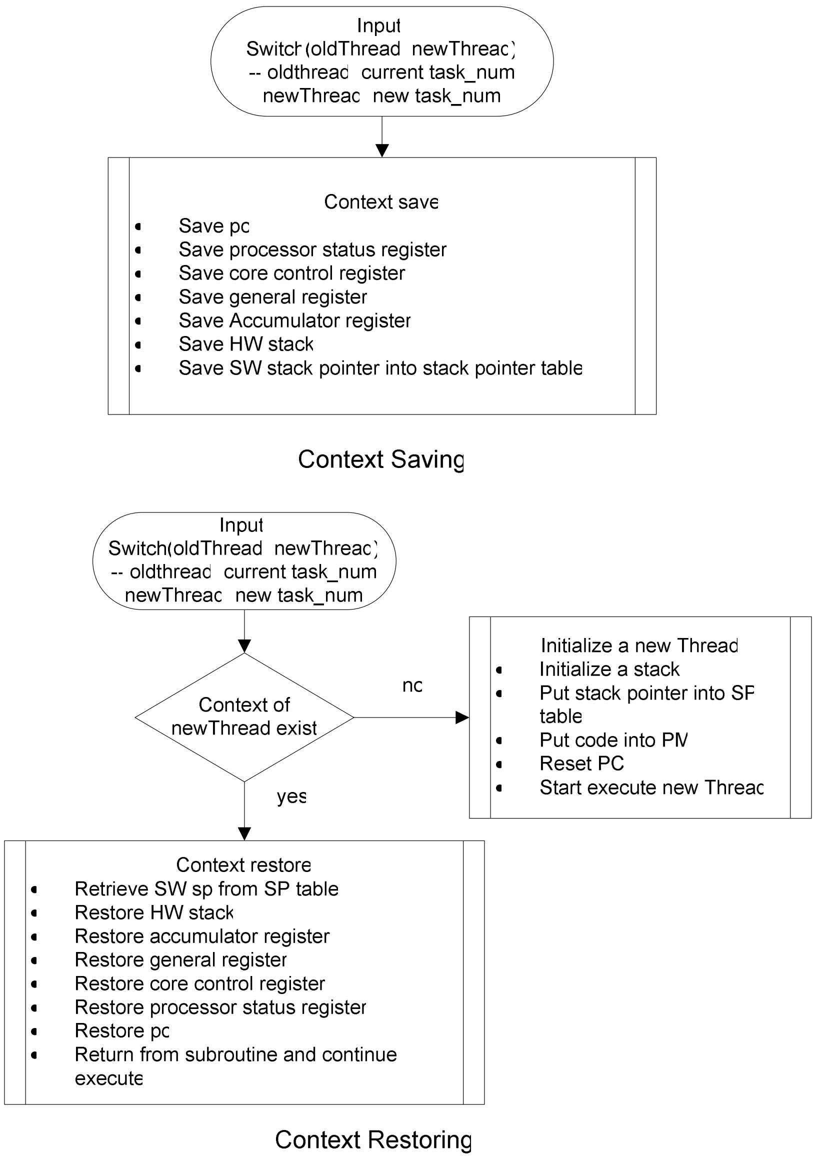

Input
Switch (oldThread newThreac)
-- oldthreac current task_num
newThreac new task_num
Context save
Save pc
Save processor status register
Save core control register
Save general register
Save Accumulator register
Save HW stack
Save SW stack pointer into stack pointer table
Context Saving
Input
Switch(oldThread newThreac)
-- oldthreac current task_num
newThreac new task_num
Context of newThread exist
nc
yes
Initialize a new Thread
Initialize a stack
Put stack pointer into SF table
Put code into PM
Reset PC
Start execute new Threac
Context restore
Retrieve SW sp from SP table
Restore HW stack
Restore accumulator register
Restore general register
Restore core control register
Restore processor status register
Restore pc
Return from subroutine and continue execute
Context Restoring

Figure 5-5 Detailed Context Switching steps

The Context Restoring has two alternative situations. When the new thread is the first time running in the processor, there will be no context stored in its software stack, so the Context Restoring is actually the Initialization steps for a new thread. In fact, it is the dirty work that **Create()** function does, which we can see from Figure 5-5. On the other hand, if a thread has already had Context saving in its software stack, the Context Restoring is definitely the opposite progress of Context Saving. The way to distinguish a thread having context or not is simply to check its relative stack pointer table is empty or not.

Chapter 6

Offline Optimizer

To run multiple tasks, timing should be checked by an Offline Optimizer (OO). The offline optimizer offers the functions such as **Stimuli()** for generating the task coming signal, **Cycle counting** for checking the run time cycle cost, **WCRT** (worst case run time) checking to make sure tasks meet their deadline, etc. The OO also contains a clock *cycle* as global variable so the time cost can be measured with strict temporal accuracy.

6.1 Stimuli – task coming detector

We have discussed in chapter 5, the whole processing is divided into many small executing iteration period. The interval of each small period is decided based on the smallest time unit of all the tasks. Keeping this small time unit as the interval can guarantee the **Stimuli()** has the capacity to discover all the coming tasks on time when they arriving, since all the tasks will be arrived at integral times of this small period.

The inputs of **Stimuli()** are system clock *cycle* (**stats->userTicks**), task arriving time and system frequency. As a global variable, the clock keeps counting the time passed. When the time is up to a new execution iteration period, the **Stimuli()** will be called. It will do a iteration checking of all the tasks to see which one or more will be arrived at this time, put the arriving tasks into the ready queue (**schqueue**) in sequence of their priority, then return the number of these arriving tasks.

The **Stimuli()** is an asynchronous program only used to generate the signals of task interrupt for the **Cycle counting**.

Figure 6-1 gives the execution process of **Stimuli().**

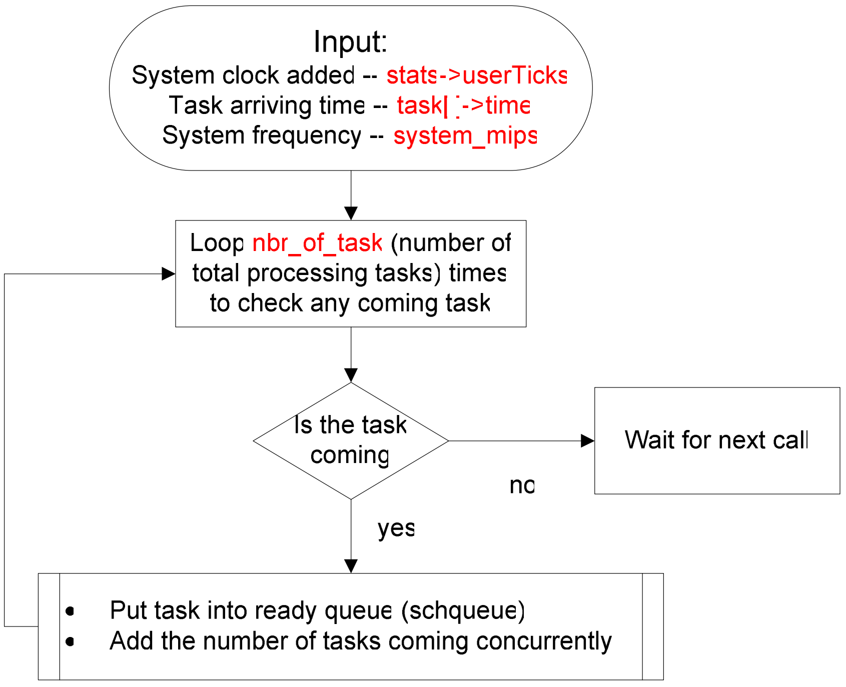

Input:
System clock added -- stats->userTicks
Task arriving time -- task[ ]->time
System frequency -- system_mips
Loop nbr_of_task (number of total processing tasks) times to check any coming task
Is the task coming
no
Wait for next call
yes
Put task into ready queue (schqueue)
Add the number of tasks coming concurrently

Figure 6-1 execution process of **Stimuli()**

6.2 Cycle counting

The main function of *Cycle counting* is to check the run time cycle cost of each task and check the correctness implementation of *super mode*, so it is simply designed as a run time cycle counter for each task. The pre-defined cycle cost of each task is configured in the *configuration part*, as each task running in the processor, the only function they executing is down counting "-1" within each cycle, until the total pre-defined cycle become zero, the task finishes. If a task is finished, *Cycle counting* will set the flag **threadToBeDestroyed** with **currentThread**, then call function **currentThread->Finish()** so that another corresponding task will be executed since the next cycle.

Meanwhile, for preventing hardware resources collision, the *Cycle counting* offers a mechanism **Semaphore** for protecting code critical regions so that certain hardware can be locked when a running task uses it. A **Semaphore** has two operations: **P ( ) –** decrement the semaphore's count, blocking the caller if the count is zero; **V ( )** – increment, waking up a thread waiting in **P ( )** if necessary.

As the lowest priority task is finished meaning all the tasks completes their jobs, then the *Cycle counting* will give the report of total cycle cost of the DSP core. This value should be less than the deadline. Otherwise, re-scheduling is needed: either increasing the system frequency or cutting off some tasks, and this function is provided by *WCRT checking.*

Figure 6-2 shows *Cycle counting* processing steps. In the beginning, the *background task* is initialized as the first task running in the processor. The function **Create (VoidFunctionPtr func, int arg)** will invoke the *Cycle counting* as the function pointer and pass the parameter **[task_n]** which is the task serial number in the *Task* array.

During each small executing iteration period, super mode will check any interrupt signal generated by interrupt handler once a cycle, if no interrupt, the *Cycle counting* will execute one instruction that actually down count "-1" of the task.

As a new small executing iteration period start, super mode will check any other higher priority task coming, the detailed process has been discussed in *Task Management Process* part in chapter 5.

The *Cycle counting* will execute the highest priority task until it finishes, then super mode will check any waiting tasks in the ready queue, remove the highest priority task in the queue now and make it as the *currentThread* to start executing it.

The super mode will examine if there is any task has not yet coming even when the *background task* had finished. In case there is one, *Cycle counting* will quickly advance clock cycle with doing nothing until the time that task arriving, then start executing it.

When all the tasks have finished, last but not least, the total DSP core cycle cost should be checked to be less than the deadline. This function is covered by ***WCRT** (Worst Case Run Time) checker*. If the cycle cost is less than the deadline, then we got a successful system. Otherwise, we need to reschedule the system by either increasing the system frequency or cutting off some tasks. Here we pick the choice of increasing the system frequency because the tasks have defined by customer. We also reset the tasks pre-defined value so they can be down counting again, reset system clock cycle to zero, and then do the whole processing loop again.

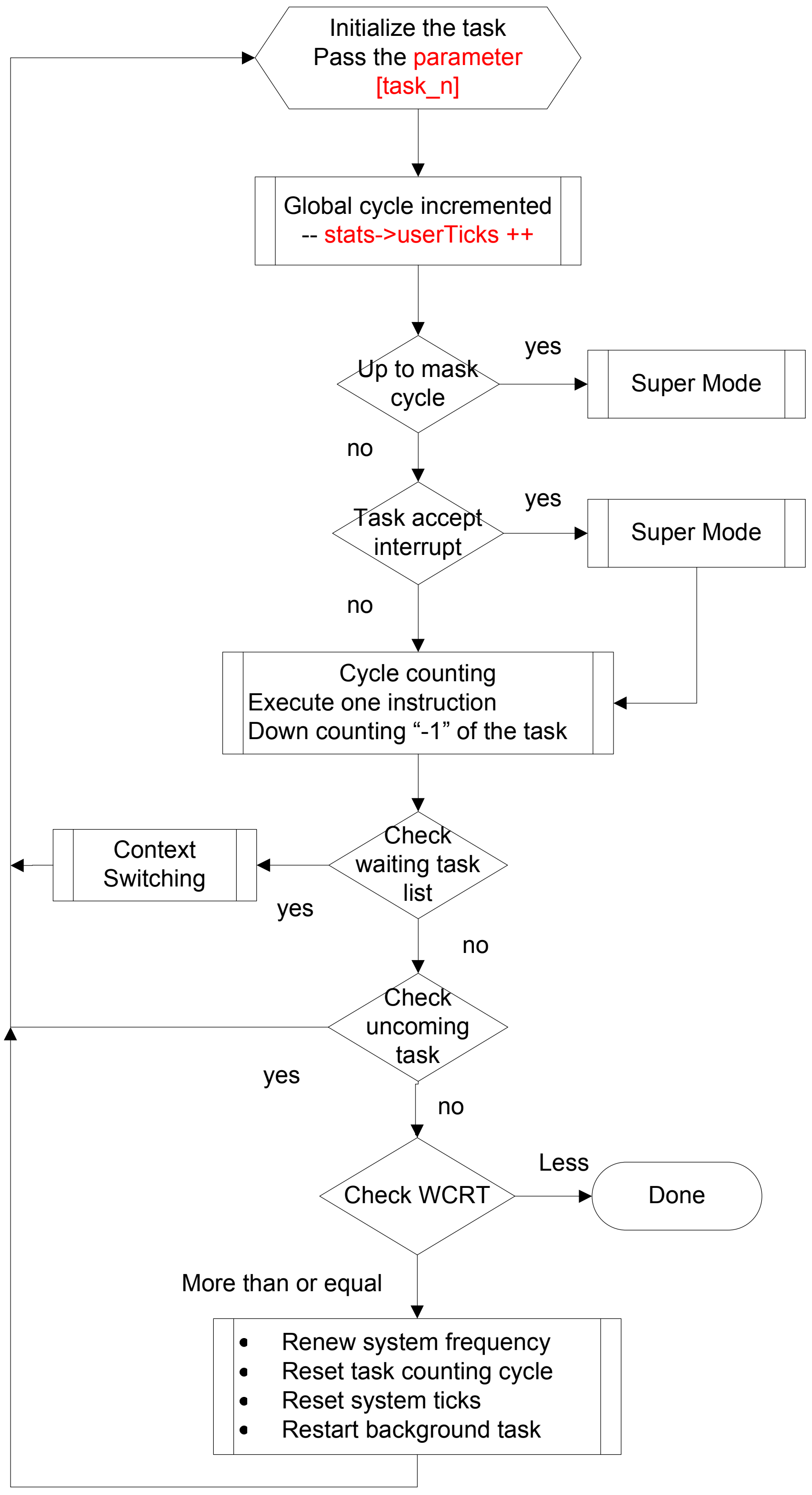


Figure 6-2 Cycle counting processing

Chapter 7

Simulations and discussions

7.1 Timing validation

With Offline Optimizer, the timing consumption can be checked for the super mode. Besides the cycle cost for running the defined task with OO, the other overheads of the super mode are coming from serving the ISR of I/O interrupts, timing cost on Context Switching, and little cycle cost generated by masking all the interrupts in the end of each small iteration executing period.

Following is a test example to compare the run time cost with results from static schedule of offline optimizer. The predefined tasks are listed in table 7-1.

Task Number	Arriving Period (ms)	MIPS (E6 MHz)	Predefined counting value
1	3	33	99
2	7	59	413
3	13	135	1755
4	21	543	11403

Table 7-1 An example of test tasks

The system frequency is decided to be 847MHz by the super mode corresponding to the data in table 7-1. A static schedule result from OO is listed below.

Response time for **Task4** is 14479 cycles, processed 5 Task1, 2 Task2 and 1 Task3
Response time for **Task3** is 11384 cycles, processed 5 Task1, 2 Task2
Response time for **Task2** is 6342 cycles, processed 2 Task1
Response time for **Task1** is 2640 cycles

Table 7-2 static scheduling result

```
The background task is task 4 with arriving period 21ms
system_mips is 847MHz

Background task start to run!

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 2642 cycles.

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 5183 cycles.

Current Thread is Task 04

Current Thread is Task 02

Task 2 still has 0 cycle need to finish
Response time for Task 2 is 6344 cycles.

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 7724 cycles.

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 10265 cycles.

Current Thread is Task 04

Current Thread is Task 03

Current Thread is Task 02

Task 2 still has 0 cycle need to finish
Response time for Task 2 is 12273 cycles.

Current Thread is Task 03

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 12806 cycles.
```

```
Current Thread is Task 03

Task 3 still has 0 cycle need to finish
Response time for Task 3 is 13298 cycles.

Current Thread is Task 04

Task 4 still has 0 cycle need to finish
Response time for Task 4 is 14598 cycles.

Task 1 has finished!

Task 2 has finished!

Task 3 has finished!

Task 4 has finished!

WCRT has been checked!

All tasks have finished ! Total Ticks used is 14598 equal to 17ms
Deadline of Background task 21ms
```

Figure 7-1 running time results with response cycle cost

The running results are shown in figure 7-1. The figure also illustrates the execution sequence of four tasks with response cycle cost. The total running cost is less than the deadline, fulfilled requirement. For comparing with the static scheduling result, the run time cycle cost for each task is collected in the table 7-3.

Response time for **Task4** is 14598 cycles, processed 5 Task1, 2 Task2 and 1 Task3
Response time for **Task3** is 13298 cycles, processed 5 Task1, 2 Task2
Response time for **Task2** is 6344 cycles, processed 2 Task1
Response time for **Task1** is 2642 cycles

Table 7-3 running time results collection

Compared Table 7-3 with Table 7-2, the total run cost is a little bit higher than static scheduling results due to the overheads mentioned in the beginning.

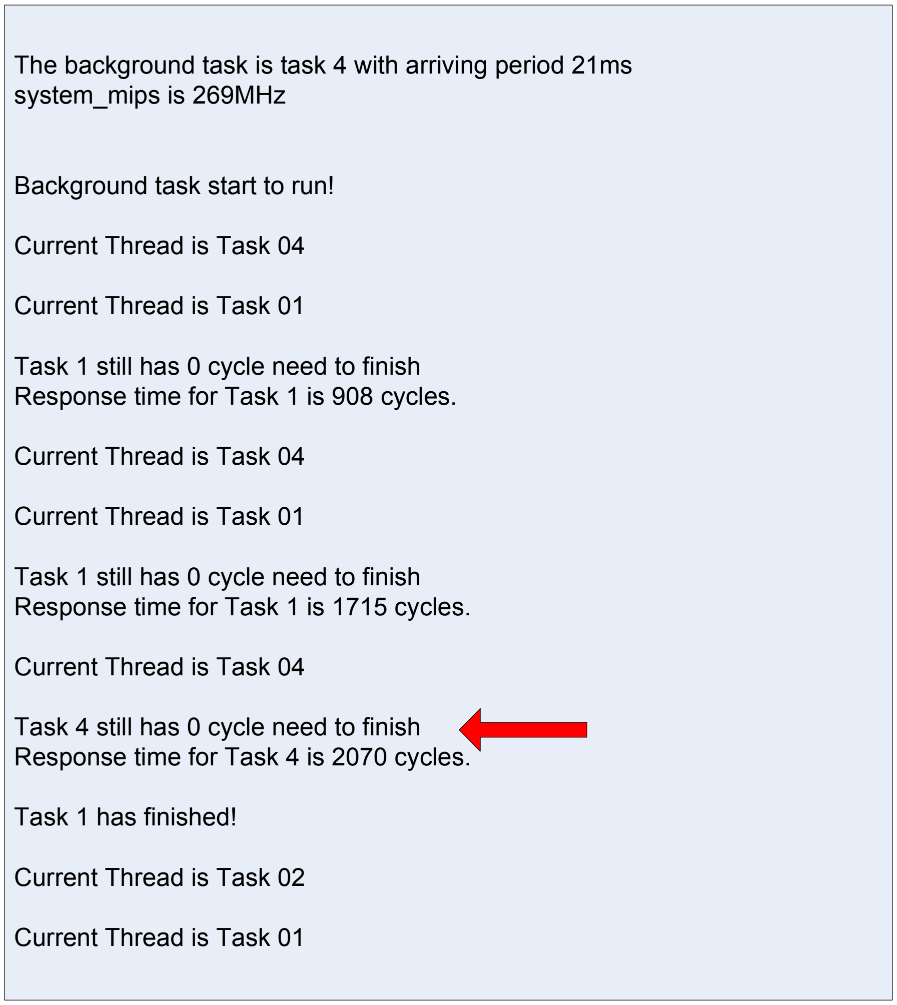
7.2 Function verification

From the task data in table 7-1 and execution results shown in figure 7-1, the basic functions of the super mode is tested and successfully verified, including background task identification, priority setting, task management and interrupt handler, context switching, and WCRT checking.

However, for better checking some corner cases like WCRT not fulfilled, or some task came later than the completion of the background task, we tested the system with another set of data shown in table 7-4.

Task Number	Arriving Period (ms)	MIPS (E6 MHz)	Predefined counting value	Further interrupt accept (1/0)
1	3	33	99	0
2	13	59	767	1
3	17	66	1122	1
4	21	87	1827	1

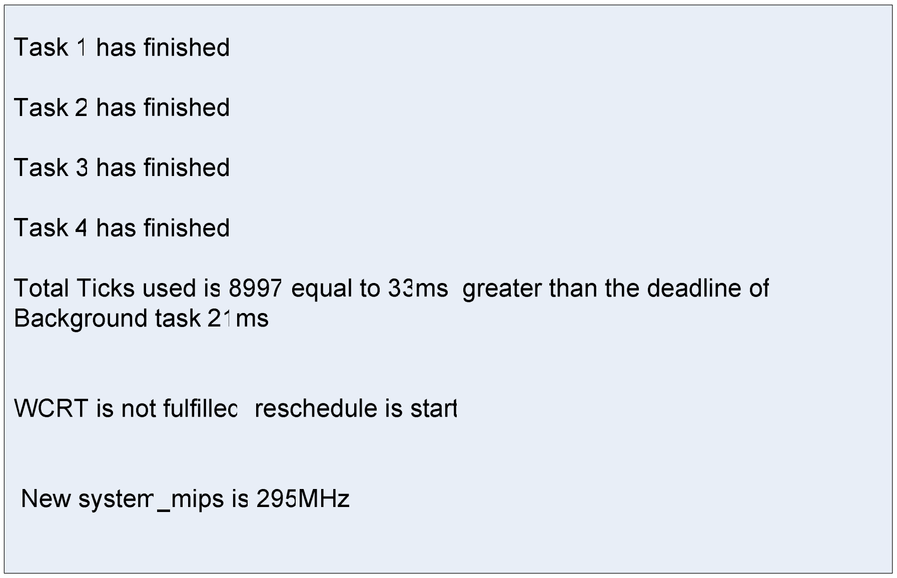
Table 7-4 An new set of test tasks



```
The background task is task 4 with arriving period 21ms
system_mips is 269MHz


Background task start to run!

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 908 cycles.

Current Thread is Task 04

Current Thread is Task 01

Task 1 still has 0 cycle need to finish
Response time for Task 1 is 1715 cycles.

Current Thread is Task 04

Task 4 still has 0 cycle need to finish
Response time for Task 4 is 2070 cycles.

Task 1 has finished!

Current Thread is Task 02

Current Thread is Task 01
```

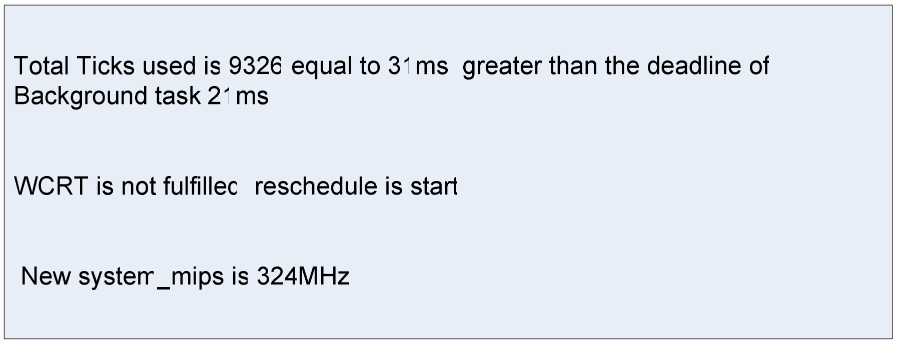
Figure 7-2 running results for checking late coming task

From the figure 7-2, we can see the background task has finished until the place pointed by the red arrow. However, there are still some tasks that have not come yet, so the system continue to execute until the late coming task arrives, then execute them.



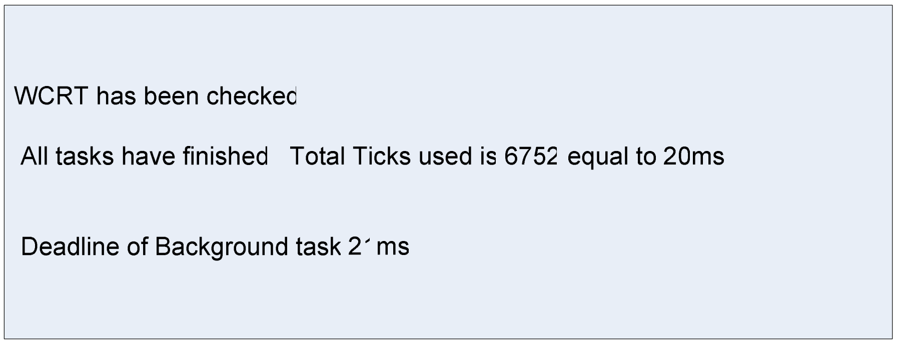
```
Task 1 has finished

Task 2 has finished

Task 3 has finished

Task 4 has finished

Total Ticks used is 8997 equal to 33ms  greater than the deadline of
Background task 21ms


WCRT is not fulfilled  reschedule is start


 New system_mips is 295MHz
```

Figure 7-3 WCRT checking failed first time



```
Total Ticks used is 9326 equal to 31ms  greater than the deadline of
Background task 21ms


WCRT is not fulfilled  reschedule is start


 New system_mips is 324MHz
```

Figure 7-3 WCRT checking failed second time



```
WCRT has been checked

 All tasks have finished   Total Ticks used is 6752 equal to 20ms


 Deadline of Background task 21ms
```

Figure 7-3 WCRT checking successfully

Three figures above illustrate the results for running WCRT checking for the data in table 7-4. At the beginning, the system frequency was not high enough, so the WCRT checking was failed and then system rescheduled. The same case still happened in second time. After two time of rescheduling, the WCRT checking passed successfully. We can see the running time cost is less than the deadline.

7.3 Assembly coding and Benchmark

As we discussed in Chapter 5, Context switching is inherently machine dependent. This project is supposed to run and be simulated on *Senior* DSP processor, so all the instruction sets of context switching are implemented using assembly language with constraints of *Senior* processor. Here we attached the assembly implementation and a benchmark for checking the correctness of the implementation.

```
;;-------------------------------------------------------------------------------------
;; Context Saving
;; r27 to r31 are the registers used in context switch
;;-------------------------------------------------------------------------------------

context_s
        move r27,fl0                ; save fl0 value into HW stack, which is processor status register
        nop
        nop
        push r27
        nop
        nop

        ld0 r28,(ar2,r30)           ; load SW stack pointer of thread1
        nop
        nop


        move ar0, r28               ; set ar0 with software stack pointer
        set step0,1
        nop
        nop
        st0 (ar0++),r0              ;store general register value
        st0 (ar0++),r1
        st0 (ar0++),r2
        st0 (ar0++),r3
        st0 (ar0++),r4
        st0 (ar0++),r5
        st0 (ar0++),r6
        st0 (ar0++),r7
        st0 (ar0++),r8
        st0 (ar0++),r9
        st0 (ar0++),r10
        st0 (ar0++),r11
        st0 (ar0++),r12


        move r27, rnd acr0          ; move the high part of ACR0
        move r28, rnd mul65536 acr0         ; move the low part of ACR0 by the way of shiftting 16
                                            ; bits of ACR0
        nop
        nop
        push r27
        push r28
        nop
        nop
        clr.ne acr0
```

```
            move r28, fl1                       ; check the value of HW stack pointer
            nop
            nop
            andn r28, 15
            nop
            nop
            move r1, r28                        ; save the value of HW stack
            nop
            nop


HWstack_s
            dec r28
            pop r0
            nop
            nop
            st0 (ar0++),r0
            nop
            cmp 0,r28
            jump.ne HWstack_s
            st0 (ar0++),r1                      ; save the value of HWstack,software stack has been saved
            move r28, ar0
            nop
            nop
            st0 (ar2,r30),r28     ; save stack pointer into table
            nop
            nop
```

Figure 7-4 Assembly coding for Context Saving

```
context_r

            ld0 r28 (ar2 r31)                    load SW stack pointer of new thread
            nop
            nop
            nop
            cmp 0 r28
            jump ne restore
;;-----------------------------------------------------------------------------------------------------------------------------
;; if new thread is run first time  we need to initialize a stack
;;-----------------------------------------------------------------------------------------------------------------------------
             ram0
thread2
             skip 1024

             code

            set r28  thread2
            nop
            nop
            st0 (ar2 r31) r28
            nop
;;-----------------------------------------------------------------------------------------------
;; restore the context from SW stack
;;-----------------------------------------------------------------------------------------------
```

```
restore
            move ar1, r28                   ; set ar1 with software stack pointer
            set step1,1
            nop
            nop

            ld0 r1, (--ar1)                 ; get the value of HW stack
            nop
            nop

HWstack_r                                   ; restore HW stack
            dec r1

            ld0 r0, (--ar1)
            nop
            nop
            push r0
            cmp 0,r1
            jump.ne HWstack_r

            pop r27                         ; pop the low part of ACR0
            pop r28                         ; pop the high part of ACR0
            nop
            nop
            move acr0.l,r27
            nop
            nop
            move acr0.h,r28
            nop
            nop

            ld0 r12, (--ar1)                ; restore general register
            ld0 r11, (--ar1)
            ld0 r10, (--ar1)
            ld0 r9, (--ar1)
            ld0 r8, (--ar1)
            ld0 r7, (--ar1)
            ld0 r6, (--ar1)
            ld0 r5, (--ar1)
            ld0 r4, (--ar1)
            ld0 r3, (--ar1)
            ld0 r2, (--ar1)
            ld0 r1, (--ar1)
            ld0 r0, (--ar1)

            pop r27                         ; restore processor status register
            nop
            nop
            move fl0, r27
            nop
            nop
            ret                             ; restore pc
```

Figure 7-5 Assembly coding for Context Restoring

For checking the correctness of this assembly code, we built a benchmark with two simple threads down counting a predefined number as their tasks, respectively. The thread 1 was supposed to be down counted 100 times, after that, thread 2 would interrupt thread 1 to start down counting 200 times. During this process, the thread 1 was supposed to do context saving before processor execute thread 2.

```
;;------------------------------------------------------------------------------------
;;Initialize the stack pointer table and thread 1
;;------------------------------------------------------------------------------------

        .ram0
stacktable
        .skip 32                    ;set a register for store the stack pointer


        .ram0
thread1
        .skip 1024                  ; initialize a stack for thread1


        .code

        set ar2,stacktable          ; ar2 is used for saving SP table address
        set r28, thread1            ; from r27 to r31 is the register used in context switch
        nop
        nop                         ; r28 is used for transfer data, r29 is stack pointer, r30 is current
                                    ; thread's number
                                    ; r31 is for new thread
        st0 (ar2,1),r28             ; put the stack pointer into table
        nop

;;------------------------------------------------------------------------------------
;;now let's start executing thread1
;;------------------------------------------------------------------------------------
        set r30,1                   ; indicate the number of thread which will be saved
        set r31,2                   ; indicate the number of new thread, these two number will be
                                    ; transfered as parameter from subroutine call procedure
        nop
        nop


        set r0,1000
        nop
        nop

count
        repeat label1, 100          ; set Loop times
        move acr0.h,r0

        dec r0
        nop
        nop
        move acr0.l,r0

label1

        cmp 1,r31
        jump.eq end
        call context_s              ; save pc value into hardware stack
        nop
        nop
        nop
        jump count
```

Figure 7-6 Context switching Benchmark 1

The second part of this benchmark is to check context restoring. After thread 2 execute 200 times of its iteration, the thread 1 interrupted the thread 2 and that caused thread 2 did context saving. Then thread 1 was executed again by doing context restoring from its software stack, and the processor returned from this ISR and kept on executing thread 2.

```
        .ram0
thread2
        .skip 1024

        .code

        set r28, thread2
        nop
        nop
        st0 (ar2,r31),r28
        nop

        set r0,5000
        nop
        nop
count2
        repeat label2, 200          ; set Loop times
        dec r0
        nop
        nop
label2

        set r30,2                   ; in second time,  the number of currentThread is 2
        set r31,1                   ; do context switch, which make currentThread become thread 1
        nop
        nop

        call context_s
        nop
        nop
        nop

        ld0 r28,(ar2,r31)           ; load SW stack pointer of new thread (thread 1)
        nop
        nop
        nop
        cmp 0,r28
        jump.ne restore

end
        out       0x13,r0
        nop                         ; finish switch with initialize new thread
```

Figure 7-6 Context switching Benchmark 2

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this master thesis, we design and implemented a super mode for multiple streaming signal processing applications, and got the timing budget based on Senior DSP processor. This work presented great opportunity to study the real-time system and firmware design knowledge on embedded system.

From pre-study of background knowledge down to final project implementation, every step was supervised under the way in which industry works. This experience gave me the chance to learn how to do a project in a professional way in Electronic Engineering area. This is very useful for me, since my profile was mainly on Electrical Engineering before I entered this master program.

8.2 Future Work

Although the major part of the super mode was implemented in this thesis work, there are still some parts can be improved like the memory management, and power consumption of the super mode. Since the super mode is designed for the modern handset or terminal, the memory management and power consumption are two very important parts for a successful implementation. But the purpose of this project is to design a general super mode that can be mapped into any mobile terminals, so these two parts will be further improved and optimized when implement the super mode into a specific mobile terminal.

Reference

[1] Laplante Phillip A. Real-time Systems Design and Analysis, 3rd Edition. John Wiley & Sons. 2004

[2] Dake Liu. Design of Embedded DSP Processors, compendium. 2007

[3] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts, 7th Edition. John Wiley & Sons. 2005

[4] Wikipedia, the free encyclopedia. 22 November 2007. <http://en.wikipedia.org/wiki/Deadlock>

[5] Ngolah C.F.,Yingxu Wang, and Xinming Tan. Implementing task scheduling and event handling in RTOS+. Electrical and Computer Engineering, 2004. Canadian Conference on
Volume 3, Issue , 2-5 May 2004

[6] FreeRTOS™ Homepage. < http://www.freertos.org/>

[7] ENEA OSE Datasheet

[8] Jason O'Broin. Real-Time Processes (RTPs) for VxWorks 6.0 White Paper. Wind River Systems, Inc. 2006

[9] TMS320C28x DSP/BIOS 5.31 Application Programming Interface (API) Reference Guide. Texas Instruments Inc. 2006

Appendix Offline Optimizer Source Code

```
// threadtest.cc
//          Including configuration procedure for tasks, background task identification,
stimuli, main function for thread management ( Core() )
//----------------------------------------------------------------------------------------------------

#ifdef THREADS

#include "copyright.h"
#include "system.h"
#include "synch.h"
#include "boundedbuffer.h"
#include <iostream>
using namespace std;

class Task
{
public:
    Task( int n, int t, int m, bool in){
        num=n;
        time=t;
        mips=m;
        itt_accept=in;
        total_c=t*m;
    }   int num;
    int time;
    int mips;
    int total_c;
    bool itt_accept;

};

Task *task[50];
int nbr_of_task;
int system_mips = 0;

int stimuli();
void Core(int task_n);



//----------------------------------------------------------------------------------------------------
//ThreadTest
//        get configuration information from customer, define the priority of tasks, find
out background task
//        initialize the first thread, execute it
//----------------------------------------------------------------------------------------------------
void
ThreadTest()
{

//----------------------------------------------------------------------------------------------------
// config procedure for tasks
//----------------------------------------------------------------------------------------------------


   cout<<"How many tasks do you want to input ? Number:"<<endl;
   cin>>nbr_of_task;
   int i;
   for(i=0;i<nbr_of_task;i++)
      {
         int number,time,mips;
         bool in;
         cout<<"Please input task number,arriving period(ms), MIPS(MHz),further
interrupt acceptance(1/0) for each task"<<endl;
         cin>>number>>time>>mips>>in;
         Task *t= new Task(number,time,mips,in);
         task[i]=t;
         system_mips +=task[i]->mips;
      }
   for(i=0;i<nbr_of_task;i++)
      {
         int j, t;
         Task *m;
         t=nbr_of_task-i-1;
         for(j=0;j<t;j++)
     if(task[j]->time > task[j+1]->time)
        {
            m=task[j];task[j]=task[j+1];task[j+1]=m;
          }
        }

    cout<<"The background task is task "<< task[nbr_of_task-1]->num << " with
arriving period "<< task[nbr_of_task-1]->time<<"ms"<<'\n';

    system_mips += system_mips/10;
    cout<<"\nsystem_mips is "<<system_mips<<"MHz\n";



//----------------------------------------------------------------------------------------------------
// initialize the background task and start progress
//----------------------------------------------------------------------------------------------------
    char tname[] = "Task";
    char *str = new char[strlen(tname)+4];
    sprintf(str,"%s %02d", tname, task[nbr_of_task-1]->num);
    interrupt->setStatus(UserMode);
    Thread *t=new Thread(str);
    t->Fork(Core,(nbr_of_task-1));

cout<<"current Ticks is "<< stats->userTicks<<"\n\n";
cout<<"Background task start to run!\n\n";

}


//---------------------------------------------------------------------------
// stimuli
//          called by Core() when userTicks equal to system_mips
//---------------------------------------------------------------------------
int stimuli()
{
    int i;
    int count = 0;
    for (i=0;i<nbr_of_task;i++)
        {
          if ((stats->userTicks % (task[i]->time * system_mips)) == 0)
       {
          schqueue->SortedInsert((void *)task[i], task[i]->time);
          count +=1;
       }
        }
```

```
   return count;
}



//----------------------------------------------------------------------------
// main function for thread, containing the TC inside
//----------------------------------------------------------------------------

void
Core(int task_n)
{
   cout<<"Current Thread is "<<currentThread->getName()<<"\n\n";
   while( task[task_n]->total_c != 0)
      {
//---------------------------------------------------------------------------------------------------
//From here is TC
//---------------------------------------------------------------------------------------------------
         if((stats->userTicks +5) % system_mips == 0)      // mask any threads or
interrupts for the new cicle to start
      {
         IntStatus oldLevel = interrupt->SetLevel(IntOff);

         stats->userTicks +=4;

         (void) interrupt->SetLevel(oldLevel);
      }

         if(stats->userTicks % system_mips == 0)   // check stimuli for any arriving
task and append them to arriving task list
      {
         int n = 0;

         if (n = stimuli())
            {
              Task *t;
              schqueue->SortedInsert((void *)task[task_n], task[task_n]->time);
              t = (Task *)schqueue->Remove();
              if(t != task[task_n])
           {
              int i,j;
              for (i=0;i<nbr_of_task;i++)
                 {if(t == task[i]) j=i;}
              char tname[] = "Task";
              char *str = new char[strlen(tname)+4];
              sprintf(str,"%s %02d", tname, t->num);
              Thread *h = new Thread(str);
              h->Fork(Core,j);    // set up a new thread

              IntStatus oldLevel = interrupt->SetLevel(IntOff);
              currentThread->Sleep();    // context switch, start the new thread
              (void) interrupt->SetLevel(oldLevel);
            }
             }
      }

          if(task[task_n]->itt_accept) // if the task can accept interrupt, check if there is
an interrupt happen in this tick
      interrupt->OneTick();            // advance one tick, and check if there is an interrupt
          else                                        // nothing happen, continue counting cycle
cost for thread
      {
         stats->totalTicks ++;
         stats->userTicks ++;
      }

          task[task_n]->total_c --;

       }
   cout<<"Task "<<task[task_n]->num<<" still has "<<task[task_n]->total_c<<" cycle
need to finish\n\n";
//----------------------------------------------------------------------------------------------------
//end of TC
//----------------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------------
// when one task finished , check if any tasks in the waiting list, if any, set up the
thread, invoke Core again
// until background task finished, end system
//----------------------------------------------------------------------------------------------------
   Task *t;

   t = (Task *)schqueue->Remove();
   if(t != NULL)
      {

          int i,j;
          for (i=0;i<nbr_of_task;i++)
```

```
        {if(t == task[i]) j=i;}
            char tname[] = "Task";
            char *str = new char[strlen(tname)+4];
            sprintf(str,"%s %02d", tname, t->num);
            Thread *h = new Thread(str);
            h->Fork(Core,j);    // set up a new thread

            currentThread->Finish();
            (void) interrupt->SetLevel(IntOn);
        }


//----------------------------------------------------------------------------------------------------
// check any task didn't arrive before the background task finished
//----------------------------------------------------------------------------------------------------
    int i;
    for(i=0;i<nbr_of_task;i++)
        {
            if(task[i]->total_c !=0 )
        {
           stats->userTicks = task[i]->time * system_mips;
           stats->totalTicks = task[i]->time * system_mips;
           schqueue->SortedInsert((void *)task[task_n], task[task_n]->time); // put
currentThread (background task) into schqueue again

           char tname[] = "Task";
           char *str = new char[strlen(tname)+4];
           sprintf(str,"%s %02d", tname, task[i]->num);
           Thread *h = new Thread(str);
           h->Fork(Core,i);    // set up a new thread
           IntStatus oldLevel = interrupt->SetLevel(IntOff);
           currentThread->Sleep();    // context switch, start the new thread
           (void) interrupt->SetLevel(oldLevel);

        }
            else cout<<"Task "<<task[i]->num<<" has finished!\n\n";
        }

//----------------------------------------------------------------------------------------------------
// check WCRT is more than deadline or not
//----------------------------------------------------------------------------------------------------

    if (stats->userTicks >= (task[nbr_of_task-1]->time * system_mips))
        {
```

```
        cout<<"Total Ticks used is "<<stats->userTicks<<" equal to
"<<stats->userTicks/system_mips<<"ms, greater than the deadline of Background
task "<<task[nbr_of_task-1]->time<<"ms \n\n\n";
        cout<<"WCRT is not fulfilled, reschedule is start\n\n";
        system_mips += system_mips/10;


        cout<<"\n New system_mips is "<<system_mips<<"MHz\n";
        int j;
        for(j=0;j<nbr_of_task;j++)
    task[j]->total_c = task[j]->time * task[j]->mips;

        char tname[] = "Task";
        char *str = new char[strlen(tname)+4];
        sprintf(str,"%s %02d", tname, task[nbr_of_task-1]->num);

        Thread *h=new Thread(str);
        h->Fork(Core,(nbr_of_task-1));

        stats->userTicks = 1;
      }
   else
      {
        cout<<"\nWCRT has been checked!\n\n";
        cout<<" All tasks have finished ! Total Ticks used is "<<stats->userTicks<<"
equal to "<<stats->userTicks/system_mips<<"ms \n\n\n";
        cout<<" Deadline of Background task "<<task[nbr_of_task-1]->time<<"ms
\n\n\n";
      }
}


#endif    //THREADS
```