Implementation of a Slack Stealing Algorithm for a Time Triggered Cooperative Scheduler (TTCS) Embedded System

Submitted by : Suguna Thanagasundram
Project Supervisor: Michael J Pont

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Informations & Communications Technology
in University of Leicester,2003

Acknowledgements

I would like to take this opportunity to thank my project supervisor Dr. Michael. J. Pont for his guidance, patience and knowledgeable insight that he has provided me during the course of this project. He has spent a lot of time through project meetings, weekly, progress reports and prompt email replies to all the constant queries of his project students despite the fact that he had six Masters students to supervise. He has been very supportive in making every bit of this project a success. Hats off to him.

I am also grateful to all my Lecturers in University of Leicester who have taught me all the subject modules during the one year of my stay here. I am also especially grateful to Dr. Alan Stocker, my Personal Tutor.

Finally, I am forever indebted to my parents for their endless support and encouragement.

Abstract

The focus of this thesis is to show how periodic tasks with hard deadlines and tasks with soft deadlines can be scheduled jointly using a time triggered cooperative approach. Although there have been many algorithms proposed by many researchers in the past, they are mainly for the preemptive case. For the first time, this problem has been addressed for a time driven non-preemptive scheduler and uses the concept of "slack stealing". The original Time Triggered Cooperative Scheduler (TTCS) developed by Pont (2001) [ref.1] was extended to incorporate slack stealing. The slack stealing algorithm proposed efficiently computes the slack available and makes optimal usage of the slack available to schedule the tasks with soft deadlines. The main development was done in C using a Keil compiler over a period of two months and was tested on a MCB900 evaluation board with a 89LPC932 Philips chip. The TTCS scheduler with Slack Stealing's performance was measured against a purely TTCS scheduler and the experimental results are presented for the CPU loading, memory and power consumptions usage. The TTCS scheduler with Slack Stealing was also tested for release "jitter". Finally applications where this TTCS with slack stealing are beneficial and where they are not are discussed.

Table of Contents

Chapter

List of Tables and Figures

Chapter 1

1. Introduction

1.1 Overview

Many real time applications today have a mixture of tasks with hard and soft timing constraints. In this project, the tasks with hard real-time timing constraints were modeled as periodic tasks. The periodic tasks have regular arrival times, execute critical control activities and have hard deadlines. The deadline is hard if failure to produce the correct result by the deadline may cause catastrophic results and soft otherwise. Typical of such tasks would include safety critical systems such as avionics, air traffic control and defense control. Tasks with soft timing constraints and which normally run in the background were modeled as slack tasks. Typical slack tasks would include tasks such as those of sending out or receiving data through the RS232 communications, LCD applications and batch processing applications. These tasks have random arrival times and normally have soft deadlines. If all the hard deadline periodic tasks can observe their deadlines, then slack time > 0. If the converse is true, i.e. slack time < 0, then not all hard deadline tasks can observe their deadlines. A scheduler which would want to jointly schedule tasks with both hard and soft timing constraints would want to guarantee that all hard deadlines are met and as well as complete the tasks with soft deadlines as soon as possible to improve their response times. It was the objective of this project to address this issue for a TTCS (Time Triggered Cooperative Scheduler). The TTCS developed by

Pont (2001) [ref.1] was further developed to handle both these types of tasks through the concept of "slack stealing". The key point about the proposed TTCS scheduler with Slack Stealing is that it makes scheduling decisions dynamically and hence even if the periodic tasks exhibit characteristic such as release jitter, the performance of the scheduler is not adversely affected.

1.2 Problem Statement

A TTCS schedules hard deadline tasks periodically using a time-line approach. When a task is ready to run, it is added to the waiting list. At the generation of next timer interrupt and if there is a ready task, the current waiting task is executed. The task runs to completion and then returns control to the scheduler. Because of this, TTCS schedulers are reliable and predictable.

Now however most of these periodic tasks may have an execution time less than the timer interrupt interval and because of this the CPU remains idle for some time after executing the task till the generation of the next timer interrupt. This is what is known as the "**slack"** time. It is possible to execute the lower priority tasks with soft deadlines in this "slack" time. This project will implement a **slack-stealing algorithm** for a TTCS in a single processor environment.

However there are certain problems associated with scheduling the periodic tasks with hard deadlines together with tasks with soft deadlines as the requirements imposed by these tasks are different.

1) The higher priority periodic tasks have hard deadlines so it has to be ensured that none of these periodic tasks miss their deadlines. And these tasks cannot be delayed and have to finish executing within their deadlines. If they are delayed, they are considered to be executing "jitter". This is what is known as testing for "jitter" as later illustrated in the report.
2) For the lower priority tasks with soft deadlines it is desired to schedule them as soon as possible to improve their response times.

Therefore the proposed scheduler has to efficiently balance the different requirements imposed by these tasks while ensuring that the system is not overloaded and has sufficient resources.

1.3 Research Scope and Contributions

By implementing a "slack stealing algorithm for a TTCS, it was envisioned that the TTCS developed by Pont (2001) [ref.1] could handle both periodic with hard deadlines and tasks with soft deadlines simultaneously. The concept of slack scheduling can be extended for a multiprocessor environment as shown by [ref.12] and [ref.16] but the intention here was to only concentrate on cooperative time driven uniprocessor scheduling. Event triggered and non-cooperative or preemptive scheduling are all out of the scope of this project. It was assumed that all the periodic tasks have higher priority and have hard deadlines. The tasks with soft deadlines all have lower priority. Hence it doesn't matter if these tasks do not finish executing within their allocated time.

The benefits gained by doing so are:

1) Previously, the scheduler remains idle (SCH_Go_To_Sleep) when it has nothing to do. Now it does some useful work by servicing these tasks with soft deadlines, like a printing job or sending some data through the RS232 link.
2) Previously, the tasks with soft deadlines had to wait until the higher priority periodic tasks have finished executing and then they are run in the background. Hence their response time was poor when the periodic load was high. Now by concurrently running them as slack tasks with the periodic tasks, their response times are improved.

1.4 Thesis Organization

This thesis is organized as follows.

Chapter 2 introduces the TTCS architecture in detail, major concepts and problem formulation. A detailed discussion of related work is given at the end of the chapter.

Chapter 3 describes the rationale of the design for the TTCS scheduler with Slack Stealing and presents the algorithm itself in the form of a flowchart with a specific example to illustrate the algorithm. This chapter also discusses the major features of the TTCS scheduler with Slack Stealing.

Chapter 4 talks about the implementation of the algorithm on the MCB900 board.

Chapter 5 discusses the testing strategy and does a comparison test between a purely cooperative TTCS scheduler and a TTCS with Slack Stealing in terms of CPU loading, power consumptions, memory usage.

Chapter 6 discusses the applications where the proposed TTCS scheduler with Slack Stealing may be useful and where they are not.

Chapter 7 finally briefly summarizes the technical achievements of the project and directions for future work.

In the final chapter, some concluding remarks have been presented.

Chapter 2

2. Background Theory & Related Research

In this chapter, fundamental concepts central to this project, a Time Triggered Cooperative Scheduler (TTCS) and slack will be investigated. Then related work will be reviewed and an analysis on how others have approached this problem will also be presented.

2.1 What is a TTCS (Time-Triggered Cooperative Scheduler)?

A scheduler is part of an operating system that decides which task is to run next. This decision is based on the readiness of each task, their relative priorities, and the specific scheduling algorithm implemented.

The time-driven scheduling approach is the most widely used approach in existing time critical and safety critical systems. TTCS is a time-triggered approach to scheduling of tasks (also often known as static cyclic scheduling). Tasks are scheduled to run at specific times (either on a periodic or one-shot basis). When a task is scheduled to run, it is added to the waiting list. And when the CPU (Central Processing Unit) is free, the next waiting task is executed. The task runs to completion and returns control to the scheduler. The other equally viable and attractive alternative to TTCS is to use events and this is known as Event Triggered Cooperative Scheduling (ETCS). Instead of a cyclic system calling a set of tasks sequentially, the tasks are activated by events. The scheduler is responsible for ensuring that the highest priority read-to-run task actually gets to execution on the processor. But in this project it was mainly limited to TTCS as the intention was to develop slack scheduling for this type of scheduling. The majority of embedded real-time software is still implemented using time-triggered scheduling today. This type of scheduler is simple and can be implemented in small amount of code. TTCS is typically just a simple timer poll that is required to synchronize with time, and the scheduler is merely a sequence of function calls. Because of this it's got low overheads: a single stack is used, and the stack depth is kept small. Even the memory allocation is done only for a single task at a time. This scheduler is written in a high level language like C or C++. It is not a separate application and it is often a part of the developer's code. The scheduler is hardwired into a piece of program code and is small and fast. With time-triggered scheduling there is no need to worry about concurrency control. Tasks run sequentially one after the other and therefore mutual exclusion is guaranteed. Hence TTCS is a simple concept, and so is immediately attractive, simple, predictable and safe compared to pre-emptive scheduling and is also easy to implement. For many simple systems this kind of scheduling approach is perfect.

2.2 Investigation of the TTCS Architecture in Detail

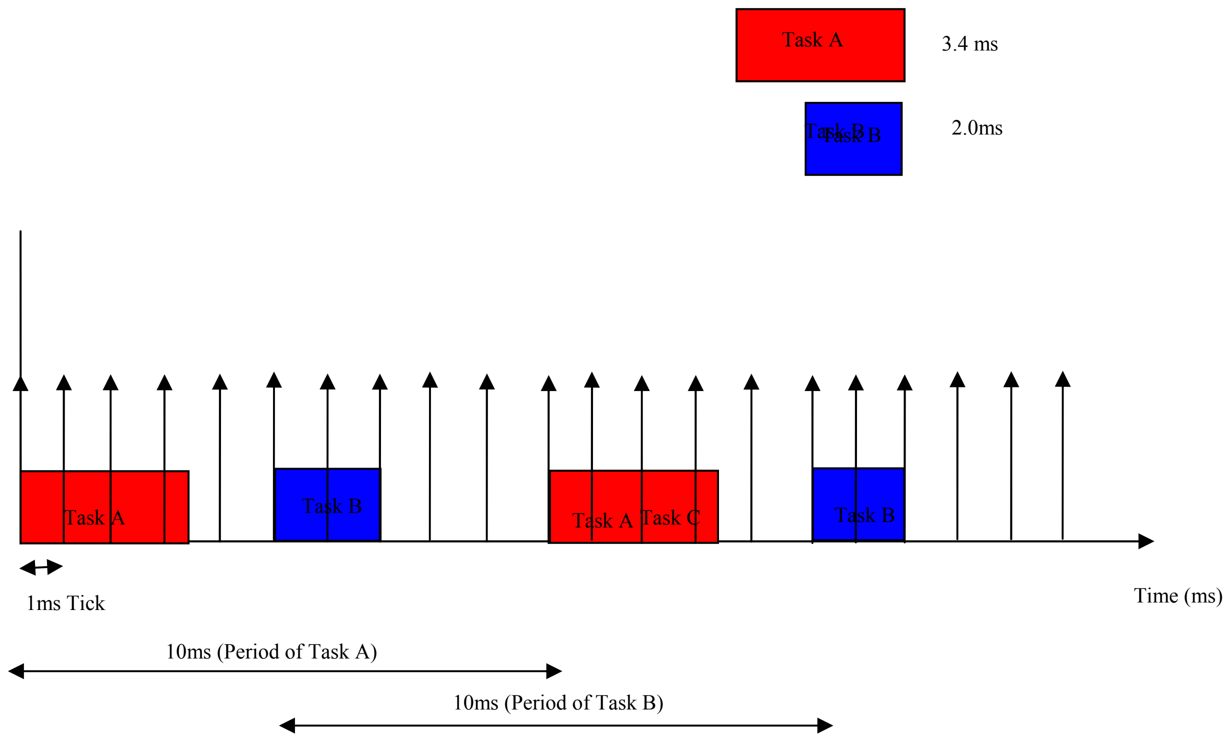

Figure 1 Time Triggered Cooperative Scheduler (Task Duration more than tick interval)

In TTCS, a timer is used to generate interrupts each time a timer overflow occurs. Most 8051 and 8052 microcontroller chips often come with at least two timers Timer 0 and Timer 1 and some extended architecture chips even have and extra Timer 2. By loading suitable initial values into the timer registers (for example if Timer 0 is used, TH0 and TL0), it can determined when the timer over flow is to occur. Hence this is called setting the tick rate. If a higher resolution scheduler is desired, then the tick rate should be set to the minimum possible. In this project Timer 0 was used. Timer 0 has been set in the 16-bit mode with manual reloading. The SCH_Init_T0() routine sets the Timer 0 tick rate.

Figure 1 shows a TTCS scheduler where the tick rate has been set to 1ms and two tasks, Task A and Task B are running periodically with a period of 10 ms and have task durations which are more than the tick interval. Task B is scheduled to run 5ms later than Task A. At each timer interrupt, the scheduler determines which is the next ready task to run. If there is a ready task, the scheduler executes that task. The task runs and at the next timer interrupt if the task is still running, the scheduler continues with the execution of the task. The key point to note is that the scheduler runs the task to completion before it executes another task. At some timer interrupts, there may be no tasks to run and the scheduler remains idles or does what is commonly known as going to sleep (SCH_Go_To_Sleep).

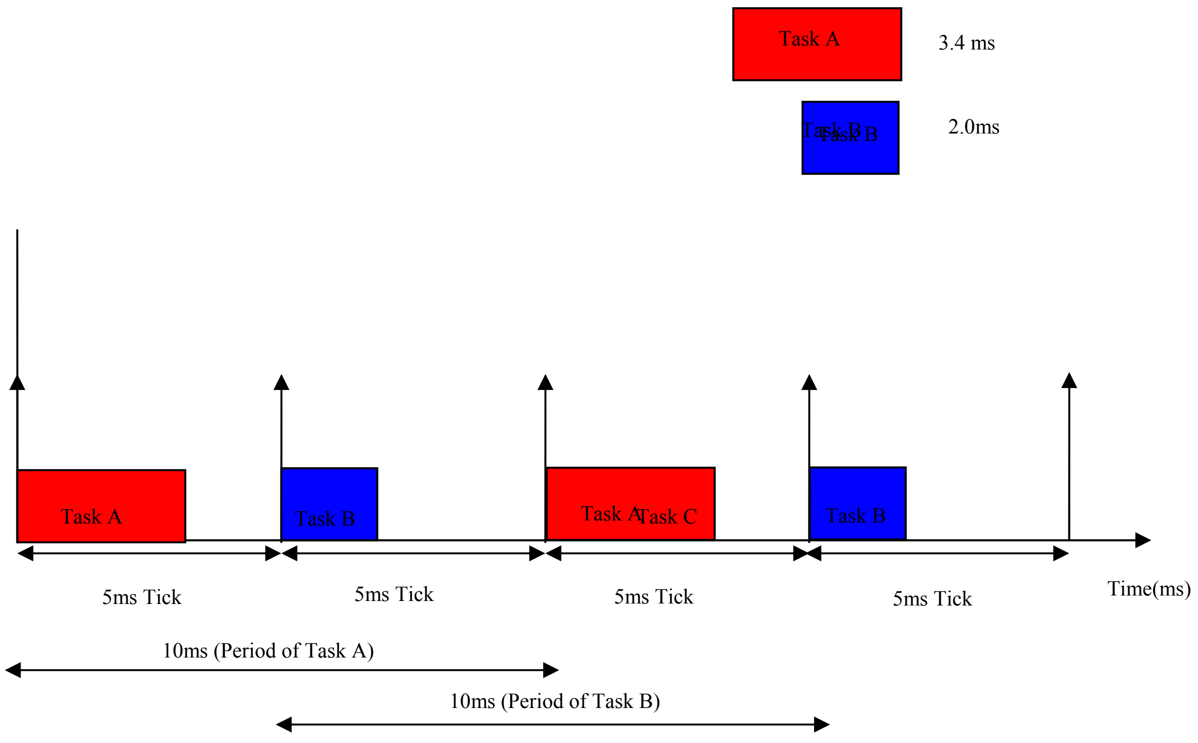

Figure 2 Time Triggered Cooperative Scheduler (Task Duration is less than tick interval)

Figure 2 shows a TTCS scheduler where the tick rate that has been set to 5ms and two tasks, Task A and Task B are running periodically with a period of 10 ms but have task durations which are less than the tick interval. Again at each tick interrupt, the scheduler determines the task to be run. For instance, in this case, the scheduler begins by running Task A. Task A has a task duration of 3.4ms but the tick interval is 5ms. Hence after finishing the execution of Task A, the scheduler has some remaining time in the tick interval before the next interrupt but since it has nothing to do it goes to sleep (SCH_Go_To_Sleep).

Whilst the TTCS scheduler can operate in both cases as shown by the figures above, it can only operate reliably with accurate timings in the latter case. And in all these examples, it was assumed that the task durations of the periodic tasks are the worst case execution times. This assumption allows us to guarantee that the deadlines of all the periodic tasks are met.

2.3 What is Slack Scheduling

Definition of Slack Time

It was the goal of this project to investigate slack scheduling for a Time Triggered Cooperative Scheduler (TTCS) embedded system. In the context of TTCS architecture, **slack time** is the amount of spare processing time available in the tick interval that can be used to perform some soft deadline lower priority task like sending some data down the RS232 link.

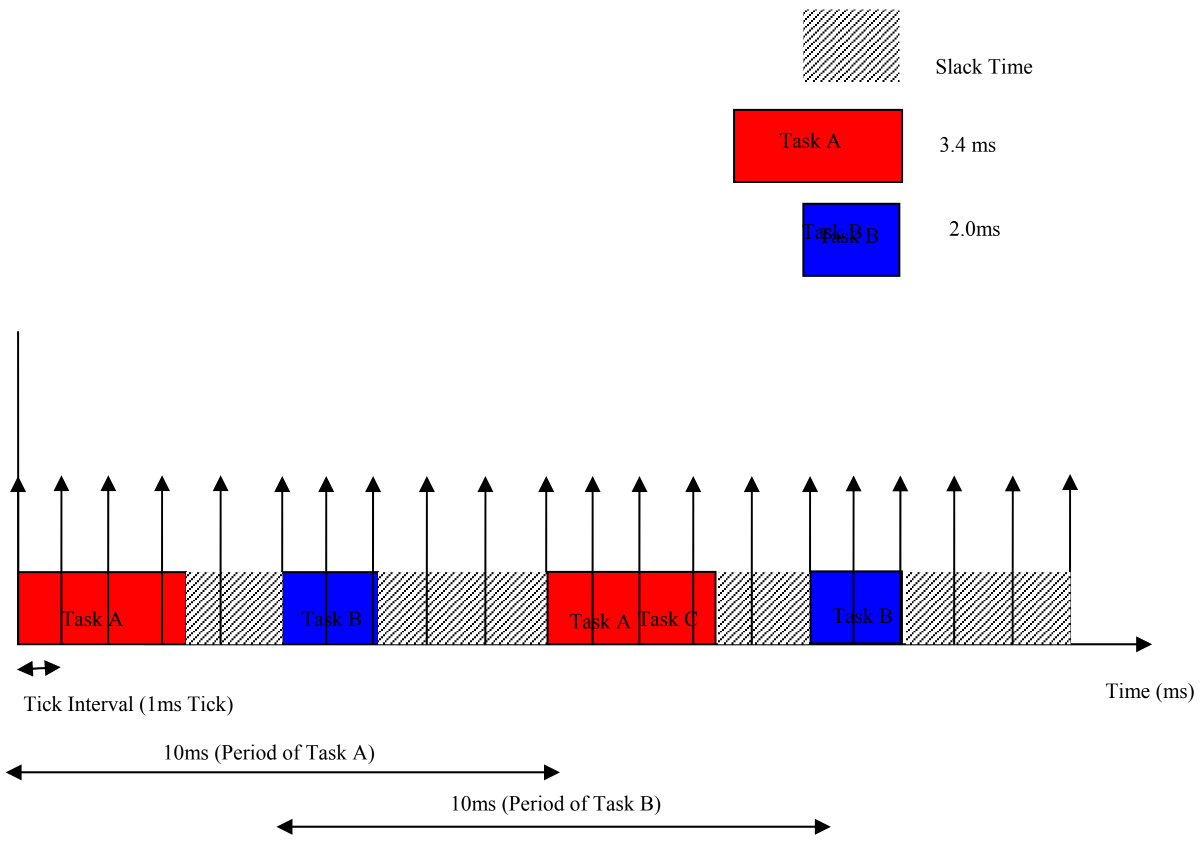

Figure 3 Slack time in a TTCS scheduler where task duration is more than tick interval

Figure 3 shows a TTCS scheduler that schedules two periodic tasks which have task durations longer than the tick interval (1ms). The available slack time in this scheduler is shown in gray. It can be seen that in some tick intervals there is no slack time when the tick interval is completely engaged in the execution of the task. In some tick intervals, only part of the tick interval is engaged in processing of the task and hence the remaining time is the slack time. It is also to be noticed that there are some tick intervals where no execution of tasks takes place and hence the whole tick interval is slack time.

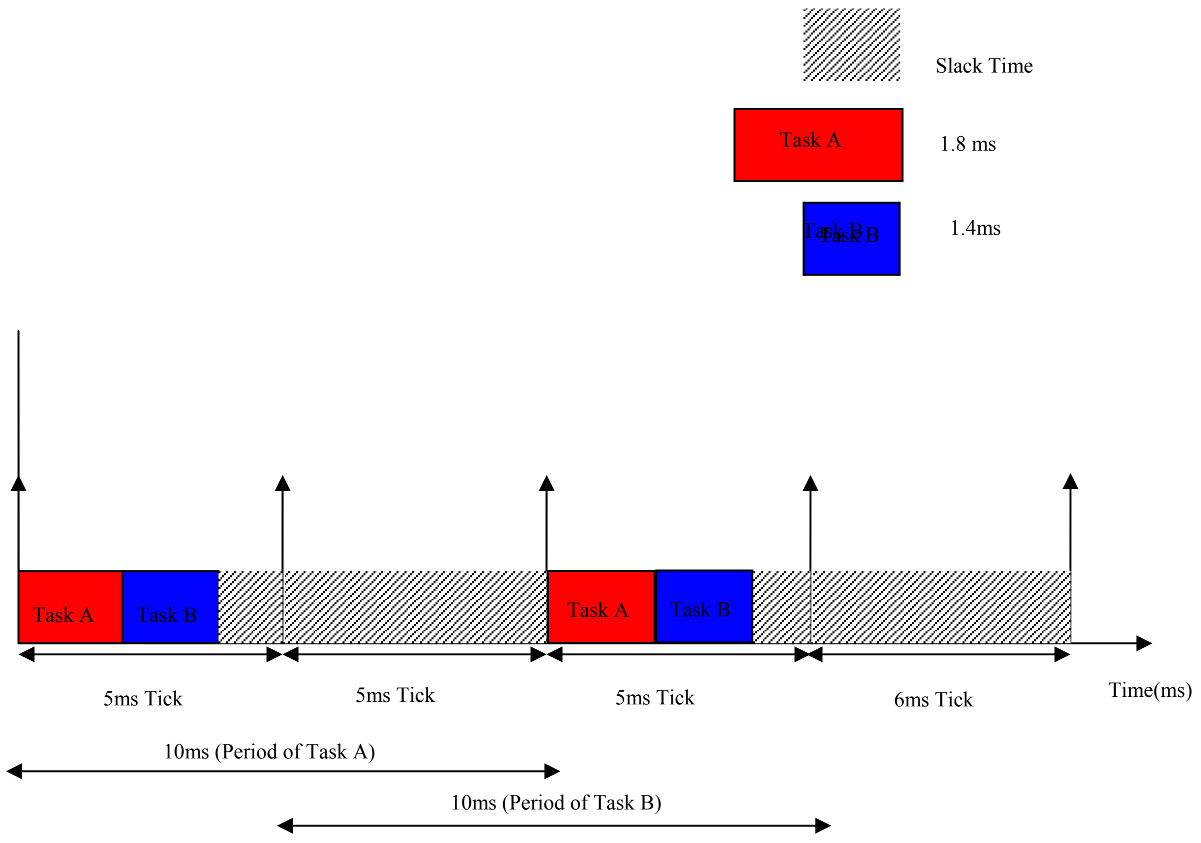

Figure 4 Slack time in a TTCS scheduler where task duration is less than tick interval

Figure 4 shows a TTCS scheduler which schedules two periodic tasks which have task durations less than the tick interval (5ms). The available slack time in this scheduler is shown in gray.

It was the objective of the project to implement a slack stealing algorithm which could handle both type of scenarios. Also in the context of this project, it was assumed that maximum slack time is the total idle time of the scheduler.

Slack Stealing Algorithm

A slack stealing algorithm can aptly be defined as a method or a piece of code which "steals" slack time from the tick interval in the context of a TTCS scheduler.

Slack time computation can be done in two ways. One way is the static approach [ref.3] where slack time from all the periodic tasks is computed before runtime and stored in a table. This approach is more efficient during runtime but requires more memory. The other way is the dynamic approach [ref.8] where slack time is computed during runtime. This approach is more time consuming during runtime but uses up less memory. In [ref.9] Davis also presents two approximate methods one static and another dynamic for determining slack which address the space and time complexity problems inherent in optimal slack stealing algorithms.

The dynamic approach was chosen as it was seen as more flexible. The amount of slack time available may vary from time to time depending on the workload for the scheduler. By choosing the dynamic approach, it is also possible to handle tasks with release jitter and synchronization requirements. Hence it was decided that the slack time will be calculated dynamically and the TTCS scheduler with Slack Stealing will make scheduling decisions about the execution of slack tasks online.

Slack task durations have to be known apriori before the scheduler can begin running and this can be achieved by either the user measuring the slack task durations and making this information available to the scheduler or the scheduler computing the slack task duration in an initialization routine. The second approach was adopted in this project as the timing of task durations may vary depending factors such as the specific chip used and oscillator frequency. By making the scheduler measure the task durations in an initialization routine it is possible to make the proposed TTCS scheduler with Slack Stealing more user friendly.

Scheduling tasks based on slack time requires cost metrics and in this case the task duration of the slack tasks was chosen as the criteria that determined whether or not the slack task will be run.

How Slack Stealing can cause Jitter

A key concern about executing slack tasks during the slack time is that they may cause "jitter" as illustrated by Figure 5. In this case the scheduler is executing a slack task in the available slack time and at the next tick interrupt, periodic Task B is scheduled to run. Hence the scheduler has to unload the slack task and load periodic Task B. This is the overhead in context switching and sometimes it is not negligible for large tasks. Sufficient allowance has to be given for the context switching otherwise periodic Task B maybe slightly delayed. This is when it said that Task B is exhibiting jitter.

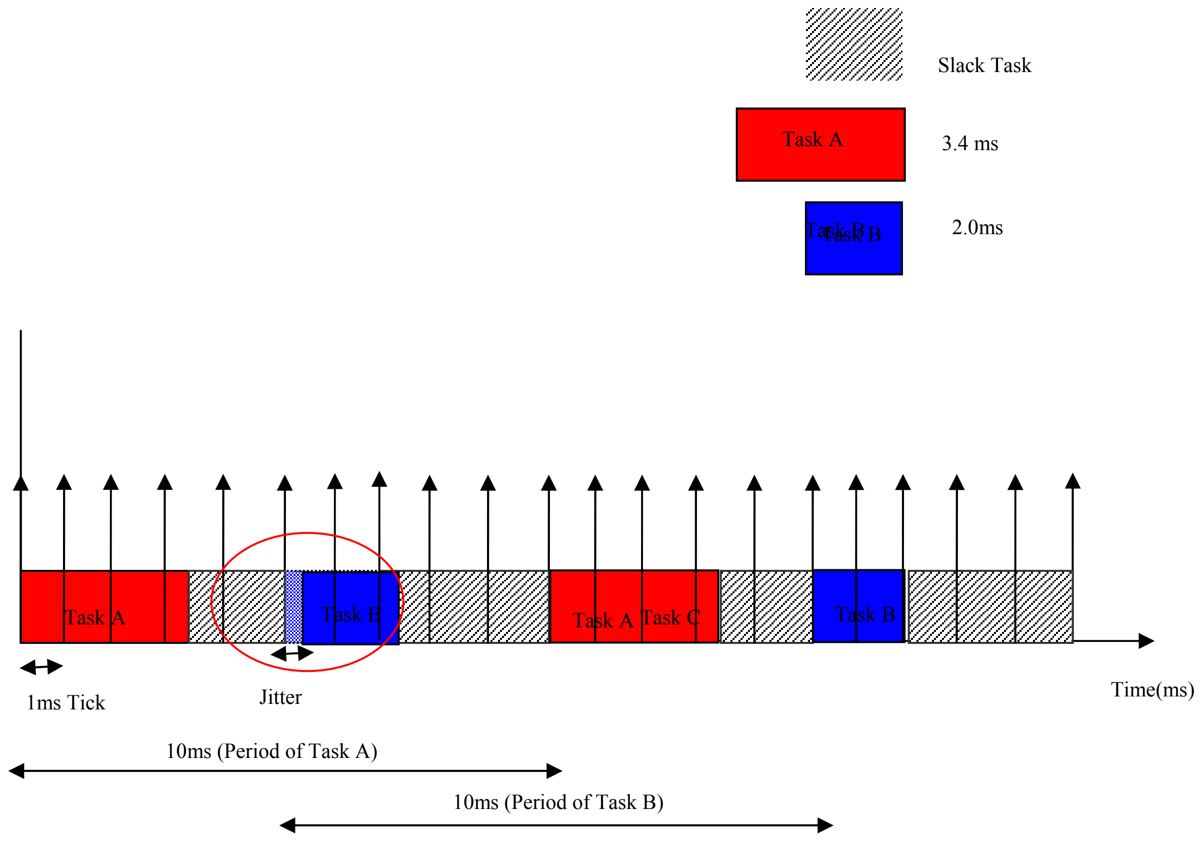

Figure 5 Slack Scheduling Causing Jitter

Suitable provisions have to be taken care of in the TTCS scheduler with Slack Stealing to prevent jitter problems.

Related Work

Scheduling hard real time jobs with soft real time tasks is a very command practice in embedded systems. In [ref.2] Kim has given a detailed example of how the response times of soft real-time end-to-end tasks can be enhanced while guaranteeing deadlines of hard real-time local tasks. In this paper, he talks in depth of a Distributed Control System (DCS) which consists of a Supervisory Control Computer (SCC) and multiple local cell controllers (LCC's). The LCC's and the SCC perform time critical low level control jobs and high level supervisory control jobs respectively. In such a system missing the deadlines of low level control loops may cause catastrophe because timely execution is directly related to the stability of the system. Hence these control loops are classified as hard real-time tasks. The control loops are configured by local I/O's and do not interact with other LCCS's. Hence they are called local tasks. The supervisory controls do not have catastrophic results by missing their deadlines but influence the throughput as their timely response minimizes unnecessary pause time and enhance quality of products. Thus the supervisory controls are classified as soft real-time tasks. The supervisory controls can be decomposed into 5 subtasks, hence they are also called end-to-end subtasks. Kim proposes a technique which is derived by timing the attributes of subtasks and then finding the highest possible priorities of subtasks using the slack of local tasks.

Some other methods suggested by past researchers for servicing hard deadline tasks and soft deadline tasks simultaneously are a deferrable server method [ref.13], priority exchange method [ref.14] and sporadic server [ref.17] method but it has been shown that the slack stealing method has substantial improvements in terms of performance compared to these methods as proven in [ref.8].

The concept of slack stealing is not new. This concept was introduced by Thuel and Lehoczky in 1992 [ref 3], [ref.4], [ref.5], [ref.6] and [ref.15] in which they provide a method for guaranteeing and scheduling aperiodic tasks with periodic tasks which are scheduled by RMS (Rate Monotonic Scheduler). In [ref 6], Thuel describes a mechanism for reclaiming unused resource time known as a slack reclaimer when a job doesn't use up all of its resource requirement.

Several slack stealing algorithms have been proposed in papers [ref.3], [ref.4], [ref.7] [ref.8] and [ref.9] but they are mainly for the preemptive case which schedule soft aperiodic requests at the highest priority level so that they can make use of the slack immediately to minimize their response times.

In [ref.10] T.S.Tia has showed how the assignment of priorities to aperiodic requests whenever there is slack can be used to service the aperiodic tasks in a non greedy manner It was shown in this paper that this method makes additional slack time available and can effectively reduce the aperiodic response times even further. But again this was for a preemptive scheduler.

In [ref.11] Alia presents a Slack Stealing Job Admission Control (SSJAC), a methodology for scheduling periodic firm deadline jobs with variable resource requirements subject to controllable Quality of Service (QOS) constraints.

Hence the concept of slack stealing and slack stealing algorithms have been well analyzed by many past researchers but none have approached the problem for a time-triggered scheduler which has been done in this project.

Chapter 3

3. Method/Design

The flowchart diagram in Figure 6 illustrates how the code for the main function of the TTCS scheduler [ref.1] was extended to allow incorporation of slack scheduling to service tasks with soft deadlines. Since the testing was to be done on MCB900 with a 89LPC932 chip, Timer 0 was set in the 16 bit mode and manually reloaded for tick rate generation. Timer1 was configured for baud rate generation since it was the intention to add one of the slack tasks for RS232 communications to send data from buffer to PC. Sending data through the RS232 link is normally considered to be a lower priority task with soft deadline since this type of task can be run in the background and failure to execute the task within the scheduled deadline does not have any serious consequence. Next an initialization routine was added in the main function to measure the slack tasks durations required by dispatcher. Now as discussed before, accurate information about the slack task duration is necessary to ensure that the scheduler works reliably and with the predictability as required and to avoid problems like release jitter. Then the scheduler is started and begins running.

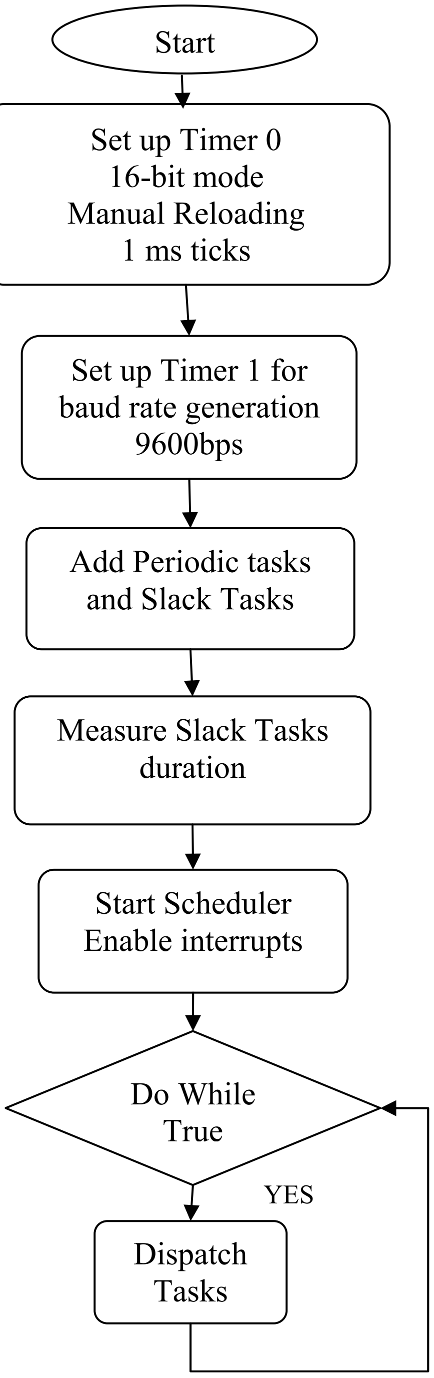

Figure 6 Flowchart of the main program

Now major changes were also done in the dispatch function (SCH_Dispatch_Tasks) routine to incorporate slack stealing as shown by the flowchart diagram in Figure 7. The dispatch function begins execution by running the periodic tasks if there are any ready. Then the function calculates the remaining slack time available in the tick interval. If the task duration of the first slack task is less than the slack time available, the dispatch function executes the first slack task. Having executed the first slack task or otherwise if the earlier condition is not true, the function calculates the slack time remaining and also checks whether there are any periodic tasks ready to run in the next tick interval. At this point, the scheduler can do take three actions depending on the circumstances.

1) If there is sufficient slack time available for the next task and there are no periodic tasks to run in the next tick interval, the scheduler runs the slack task.
2) If there is insufficient slack time available for the next task and there are no periodic tasks to run in the next tick interval, the scheduler adds the slack task's duration to the slack time available and runs the slack task
3) If there is insufficient slack time available for the next task and there are periodic tasks to run in the next tick interval, the scheduler stops running the slack tasks and goes to sleep. At the next tick interrupt, the scheduler is awaken from idle mode and begins processing the dispatch function from the beginning again.

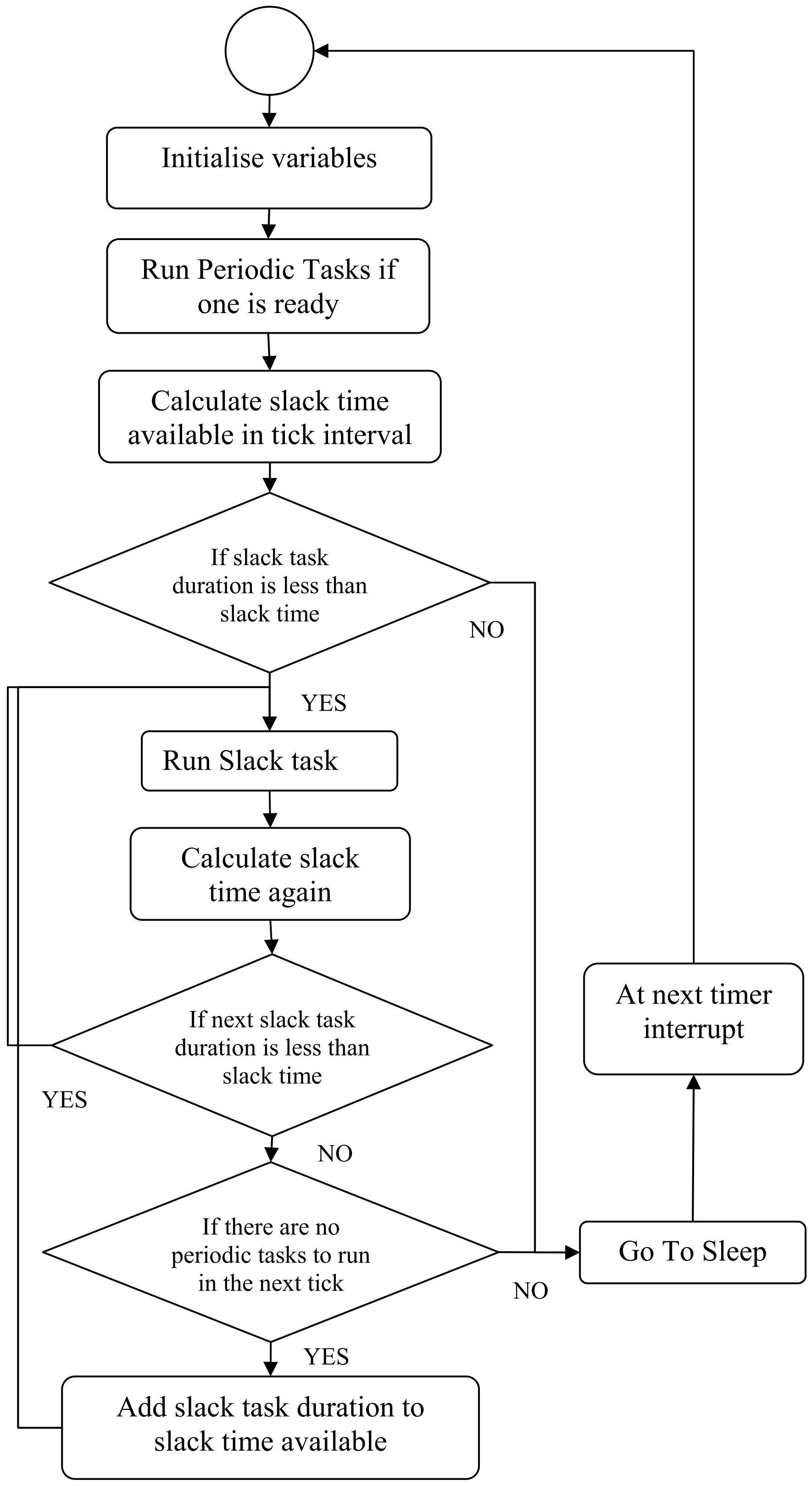
Initialise variables
Run Periodic Tasks if one is ready
Calculate slack time available in tick interval
If slack task duration is less than slack time
NO
YES
Run Slack task
Calculate slack time again
If next slack task duration is less than slack time
YES
NO
If there are no periodic tasks to run in the next tick
NO
YES
Add slack task duration to slack time available
Go To Sleep
At next timer interrupt

Figure 7 Flowchart of Dispatch function

The main features of the TTCS scheduler with Slack Stealing are:

1) It can run more than slack task. If there are three slack tasks to run, the scheduler runs slack task A, slack task B and then slack task C and again slack task A and slack task B in a round robin manner until it gives up control if there are periodic tasks to run.
2) The scheduler always gives higher priority to the periodic tasks. If there are ready periodic tasks to run in the tick interval, they are run first. Only thereafter are the slack tasks run.
3) All the slack tasks have to task durations less than the tick interval. $Duration_{Slacktask} < Tick\ Interval$. This condition has to be met for the scheduler to work reliably. Context switching was not implemented for this version of TTCS scheduler with Slack Stealing as this would mean implementing the code in assembly. This was beyond the scope of this project and because of time constraint, no work was to done to support slack tasks with task durations more than the tick interval.

Chapter 4

4. Implementation on the MCB900

The code was developed and tested on a MCB900 evaluation board with a Philips 89LPC932 microcontroller [ref. 18]. The MCB900 connects to the COM port of your PC via the serial COM interface. MCB900 includes the full suite of Keil µVision2 Development Studio which allows one to create and debug application programs. This includes a 4KB code size limited C51 Compiler (ANSI C compiler), a A51 Macro Assembler, a 4KB code size limited µVision2 Debugger with complete CPU and peripheral simulation and performance analyser which is a very useful tool that displays the execution time of functions. User applications were easily programmed onto the on-chip Flash ROM of the P89LPC932 device with a utility program called FlashMagic.

The Philips 89LPC932 microcontroller [ref .19] has an enhanced 80C51 core which runs at 6 times the speed of standard 80C51 devices. One machine cycle in 89LPC932 takes two CPU clock cycles (OSC_PER_INST) to run compared to the 6 or 12 cycles a typical 80C51 would require. It also has an internal RC oscillator that has a frequency of 7.373 MHz (OSC_FREQ). This means that an internal timer for example, Timer0, increments every 0.271µs and this is known as the timer resolution. To get a total measurement time of 1ms it has to count 3690 times until it overflows and causes an interrupt.

Chapter 5

5. Testing and Results

5.1 Test Results

Statistics are presented for comparison tests done between a TTCS scheduler and a TTCS scheduler with Slack stealing,

Memory

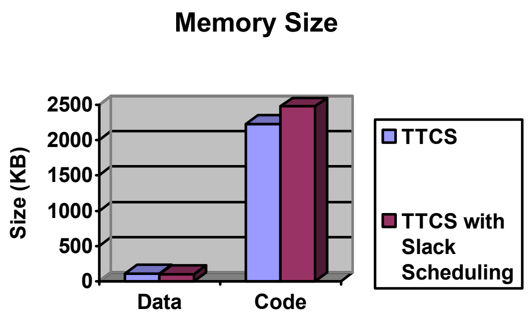

Figure 8 Comparison between a TTCS and TTCS with Slack Scheduling (Running Two tasks)

Figure 8 shows the memory usage of a purely TTCS scheduler and TTCS scheduler with Slack Stealing running two tasks (i.e running two tasks periodically in the first case and one task as a periodic task and one task as a Slack task in the latter case). The TTCS scheduler consumed 110.7 Kb of data memory while the TTCS scheduler with Slack Stealing consumed 98.7 Kb of data memory. Hence it may seem more economical to use TTCS scheduler with Slack Stealing when it comes to data memory but one has to keep in mind that every additional periodic task added to the scheduler took 7Kb of data memory whilst the addition of every slack task only used up 4Kb of data memory. That is why TTCS scheduler with Slack Stealing appears to be consuming less data memory.

But when one looks at the code memory usage it can be clearly seen that TTCS scheduler with Slack Stealing uses more code memory than a pure TTCS scheduler. In this case, the TTCS scheduler consumed 2233 Kb of data memory while the TTCS scheduler with Slack Stealing consumed 2479 Kb of data memory.

CPU loading

Next tests were carried out to find the CPU loading between a TTCS scheduler and TTCS scheduler with Slack Stealing. The TTCS scheduler and TTCS scheduler with Slack Stealing were running were two tasks (i.e running two tasks periodically in the first case and one task as a periodic task and one task as a Slack task in the latter case) as before. A function SCH_Go_To_Sleep( ) was written to allow the scheduler to enter idle mode when it has no ready tasks to execute. In the idle mode, the CPU is halted but the interrupts, timer and serial port functions continue operating. When an interrupt condition occurs, the scheduler comes out from the idle mode. Power consumption can be greatly decreased in idle mode. This piece of code forces the scheduler into idle mode.

```
void SCH_Go_To_Sleep()
{

  PCON |= 0x01;    // Enter idle mode (generic 8051 version)

}
```

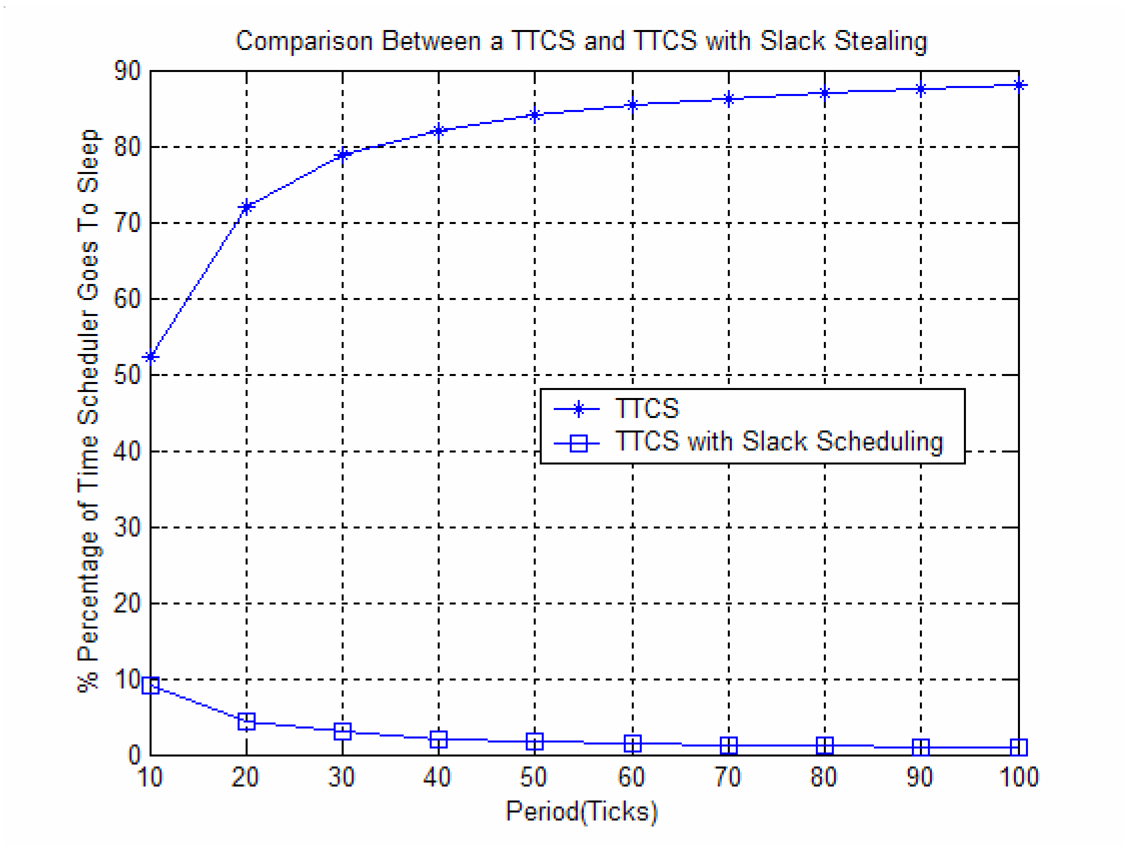

Figure 9 Percentage of time Scheduler Goes To Sleep

Measurements were done to show the proportion of time the scheduler enters idle mode for both cases as the period of the periodic tasks were increased and results are shown in Figure 9. From the results, it can be clearly seen the large difference in time that a TTCS scheduler sleeps and TTCS scheduler with Slack Stealing sleeps. For a large proportion of time, a TTCS scheduler remains idle when it has no ready tasks to execute. But this is not the case for a TTCS scheduler with Slack Stealing. In this type of scheduler, every spare processing time available between timer interrupts is stolen to execute some useful work and hence because of this TTCS scheduler with Slack Stealing sleeps very little. Also from the results shown it can be seen as the period of the periodic tasks increases, a pure TTCS scheduler tends to sleep more as seen by the percentage increase of 52% to 88% whilst the trend is reverse for a TTCS scheduler with Slack Stealing. This type of scheduler tends to sleep less as shown by the percentage decrease of 9% to 0.9%.

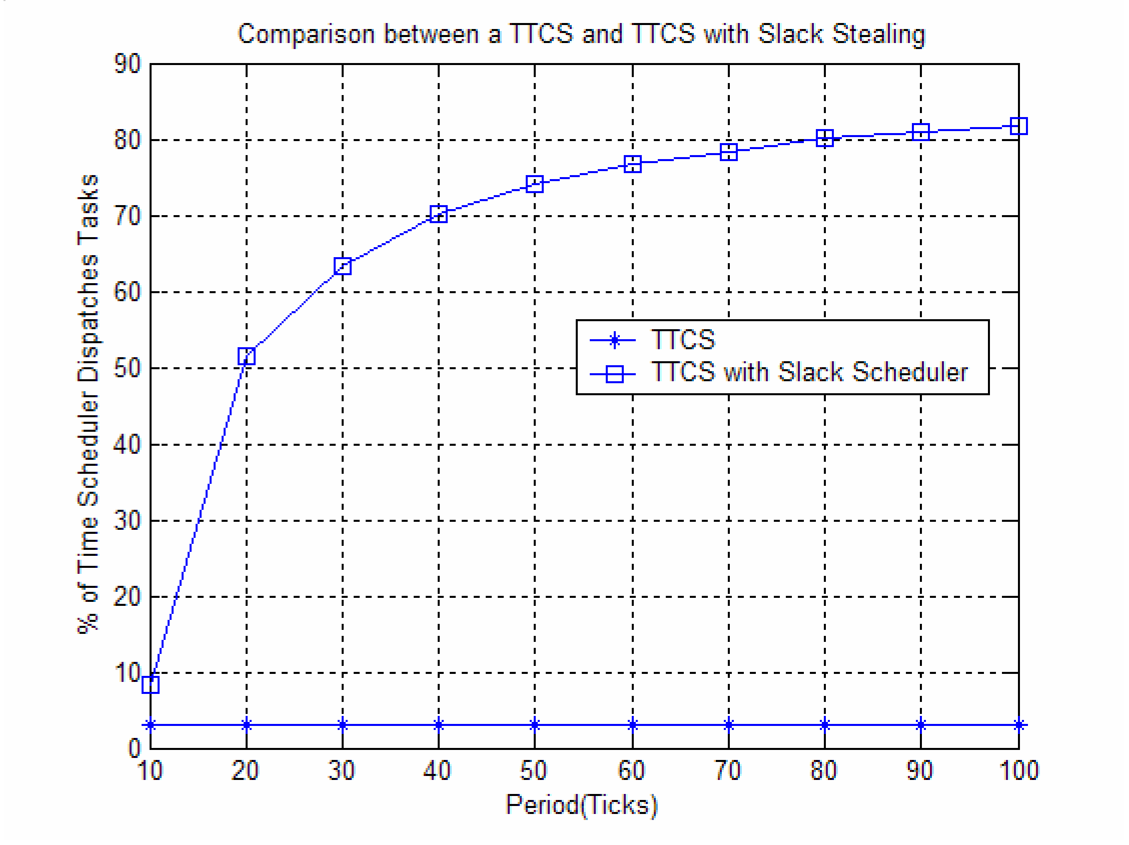

Figure 10 Percentage of time Scheduler Dispatches tasks

Next measurements were done to show the proportion of time the scheduler dispatches the tasks for the same test conditions. Figure 10 shows that for a TTCS scheduler the proportion of time the scheduler dispatches the tasks remains constant as the period of the periodic tasks increases but for the TTCS scheduler with Slack Stealing the proportion of time the scheduler dispatches the tasks increases with increasing period.

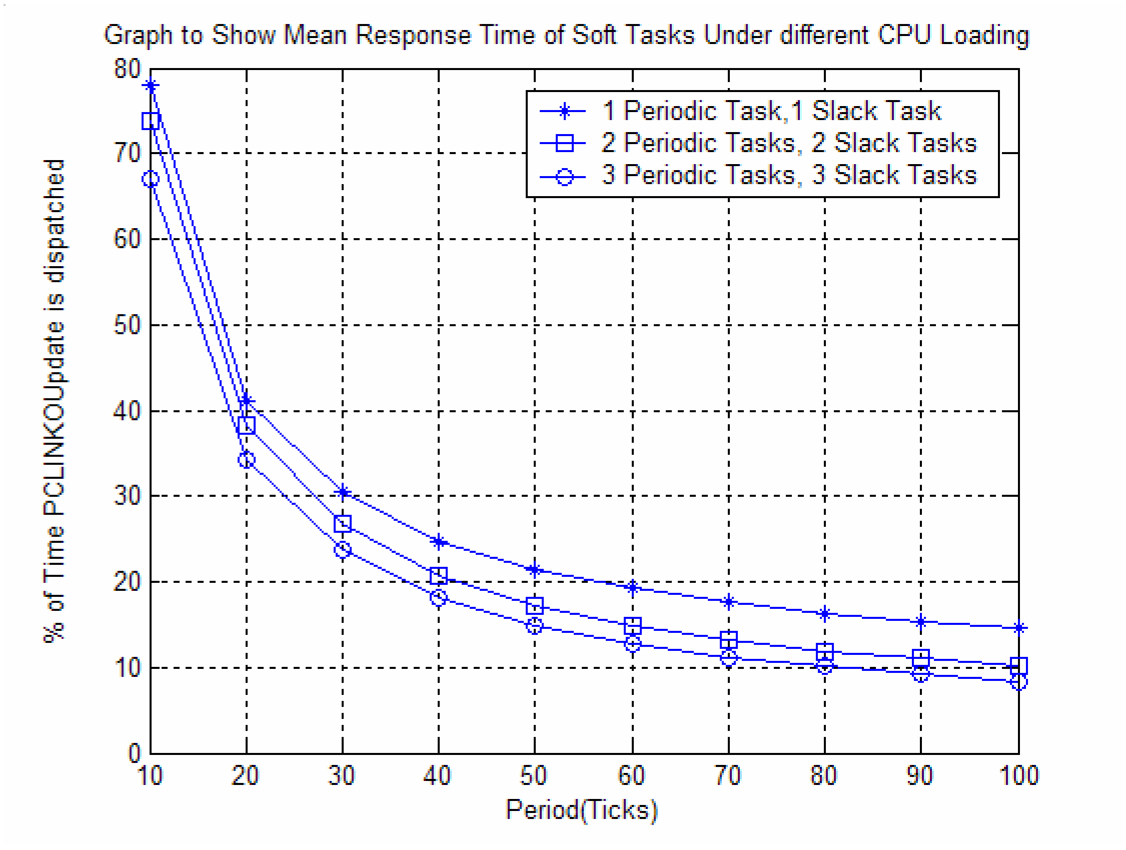

Figure 11 Mean Response Time of Soft Tasks under different CPU Loading

In Figure 11, results are presented for the percentage of time the slack task is executed under different CPU loading conditions. One particular slack task PC_LINK_O_Update was chosen as it is this slack task which sends out data out from the buffer to the RS232 link. It can be seen that as the period of the periodic tasks increases, the percentage of time this slack task is executed decreases but this percentage of time is less for higher CPU loading condition (scheduler executing 3 Periodic tasks and 3 slack tasks) than lower CPU loading condition (scheduler executing 1 Periodic task and 1 slack task).

Power Consumption

Power consumption is proportional to the time scheduler goes to sleep. Since a TTCS scheduler sleeps generally more than a TTCS scheduler with Slack Stealing, this implies that its power consumption is also less. The power consumption of a TTCS scheduler with Slack Stealing is generally many times more.

Testing for jitter with some RS232 processing

When a task is scheduled to execute at the 100th millisecond but if begins execution at the 102th millisecond, it said to be exhibiting jitter. Jitter was tested by setting suitable breakpoints at the point of entry of periodic tasks where execution begins. Here results are shown for Elapsed_Time_RS232_Update( ) periodic task with the period set at 1s.

TTCS Scheduler		**TTCS Scheduler with Slack Stealing**	
Time at breakpoint(s)	Period(s)	Time at breakpoint(s)	Period(s)
1.00878774	1.00718839	1.00881812	1.00725132
2.01597613	1.00718839	2.01606944	1.00725187
3.02316452	1.00718839	3.02332131	1.00725132
4.03035291	1.00718839	4.03057263	1.00725186
5.03754130		5.03782449	

Table 1 Testing for jitter

From the results shown in Table 1, it can be seen that the TTCS scheduler takes a constant period of 1.00718839s but the TTCS Scheduler with Slack Stealing also take takes a nearly similar period of 1.007s correct to 1/1000th of a second as that of a TTCS scheduler. Hence it can be concluded that the TTCS Scheduler with Slack Stealing exhibits no jitter.

Chapter 6

6. Applications

6.1 Applications in which slack stealing prove beneficial.

The choice of any appropriate scheduling algorithm depends strongly upon the nature of real-time application workload being considered. Some real time systems may have a mixture of hard deadline periodic tasks and soft deadline tasks. These soft deadline jobs like sending some data to and from the RS232 link, LCD applications, multimedia applications and batch processing jobs are normally run in the background. By running in the background, these soft jobs are only serviced when there are no ready periodic requests. On the contrary by using the TTCS scheduler with Slack Stealing the response time of these soft tasks is greatly improved. Also the slack scheduler can be invoked on demand. The TTCS scheduler with Slack Stealing is able to steal as much time from the periodic tasks while also guaranteeing that all the deadlines of the periodic tasks are met.

6.2 Applications in which slack stealing do not prove beneficial.

It is generally not recommended to use the TTCS scheduler with Slack Stealing in applications were timing considerations are very highly important. If even a delay of a fraction of a second can produce a very fatal result, the TTCS scheduler with Slack Stealing should not be used in such systems with such critical timing considerations.

Also since the TTCS scheduler with Slack Stealing generally consumes a lot more power than other schedulers, it is also not desirable to use this type of scheduler in systems where power is a scarce resource.

Chapter 7

7. Discussion

7.1 Technical Achievements

The major technical achievement of the project was the development and implementation of an efficient and optimal Slack Stealing algorithm for the TTCS scheduler developed by Pont [ref.1]. Every spare processing time between tick intervals was utilized to service soft tasks while previously the TTCS used to go to the idle mode when it had no periodic tasks to service. By testing for jitter, the TTCS scheduler with Slack Stealing was tested for reliability and predictability. An experimental analysis was also done for a TTCS scheduler with Slack Stealing and one with not and results presented in terms of memory, CPU loading and memory consumptions

7.2 Future Work

The TTCS scheduler with Slack Stealing can be further developed to handle context switching for slack tasks. As of now one of the restrictions of the scheduler developed is that that slack task duration should be less than the slack time available i.e $(Duration_{Slacktask} < \quad Tick\ Interval \quad)$. If long slack tasks greater than the tick interval are to serviced by this scheduler, then context switching becomes a necessary requirement. To do this the context switching functionality it has to be implemented in Assembly.

Another future direction where further work can be done is that currently the TTCS scheduler with Slack Stealing services the slack tasks in a round robin manner if there is slacktime available. A more prudent way of doing this could be by using the best fit algorithm strategy where the slack task which has the task duration that is the closest match to the available slack time is chosen and run first. By doing so, it can be ensured that every available slack time is used fruitfully to service some slack task and the scheduler sleeps the minimum possible.

Chapter 8

8. Conclusions

This project addressed the problem of jointly scheduling a set of periodic tasks with hard deadlines and tasks with soft deadlines in a Time Triggered Cooperative Scheduler (TTCS) through the concept of slack stealing. Often tasks may have an execution time less than the interrupt interval and the original TTCS scheduler developed by Pont [ref.1] used to go to idle mode when it had no periodic tasks to service. By implementing a dynamic Slack Stealing algorithm for a TTCS embedded system, slack tasks were serviced during this spare processing time. It was proven that the response time of task with soft deadlines like communications through the RS232 link can be minimized while also guaranteeing the deadlines of the periodic tasks. The performance of the proposed TTCS scheduler with Slack Stealing was also tested for jitter, memory usage, CPU loading and power consumptions and finally applications where the use of this slack scheduler proves beneficial compared to the usage of a purely cooperative scheduler were also presented.

References

[1] *Patterns for Time Triggered Embedded Systems*, Michael J.Pont, Addison Wesley, 2001.

[2] *Enhancing Response Times of End-to-End Tasks Using Slack of Local Tasks*, Namyun Kim, Taewoong Kim, Nachyuck Chang, Heonshik Shin, Department of Computer Engineering,Seoul National University

[3] *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Priority Preemptive Systems*. J. P. Lehoczky, S. Ramos-Thuel, In Proceedings Real-Time Systems Symposium, pp 11-123, Dec 1992.

[4] *On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems,*. S. R. Thuel, J.P. Lehoczky. In Proceedings 14th Real-Time Systems Symposium, pages 160-171, December 1993.

[5] *Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing,* S. R. Thuel ,J.P. Lehoczky. In Proceedings 15th Real-Time Systems Symposium,pages 22-33, San Juan, Puerto Rico, December 1994.

[6*] Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*, Sandra Ramos Thuel, Thesis, Carnegie Mellon University, May 1993.

[7] *Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed –Priority Preemptive System*s, Too Seng Tia, Jane W. S. Liu, Mallikarjun Shankar, University of Illinois at Urbana-Champaign.

[8] *Scheduling Slack Time in Fixed Priority Preemptive Systems*, R. I. Davis, K. W. Tindell, A. Burns, In Proceedings of Real-Time System Symposium, pp 223-231, 1993.

[9] *Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems*, R.I.Davis, Department of Computer Science, University of York, England

[10] *Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real Time Systems*, Too Seng Tia, University of Illinois at Urbana-Champaign.

[11] *Slack Stealing Job Admission Control*, Alia K, Atlas, Aze Bestavros, Computer Science Department, Boston University

[12] *An Efficient Scheme to Allocate Soft-Aperiodic Tasks in Multiprocessor Hard Real-Time Systems*, Josep M. Banus, Alex Arenas, University of Rovira i Virgili

[13] *The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments*, J. K. Strosnider, J. P. Lehoczky, and L. Sha, IEEE Transactions on Computers, pages 73 - 91, Volume 44, Number 1, January 1995.

[14] *Exploiting Unused Periodic Time for Aperiodic Service using the Extended Priority Exchange Algorithm*, B. Sprunt, J. Lehoczky, L. Sha, Proceedings of The IEEE Real-Time Systems Symposium, pages 251 - 258, December 1988.

[15] Scheduling Periodic and Aperiodic Tasks Using the Slack Stealing Algorithm, John. P. Lehoczcky, Sandra R. Thuel.

[16] The Non-Preemptive Scheduling of Periodic tasks Upon Multiprocessors, Sanjoy. K. Baruah, The University of North Carolina.

[17] Aperiodic Task Scheduling for Real-Time Systems, Brinkley Sprunt, PHD Dissertation, Carnegie Mellon University.

[18] MCB 900 User's Guide from the Keil's Development Tools for LPC900 CDROM.

[19] P89LPC932 User Manual.

Further Reading

The interested reader is recommended to the following literature which provide further information about the subject discussed.

[20] *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, C. L. Liu and J. W. Layland, Journal of ACM, Volume 20, Number 1, pages 46-61, 1973.

[21] *Hard Real-Time Scheduling: The Deadline Monotonic Approach*, N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, Proceedings of The 8th IEEE Workshop on Real-Time Operating Systems and Software, May 1991.

[22] *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*, N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings, Software Engineering Journal, Vol. 8, pages 284-292, September 1993,.

[23] *Implementation and Evaluation of a Time-Driven Scheduling Processor*, J. W. Wendorf, Proceedings of The IEEE Real-Time Systems Symposium, pages 172 -180, December 1988.

[24] *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks*, Kevin Jeffay, Donald .F. Stanat, Charles U. Martel, IEEE Computer Society pp 129-139, 1991.

[25] *On Non-Preemptive Scheduling of Recurring Tasks Using Inserted Idle Times*, Rodney . R. Howell, M. K. Venkatrao, Kansas State University, USA.

[26] *Online Handling of Firm Aperiodic Tasks in Time Triggered Systems*, Damir Isovic and Gerhard Fohler, Department of Computer Engineering, Malardalen University, Sweden.

[27] *Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*, Damir Isovi´c and Gerhard Fohler, Department of Computer Engineering Malardalen University, Sweden

[28] *Integrating Multimedia Applications in Hard Real-Time Systems*, Luca Abeni and Giorgio Buttazzo, Scuola Superiore S. Anna, Pisa

[29] *Scheduling Real-Time Applications in An Open Environment*, Z. Deng, J. W. S.Liu, Dept of Computer Science, University of Illinois at Urbana-Champaign.

[30] *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Kevin Jeffay, Department of Computer Science, University of North Carolina

[31] *Implications of Classical Scheduling Results for Real-Time Systems*, John A. Stankovic, Scuola Superiore S. Anna, Pisa, Italy

Appendix A

Source Code

The source code and all the supporting files have been included in the attached CD. The main code which implements the Slack Stealing algorithm is given here. Please refer to the CD for full details of all the supporting C and header files.

```
/*------------------------------------------------------------------------------------------------------------------*-
    *****************************************************************
              This code is submitted in partial fulfillment of the requirements
      for the degree of Master of Science in Informations & Communications Technology
                  in Dept of Engineering of University of Leicester,2003
              *************************************************



Project Title: Implementation of a Slack Stealing Algorithn for a TTCS Embedded System
Student Name : Suguna Thanagasundram
Project Supervisor: Michael.J.Pont
Date Created On: 1/07/2003
Last Date of Modification: 2/09/2003

-*------------------------------------------------------------------------------------------------------------------*/


/*------------------------------------------------------------------------------------------------------------------*-

  Main.c

 ------------------------------------------------------------------------------------------------------------------

  Test program for TTCS scheduler with Slack Stealing.

  TTCS scheduler was modified to incorporate Slack Stealing Algorithm.



  COPYRIGHT
  ---------

  The code for the TTCS scheduler is from the book:

  PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
  [Pearson Education, 2001; ISBN: 0-201-33138-1].

  The code for TTCS scheduler is copyright (c) 2001 by Michael J. Pont.

  See book for copyright details and other information.
```

```
-*-------------------------------------------------------------------------------------------------------------------*/

#include "Main.h"
#include "0_01_7g.h"
#include "PC_O_T1.h"
#include "Elap_232.h"
#include "Port.h"
#include "LED_Flash.h"


/* ............................................................................................................................................................ */
/* ............................................................................................................................................................ */

void main(void)
  {
  // Set up the scheduler(Enter appropriate tick interval eg 1ms tick interval)
  // Scheduler using Timer 0
  SCH_Init_T0(1);

  // Initialise Port 1 and Port 2 of MCB900 to display LEDS and RS232 output
  P2M2=0;
  P2M1=0;
  P1M1 = 0x00;
  P1M2 = 0x00;

  // Set baud rate to 9600
  // Use Timer 1 for baud rate generation
  PC_LINK_O_Init_T1(9600);

  // Prepare the elapsed time library
  Elapsed_Time_RS232_Init();

  // Prepare for dummy task
  LED_Flash_Init();

  // Add Periodic tasks (Higher Priority and hard deadlines) as appropriately here
  // TIMING IS IN TICKS

  // Update the time once per second as a periodic task
  SCH_Add_Task(Elapsed_Time_RS232_Update, 1000, 1000);

  // Commented tasks were used for testing purpose
  SCH_Add_Task(LED_Flash_Update, 0, 100);
  SCH_Add_Task(LED_Flash_Update1, 5, 100);
  //SCH_Add_Task(PC_LINK_O_Update,1,1);
  //SCH_Add_Task(LED_Flash_Update_Slack,2,100);
  //SCH_Add_Task(LED_Flash_Update_Slack1,3,100);

  // Add Slack tasks Lower Priority and soft deadlines) as appropriately here
  SCH_Add_Slack_Task(PC_LINK_O_Update);

  // Commented tasks were used for testing purpose
  SCH_Add_Slack_Task(LED_Flash_Update_Slack);
  SCH_Add_Slack_Task(LED_Flash_Update_Slack1);

  #ifdef SLACK_STEALING
```

```
   //Routine to find Slack tasks duration
   SCH_Slack_Tasks_Init();

   #endif

   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*------------------------------------------------------------------*-
  ---------------------------- END OF FILE ----------------------------
-*------------------------------------------------------------------*/
```

Listing 1 Main.c

```
/*------------------------------------------------------------------*-

   SCH51.C (v1.00)

  -------------------------------------------------------------------

   *** THESE ARE THE CORE SCHEDULER FUNCTIONS ***
   --- These functions may be used with all 8051 devices ---

   *** SCH_MAX_TASKS *must* be set by the user ***
   *** SCH_MAX_SLACK_TASKS *must* also be set by the user if slack
       scheduling is desired***
   --- see "Sch51.H" ---

   *** Includes (optional) power-saving mode ***
   --- You must ensure that the power-down mode is adapted ---
   --- to match your chosen device (or disabled altogether) ---

-*------------------------------------------------------------------*/

#include "Main.h"
#include "Port.h"
#include "Sch51.h"

// ---------------------- Public variable definitions ----------------------

// The array of tasks
```

```
sTask SCH_tasks_G[SCH_MAX_TASKS];
slackTask SCH_tasks_S[SCH_MAX_SLACK_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

// ------------------------------------------- Private function prototypes ---------------------------------------------

static void SCH_Go_To_Sleep(void);

// ------------------------------------------------ Private variables ----------------------------------------------------

// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
static tByte Last_error_code_G;


/*-------------------------------------------------------------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*-------------------------------------------------------------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;

   #ifdef SLACK_STEALING
   static tByte SlackIndex=0;
   tWord Slack_Counter=0;
   tWord Slack_Time_Remaining=0;
   tByte Periodic_Task_Ready_To_Run;
   #endif


   // Dispatches (runs) the next task (if one is ready)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      if (SCH_tasks_G[Index].RunMe > 0)
         {
         (*SCH_tasks_G[Index].pTask)();  // Run the task

         SCH_tasks_G[Index].RunMe -= 1;   // Reset / reduce RunMe flag

         // Periodic tasks will automatically run again
         // - if this is a 'one shot' task, remove it from the array
         if (SCH_tasks_G[Index].Period == 0)
            {
            SCH_Delete_Task(Index);
```

```
        }
      }
    }

  // Report system status
  SCH_Report_Status();

  /**************************************************************************************
  Slack Stealing modifications added by Suguna to dispatch slack tasks
  Date created : 12/07/2003
  Date modified: 10/08/2003
  *************************************************************************************/


  #ifdef SLACK_STEALING
  /*-----------------------------------------------------------------------------------------------------------------------*-


 This is part of the  'dispatcher' function which does the Slack scheduling
 Dertermination of the Slack time, decision to run to whether run the Slack task
 and decision to run which ready Slack task next were done dynamically.

 -*-----------------------------------------------------------------------------------------------------------------------*/

  // There are no periodic tasks to run
  Periodic_Task_Ready_To_Run=0;

  // Code to calculate slack time available
  Slack_Counter= TH0;
  Slack_Counter=(Slack_Counter<<8)+TL0;
  Slack_Time_Remaining = 65536- Slack_Counter;

  // Execute loop while there is no periodic task ready to run
  while( Periodic_Task_Ready_To_Run==0 )
  {
                    // Execute if task duration of current slack task is less than available slack time
                    if (SCH_tasks_S[SlackIndex].TaskDuration <= Slack_Time_Remaining)
                    {

                              // Run Slack task
                              (*SCH_tasks_S[SlackIndex].pTask)();

                              // Procede to next Slack task on array if one is ready
                              if (SlackIndex<SCH_MAX_SLACK_TASKS-1)
                              {
                                        SlackIndex++;
                              }
                              else
                              {
                                        SlackIndex=0;
                              }


                    }/*end of if loop*/
```

```
                /*Find if there are any periodic task to run in the next tick interval*/
                for (Index = 0; Index < SCH_MAX_TASKS; Index++)
                {
                        if (SCH_tasks_G[Index].Delay==0)
                        {
                        Periodic_Task_Ready_To_Run=1;
                        }
                }



                // Update slack time available again after ruuning one Slack task
                Slack_Counter= TH0;
                Slack_Counter=(Slack_Counter<<8)+TL0;
                Slack_Time_Remaining = 65536- Slack_Counter;


            // If there are no periodic tasks to run at the next tick interrupt and if Slack available is less
           //  the next ready Slack task duration, there is no point stopping the execution of the Slack tasks
           //  Add the Slack Task duration to the Slack time available and continue with the execution of the
           //  Slack task

                if (Periodic_Task_Ready_To_Run==0 && Slack_Time_Remaining <
                SCH_tasks_S[SlackIndex].TaskDuration)
                {
                Slack_Time_Remaining+=SCH_tasks_S[SlackIndex].TaskDuration;
                }




   }/*end of while loop*/

   #endif

   // Enter idle mode to save power
   SCH_Go_To_Sleep();

   }

/*------------------------------------------------------------------------------------------------------------------------*-

  SCH_Add_Task()

  Causes a task (function) to be executed at regular intervals
  or after a user-defined delay

  Fn_P   - The name of the function which is to be scheduled.
         NOTE: All scheduled functions must be 'void, void' -
         that is, they must take no parameters, and have
         a void return type.

  DELAY  - The interval (TICKS) before the task is first executed
  PERIOD - If 'PERIOD' is 0, the function is only called once,
         at the time determined by 'DELAY'.  If PERIOD is non-zero,
         then the function is called repeatedly at an interval
         determined by the value of PERIOD (see below for examples
         which should help clarify this).


  RETURN VALUE:

  Returns the position in the task array at which the task has been
  added.  If the return value is SCH_MAX_TASKS then the task could
  not be added to the array (there was insufficient space).  If the
  return value is < SCH_MAX_TASKS, then the task was added
  successfully.

  Note: this return value may be required, if a task is
  to be subsequently deleted - see SCH_Delete_Task().

  EXAMPLES:

  Task_ID = SCH_Add_Task(Do_X,1000,0);
  Causes the function Do_X() to be executed once after 1000 sch ticks.

  Task_ID = SCH_Add_Task(Do_X,0,1000);
  Causes the function Do_X() to be executed regularly, every 1000 sch ticks.

  Task_ID = SCH_Add_Task(Do_X,300,1000);
  Causes the function Do_X() to be executed regularly, every 1000 ticks.
  Task will be first executed at T = 300 ticks, then 1300, 2300, etc.

-*------------------------------------------------------------------------------------------------------------------------------*/
tByte SCH_Add_Task(void (code * pFunction)(),
                   const tWord DELAY,
                   const tWord PERIOD)
   {
   tByte Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

      // Also return an error code
      return SCH_MAX_TASKS;
      }

   // If we're here, there is a space in the task array
  SCH_tasks_G[Index].pTask  = pFunction;

  SCH_tasks_G[Index].Delay  = DELAY;
  SCH_tasks_G[Index].Period = PERIOD;

  SCH_tasks_G[Index].RunMe  = 0;

  return Index; // return position of task (to allow later deletion)
  }




/*------------------------------------------------------------------------------------------------------------------------*-

***************************************************************************************
SCH_Add_Slack_Task()function added by Suguna to add slack tasks
Date created : 20/07/2003
Date modified: 28/07/2003
***************************************************************************************

  SCH_Add_Slack_Task()

  Causes a Slack task (function) to be added to the Slack task list.
  Slack tasks are executed if there is available Slack time.

  Fn_P   - The name of the Slack function which is to be scheduled.
          NOTE: All scheduled functions must be 'void, void' -
          that is, they must take no parameters, and have
          a void return type.

  RETURN VALUE:

  Returns the position in the Slack task array at which the Slack task
  has been added.  If the return value is SCH_MAX_SLACK_TASKS then the
  task could not be added to the array (there was insufficient space).
  If the return value is < SCH_MAX_SLACK_TASKS,then the task was added
  successfully.

  Note: this return value may be required, if a task is
  to be subsequently deleted - see SCH_Delete_Slack_Task().

  EXAMPLES:

  Task_ID = SCH_Add_Slack_Task(Do_X);
  Causes the function Do_X() to be executed if there is available Slack time.

-*------------------------------------------------------------------------------------------------------------------------*/


tByte SCH_Add_Slack_Task(void (code * pFunction)())
{
          tByte SlackIndex = 0;

  // First find a gap in the array (if there is one)
  while ((SCH_tasks_S[SlackIndex].pTask != 0) && (SlackIndex < SCH_MAX_SLACK_TASKS))
```

```
      {
      SlackIndex++;
      }

   // Have we reached the end of the list?
   if (SlackIndex == SCH_MAX_SLACK_TASKS)
      {
      // Task list is full
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

      // Also return an error code
      return SCH_MAX_SLACK_TASKS;
      }

   // If we're here, there is a space in the task array
   SCH_tasks_S[SlackIndex].pTask  = pFunction;

   SCH_tasks_S[SlackIndex].TaskDuration  = 0x00;

   return SlackIndex; // return position of task (to allow later deletion)*/
}

/*------------------------------------------------------------------*/



/*------------------------------------------------------------------*-

  SCH_Delete_Task()

  Removes a task from the scheduler.  Note that this does
  *not* delete the associated function from memory:
  it simply means that it is no longer called by the scheduler.

  TASK_INDEX - The task index.  Provided by SCH_Add_Task().

  RETURN VALUE:  RETURN_ERROR or RETURN_NORMAL

-*------------------------------------------------------------------*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
   {
   bit Return_code;

   if (SCH_tasks_G[TASK_INDEX].pTask == 0)
      {
      // No task at this location...
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

      // ...also return an error code
      Return_code = RETURN_ERROR;
      }
   else
```

```
    {
    Return_code = RETURN_NORMAL;
    }

  SCH_tasks_G[TASK_INDEX].pTask   = 0x0000;
  SCH_tasks_G[TASK_INDEX].Delay   = 0;
  SCH_tasks_G[TASK_INDEX].Period  = 0;

  SCH_tasks_G[TASK_INDEX].RunMe   = 0;

  return Return_code;       // return status
  }



/*------------------------------------------------------------------*-

****************************************************************************
SCH_Delete_Slack_Task()function added by Suguna to add slack tasks
Date created : 25/07/2003
Date modified: 28/07/2003
****************************************************************************


  SCH_Delete_Slack_Task()

  Removes a Slack task from the scheduler.  Note that this does
  *not* delete the associated function from memory:
  it simply means that it is no longer called by the scheduler.

  TASK_INDEX - The task index.  Provided by SCH_Add_Slack_Task().

  RETURN VALUE:  RETURN_ERROR or RETURN_NORMAL

-*------------------------------------------------------------------*/
bit SCH_Delete_Slack_Task(const tByte SLACK_TASK_INDEX)
  {
  bit Return_code;

  if (SCH_tasks_S[SLACK_TASK_INDEX].pTask == 0)
    {
    // No task at this location...
    //
    // Set the global error variable
    Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

    // ...also return an error code
    Return_code = RETURN_ERROR;
    }
  else
    {
    Return_code = RETURN_NORMAL;
    }

  SCH_tasks_S[SLACK_TASK_INDEX].pTask   = 0x0000;
```

```
   SCH_tasks_S[SLACK_TASK_INDEX].TaskDuration   = 0;

   return Return_code;       // return status
   }

/*------------------------------------------------------------------------------------------------------------------------*-

  SCH_Report_Status()

  Simple function to display error codes.

  This version displays code on a port with attached LEDs:
  adapt, if required, to report errors over serial link, etc.

  Errors are only displayed for a limited period
  (60000 ticks = 1 minute at 1ms tick interval).
  After this the the error code is reset to 0.

  This code may be easily adapted to display the last
  error 'for ever': this may be appropriate in your
  application.

  See Chapter 10 for further information.

-*------------------------------------------------------------------------------------------------------------------------*/
void SCH_Report_Status(void)
   {
#ifdef SCH_REPORT_ERRORS
   // ONLY APPLIES IF WE ARE REPORTING ERRORS
   // Check for a new error code
   if (Error_code_G != Last_error_code_G)
      {
      // Negative logic on LEDs assumed
      Error_port = 255 - Error_code_G;

      Last_error_code_G = Error_code_G;

      if (Error_code_G != 0)
         {
         Error_tick_count_G = 60000;
         }
      else
         {
         Error_tick_count_G = 0;
         }
      }
   else
      {
      if (Error_tick_count_G != 0)
         {
         if (--Error_tick_count_G == 0)
            {
            Error_code_G = 0; // Reset error code
            }
         }
      }
#endif
   }


/*---------------------------------------------------------------------------------------------------------------------------*-

  SCH_Go_To_Sleep()

  This scheduler enters 'idle mode' between clock ticks
  to save power.  The next clock tick will return the processor
  to the normal operating state.

  Note: a slight performance improvement is possible if this
  function is implemented as a macro, or if the code here is simply
  pasted into the 'dispatch' function.

  However, by making this a function call, it becomes easier
  - during development - to assess the performance of the
  scheduler, using the 'performance analyser' in the Keil
  hardware simulator. See Chapter 14 for examples for this.

  *** May wish to disable this if using a watchdog ***

  *** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

-*---------------------------------------------------------------------------------------------------------------------------*/
void SCH_Go_To_Sleep()
   {
   PCON |= 0x01;    // Enter idle mode (generic 8051 version)

   // Entering idle mode requires TWO consecutive instructions
   // on 80c515 / 80c505 - to avoid accidental triggering
   //PCON |= 0x01;    // Enter idle mode (#1)
   //PCON |= 0x20;    // Enter idle mode (#2)
   }




/*---------------------------------------------------------------------------------------------------------------------------*-

*****************************************************************************************
SCH_Slack_Tasks_Init()function added by Suguna to find slack tasks duration
Date created : 23/08/2003
Date modified: 28/08/2003
*****************************************************************************************

  SCH_Slack_Tasks_Init()

  This function finds the Slack tasks duration in terms of "TICK"s
  despite the oscillator speed,cycles per instruction and tick
  interval size.
  Must be called after the Slack tasks have been added in the main
  function.
```

```
-*----------------------------------------------------------------------------------------------------------------------*/
void SCH_Slack_Tasks_Init(void)
{
        tByte i;
        tWord Slack_Counter1=0;
        tWord Slack_Counter2=0;


        for(i=0;i<SCH_MAX_SLACK_TASKS;i++)
        {
        Slack_Counter1= TH0;
        Slack_Counter1=(Slack_Counter1<<8)+TL0;
        (*SCH_tasks_S[i].pTask)();
        Slack_Counter2= TH0;
        Slack_Counter2=(Slack_Counter2<<8)+TL0;
        SCH_tasks_S[i].TaskDuration=Slack_Counter2-Slack_Counter1;
        }

}

/*----------------------------------------------------------------------------------------------------------------------*-
  ------------------------------------------------ END OF FILE --------------------------------------------------------
-*----------------------------------------------------------------------------------------------------------------------*/
```

Listing 2 Sch51.c

```
/*----------------------------------------------------------------------------------------------------------------------*-

   0_01_7g.C (v1.00)

  ------------------------------------------------------------------------------------------------------------------------

   *** THIS IS A SCHEDULER FOR STANDARD 8051 / 8052 ***

   *** Uses T0 for timing, 16-bit manual reload ***
   *** 7.373 MHz oscillator -> 1 ms (precise) tick interval ***

   ***********************************************************************************
   This code was modified to use Timer 0 with 16-bit manual reloading
   and variable tick rate Setting  by Suguna
   Date created : 23/08/2003
   Date modified: 28/08/2003
   ***********************************************************************************


-*----------------------------------------------------------------------------------------------------------------------*/

#include "0_01_7g.h"
```

```
// ------ Private variable definitions ----------------------------
static tByte Reload_08H;
static tByte Reload_08L;

static tByte Tick_ms_G;
static tWord Millisecond_count = 0;


// ------------------------------------------ Public variable declarations ---------------------------------------------

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];
extern slackTask SCH_tasks_S[SCH_MAX_SLACK_TASKS];


// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
extern tByte Error_code_G;




/*-------------------------------------------------------------------------------------------------------------------------*-

  SCH_Manual_Timer0_Reload()

  This scheduler uses a (manually reloaded) 16-bit timer.
  The manual reload means that all timings are approximate.
  THIS SCHEDULER IS NOT SUITABLE FOR APPLICATIONS WHERE
  ACCURATE TIMING IS REQUIRED!!!
  Timer reload is carried out in this function.

-*-------------------------------------------------------------------------------------------------------------------------*/
static void SCH_Manual_Timer0_Reload()
   {
   // Stop Timer 0
   TR0 = 0;

   // Philips 89LPC932, 7.373 MHz
   // The Timer 0 resolution is 0.271 µs
   // The required Timer 0 overflow is 0.001 seconds (1 ms)
   // -> we are generating timer ticks at ~1 ms intervals


   // See 'init' function for calculations
   TL0  = Reload_08L;
   TH0  = Reload_08H;

   //  Start Timer 0
   TR0  = 1;
   }
```

```
/*------------------------------------------------------------------*-

  SCH_Init_T0()

  Scheduler initialisation function.  Prepares scheduler
  data structures and sets up timer interrupts at required rate.
  Must call this function before using the scheduler.

-*------------------------------------------------------------------*/
void SCH_Init_T0(const tByte TICK_MS)
   {
   tByte i;
   tLong Inc;
   tWord Reload_16;

   // Store the tick data
   Tick_ms_G = TICK_MS;


   for (i = 0; i < SCH_MAX_TASKS; i++)
      {
      SCH_Delete_Task(i);
      }

    for (i = 0; i < SCH_MAX_SLACK_TASKS; i++)
      {
      SCH_Delete_Slack_Task(i);
      }

   // Reset the global error variable
   // - SCH_Delete_Task() will generate an error code,
   //   (because the task array is empty)
   Error_code_G = 0;


   // Using Timer 0, 16-bit *** manual reload ***
   TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
   TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

   // Number of timer increments required (max 65536)
   Inc = ((tLong)TICK_MS * (OSC_FREQ/1000)) / (tLong)OSC_PER_INST;

   // 16-bit reload value
   Reload_16 = (tWord) (65536UL - Inc);

   // 8-bit reload values (High & Low)
   Reload_08H = (tByte)(Reload_16 / 256);
   Reload_08L = (tByte)(Reload_16 % 256);

   TL0  = Reload_08L;
   TH0  = Reload_08H;

   // Timer 0 interrupt is enabled, and ISR will be called
   // whenever the timer overflows.
   ET0 = 1;
   // Start Timer 0 running
   TR0 = 1;

   // Globally enable interrupts
   EA = 1;
   }


/*------------------------------------------------------------------*-

  SCH_Start()

  Starts the scheduler, by enabling interrupts.

  NOTE: Usually called after all regular tasks are added,
  to keep the tasks synchronised.

  NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

-*------------------------------------------------------------------*/
void SCH_Start(void)
   {
   EA = 1;
   }

/*------------------------------------------------------------------*-

  SCH_Update()

  This is the scheduler ISR.  It is called at a rate
  determined by the timer settings in the 'init' function.

  This version is triggered by Timer 2 interrupts:
  timer is automatically reloaded.

-*------------------------------------------------------------------*/
void SCH_Update(void) interrupt INTERRUPT_Timer_0_Overflow
   {
   tByte Index;

   // Reload the timer
   SCH_Manual_Timer0_Reload();


   // NOTE: calculations are in *TICKS* (not milliseconds)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      // Check if there is a task at this location
      if (SCH_tasks_G[Index].pTask)
         {
         if (SCH_tasks_G[Index].Delay == 0)
            {
            // The task is due to run
            SCH_tasks_G[Index].RunMe += 1;  // Inc. the 'RunMe' flag

            if (SCH_tasks_G[Index].Period)
            {
            // Schedule periodic tasks to run again
            SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period-1;
            }
          }
        else
          {
          // Not yet ready to run: just decrement the delay
          SCH_tasks_G[Index].Delay -= 1;
          }
        }
      }
    }


/*------------------------------------------------------------------------------------------------------------------------*-
  ---- ---------------------------------------------END OF FILE --------------------------------------------------------
-*------------------------------------------------------------------------------------------------------------------------*/
```

Listing 3 0_01_7g.C

Appendix B

Datasheet 89LPC932

The data sheet for the Philips 89LPC932 is given in the accompanying CD for reader’s reference.

Appendix C

Financial Statement

Description of the expenditure incurred in project	Duration	Unit	No of units	Unit Rate(£)	Total amount
Personnel					
Long-term					
Short-term					
Activities					
Activity 1					
Activity 2					
Activity 3					
Publications and reports					
CD-R		1	1	0.50	50p
Paper		1 rim	1	2	2.00
Ink Cartrige		Cartridge	2	20	40
Printing Cost in Library				5	5
Equipment					
MCB900 Board		On Loan from Embedded Lab	0	0	0
USB to Serial Cable		1 cable	15	15	15
Total expenditure					
					62.5

Appendix D

Proposal