DESIGN AND IMPLEMENTATION OF DYNAMIC SCHEDULER ON REAL TIME OS

Sanjana V Kalyanappagol

Shreyas Vasanthkumar

Final Project Report
ECEN 5613 Embedded System Design
May 06, 2017

INDEX

List of Figures

List of Tables

1 INTRODUCTION

A clear majority of all microprocessors manufactured today are used in embedded systems. These systems are distinguished by their interaction with unique hardware, concurrent measurement and control of analog environmental variables, predictable response times, reliability and safety. Real-time computing necessitates that the results of a computation must satisfy timing constraints and logical correctness. This is governed by laws of pessimism such as Murphy's Law which states, "If something can go wrong, it will go wrong" [1]. This can prove disastrous when systems are life-critical such as missile detection, flight control, medical systems and so on. A dynamic scheduler that considers the Worst-Case Execution Time(WCET) of the tasks to ensure adherence to timing constraints is therefore necessary for such systems. This project aims to design and implement a scheduler that can provide confirmation of schedule-ability of the given set of tasks prior to execution and increase processor utilization with a reduction in the number of deadline misses.

1.1 System Overview

The system basically consists of a microcontroller which runs three different types of tasks namely aperiodic, periodic and sporadic task. The scheduling of these tasks is done based on dynamic scheduling algorithms, EDF (Earliest Deadline First) and LST (Least Slack Time) respectively. The scheduler assigns priority to these tasks at every tick count based on the type of algorithm used for scheduling. Totally four tasks are run on the system where the first two make use of the GPIO resource as the mutex. A seven-segment LED display is interfaced with the controller to indicate the task number which is currently running. An external push button switch along with two on-board switches are used to indicate an aperiodic task to be executed. An SD card is also interfaced to the controller and data is written to the SD card whenever the second task is scheduled to be run. A serial terminal is used to display the time at which each task started and finished its execution.

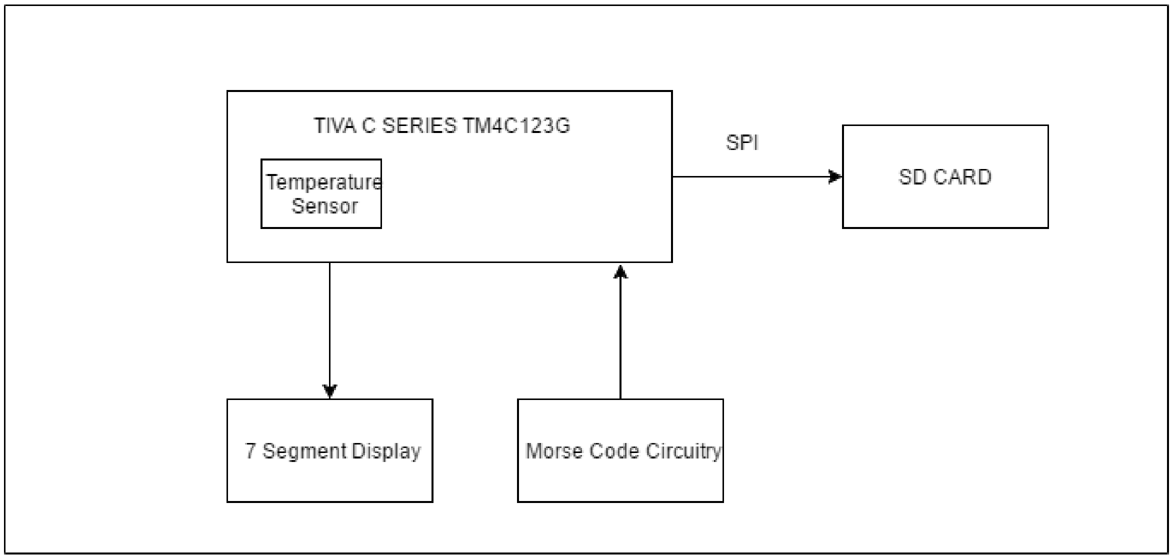


Figure 1 Hardware Block Diagram

2 TECHNICAL DESCRIPTION

The following sections detail the design and implementation of the dynamic scheduler on a Real-time OS.

2.1 Board Design

The scheduler is implemented using the EDF/LST scheduling algorithm, by creating two periodic tasks, one aperiodic task and once sporadic task which are initialized at the beginning. This scheduler is simulated in the Code Composer Studio (CCS 7.1.0). Once the result is obtained the RTOS is ported onto ARM board Tiva C Series EK-TM4C123GXL (hence forth referred as the MCU) and the scheduler is checked for desired operation. The result i.e. status of the tasks is observed on the serial monitor and the seven-segment display. A secure digital card (SD) card is interfaced with the MCU which operates on the SPI protocol. The SD card is accessed whenever the second task is run by the scheduler. Hence every time the second task is run a message that the second task was scheduled is written to the card.

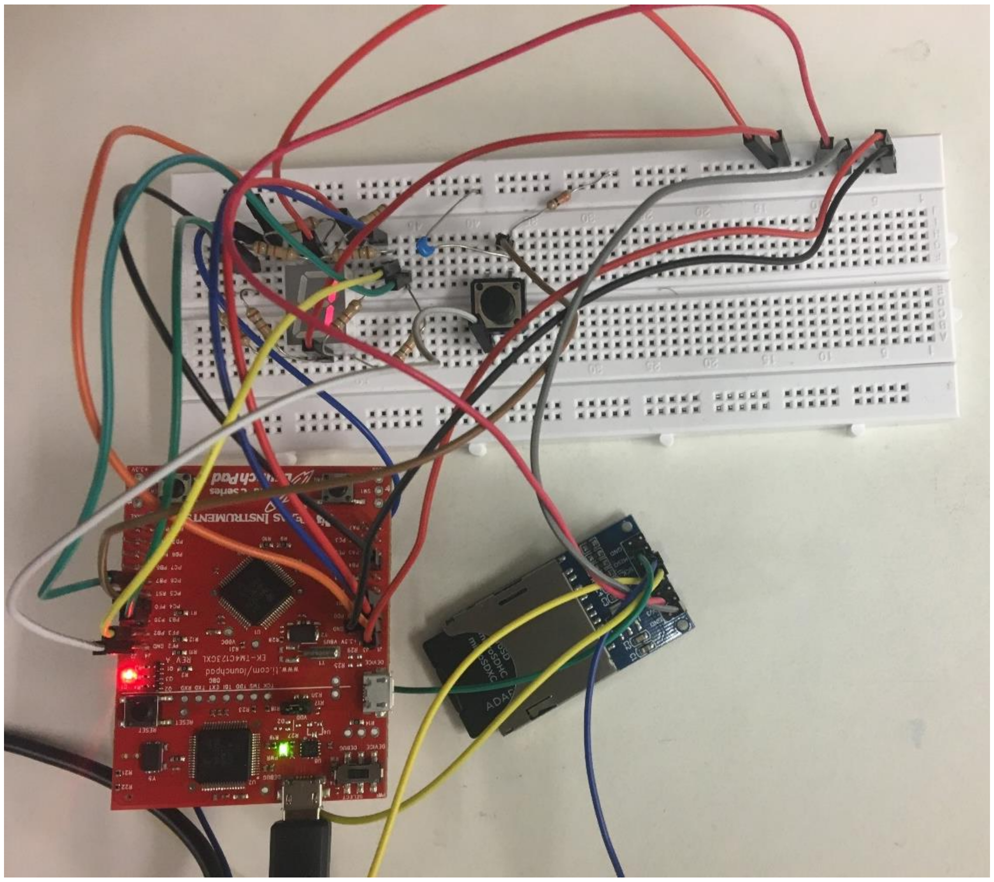


Figure 2 Final Board Setup

2.2 Scheduling Algorithms Design

The scheduler plays the most important role in the execution of the program. It decides which tasks to run. It also determines whether a task is ready to be executed, if it has finished executing and take necessary steps for these conditions.

Once a task is created it is put into the ready queue. Then it is sent to the scheduler where depending upon the scheduling algorithm priorities are assigned and put into the execution stage where the task is run by the processor. Once a task finishes its execution it is put into a waitlist until it is woken up again. A task gets pre-empted and put back into the ready queue while in the execution stage if a higher priority task needs to be serviced.

The scheduling algorithms used in this application can be categorized into static and dynamic scheduling algorithms.

In static scheduling algorithm, the same priority is assigned to all the tasks. Dynamic scheduling algorithms assign priorities to tasks in the execution stage. There are two dynamic scheduling algorithms implemented in this application. They are Earliest Deadline First (EDF) and Least Slack Time (LST) respectively.

EDF takes into account the absolute deadline of the task. Hence tasks with absolute deadline which are earlier will be assigned a higher priority. EDF is typically executed in preemptive mode, thus the currently executing task is preempted whenever another periodic instance with earlier deadline becomes active. Since EDF does not take into account the periodicity of tasks, it can be used to schedule aperiodic tasks as well.

LST considers the slack time of each task before assigning priority. Slack time for a task at the current instant can be defined as the difference between the deadline time and remaining execution time minus the current instant of time. LST can be used to schedule all tasks as mentioned by the previous statement. LST is the optimal algorithm for a single processor.

The schedule-ability of a periodic task set handled by EDF and LST can be verified through the processor utilization factor.

“A set of periodic tasks is schedulable with EDF and LST if and only if

Where Ci=Computation time,

Ti=Time period

$$\sum_{i=0}^{n} \frac{Ci}{Ti} \leq 1$$

n=number of tasks

Where C/T is called the Utilization ratio for a task” [4]

2.3 Hardware Description

2.3.1 Tiva TM4C123G Launchpad

The Tiva™ C Series TM4C123G LaunchPad Evaluation Board (EK-TM4C123GXL) is a low-cost evaluation platform for ARM® Cortex™-M4F-based microcontrollers. The Tiva C Series LaunchPad design highlights the TM4C123GH6PMI microcontroller USB 2.0 device interface, hibernation module, and motion control pulse-width modulator (MC PWM) module. The Tiva C Series LaunchPad also features programmable user buttons and an RGB LED for custom applications. The stackable headers of the Tiva C Series TM4C123G LaunchPad BoosterPack XL interface demonstrate how easy it is to expand the functionality of the Tiva C Series LaunchPad when interfacing to other peripherals on many existing BoosterPack add-on boards as well as future products. The below figure shows a photo of the Tiva C Series LaunchPad.

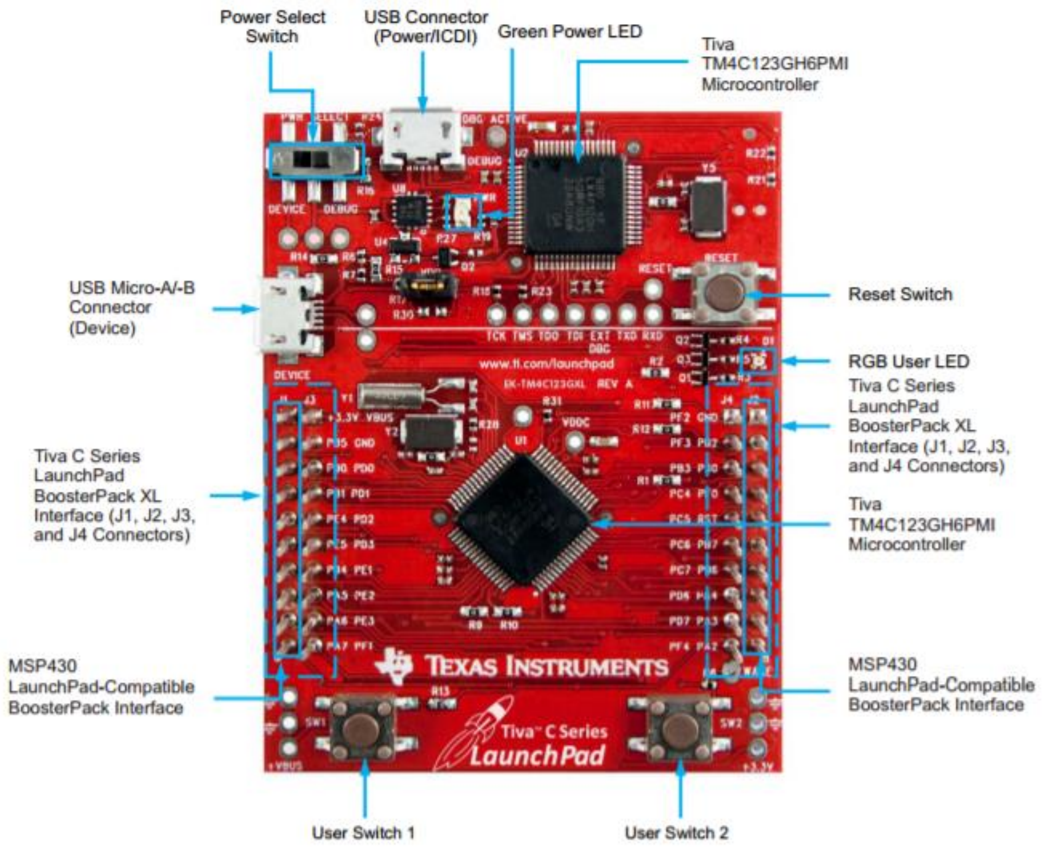

Power Select Switch
USB Connector (Power/ICDI)
Green Power LED
Tiva TM4C123GH6PMI Microcontroller
USB Micro-A/-B Connector (Device)
Reset Switch
RGB User LED
Tiva C Series LaunchPad BoosterPack XL Interface (J1, J2, J3, and J4 Connectors)
Tiva C Series LaunchPad BoosterPack XL Interface (J1, J2, J3, and J4 Connectors)
Tiva TM4C123GH6PMI Microcontroller
MSP430 LaunchPad-Compatible BoosterPack Interface
MSP430 LaunchPad-Compatible BoosterPack Interface
User Switch 1
User Switch 2

Figure 3 TM4C1234G Launchpad[17]

The main features of the Tiva C Series Launchpad are as follows:

• Tiva TM4C123GH6PMI microcontroller.

• Motion control PWM.

• USB micro-A and micro-B connector for USB device, host, and on-the-go (OTG) connectivity.

• RGB user LED.

• Two user switches (application/wake).

• Available I/O brought out to headers on a 0.1-in (2.54-mm) grid.

• On-board ICDI.

• Switch-selectable power sources:

– ICDI

– USB device

• Reset switch.

• Preloaded RGB quickstart application.

• Supported by TivaWare for C Series software including the USB library and the peripheral driver library.

• Tiva C Series TM4C123G LaunchPad BoosterPack XL Interface, which features stackable headers to expand the capabilities of the Tiva C Series LaunchPad development platform.

2.3.2 Seven Segment Display(SSD)

The 14.2 mm (0.56 inch) LED seven segment displays are designed for viewing distances up to 7 meters (23 feet). These devices use an industry standard size package and pinout. Both the numeric and ±1 overflow devices feature a right hand decimal point. All devices are available as either common anode or common cathode. These displays are ideal for most applications. Pin for pin equivalent displays are also available in a low current design. The low current displays are ideal for portable applications. SSD comes both in common anode and common cathode form. The one which has been used for interfacing purpose is common anode type.

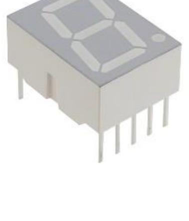


Figure 4 Seven Segment Display[12]

2.3.3 Push Button

A push-button (also spelled pushbutton) or simply button is a simple switch mechanism for controlling some aspect of a machine or a process. Buttons are typically made out of hard material, usually plastic or metal. The surface is usually flat or shaped to accommodate the human finger or hand, so as to be easily depressed or pushed. Buttons are most often biased switches, although many un-biased buttons (due to their physical nature) still require a spring to return to their un-pushed state. It is used along with the on-board switch for Morse code application.

Figure 5 Pushbutton[14]

2.3.4 Breadboard

A breadboard is a construction base for prototyping of electronics. Originally it was literally a bread board, a polished piece of wood used for slicing bread. The type of breadboard used here is a solderless breadboard. Because the solderless breadboard does not require soldering, it is reusable. This makes it easy to use for creating temporary prototypes and experimenting with circuit design. It is used as a base for interfacing seven segment display and push button.

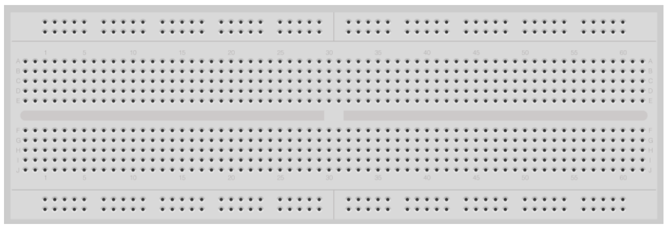


Figure 6 Breadboard[16]

2.3.5 SD Card Reader Module

The Arduino SD Card Shield is a simple SD card reader that can be used for transferring data to and from a standard SD card. It works based on the SPI protocol. It allows you to add mass storage and perform data logging. The SD card module was used in our project to write data whenever the scheduler ran task 2.

Figure 7 SD Card Module[15]

2.4 Firmware Design

The firmware design of this project included the drivers for the UART and SPI interfaces, reading the hardware ADC and to transfer data between the MCU and the SD card. The firmware was developed with the Code Composer Studio IDE (CCS 7.1.0) in C for the Tiva TM4C123G launchpad.

2.4.1 UART Driver

The UART driver is used to communicate over the USB port to the computer. In this project, it is basically used to print messages and data about the execution times of tasks on the screen. Any serial terminal can be used to print the UART statements on the screen. TI’s energia serial terminal has been used in our application. Any other serial terminal such as Tera term, Real Term can also be used for this purpose.

The data rate is set for 8 bits with 1 bit stop bit. The UART is set to double speed and the baud rate is set to 115200 for the fastest possible low error rate transmission.

2.4.2 SPI Driver

The SPI driver also known as SSI driver available from Free RTOS was used to design our application specific library function. This library function was basically used to interface the MCU and the SD card reader module. The MCU was configured as the master and the SD card reader was configured as the slave. Before data can be written into or read from the SD card it needs to be initialized with a set of commands. During the initialization process of the SD card it was necessary to use a baud rate between 100 and 400 KHz. After every command the SD card sends back valid responses which verifies whether the SD card accepted the command or not. Each of these commands has an index number, argument, crc (control redundancy check) bits along with the data to be sent. The sequence of commands to be sent can be summarized in the form of a flowchart as shown below.

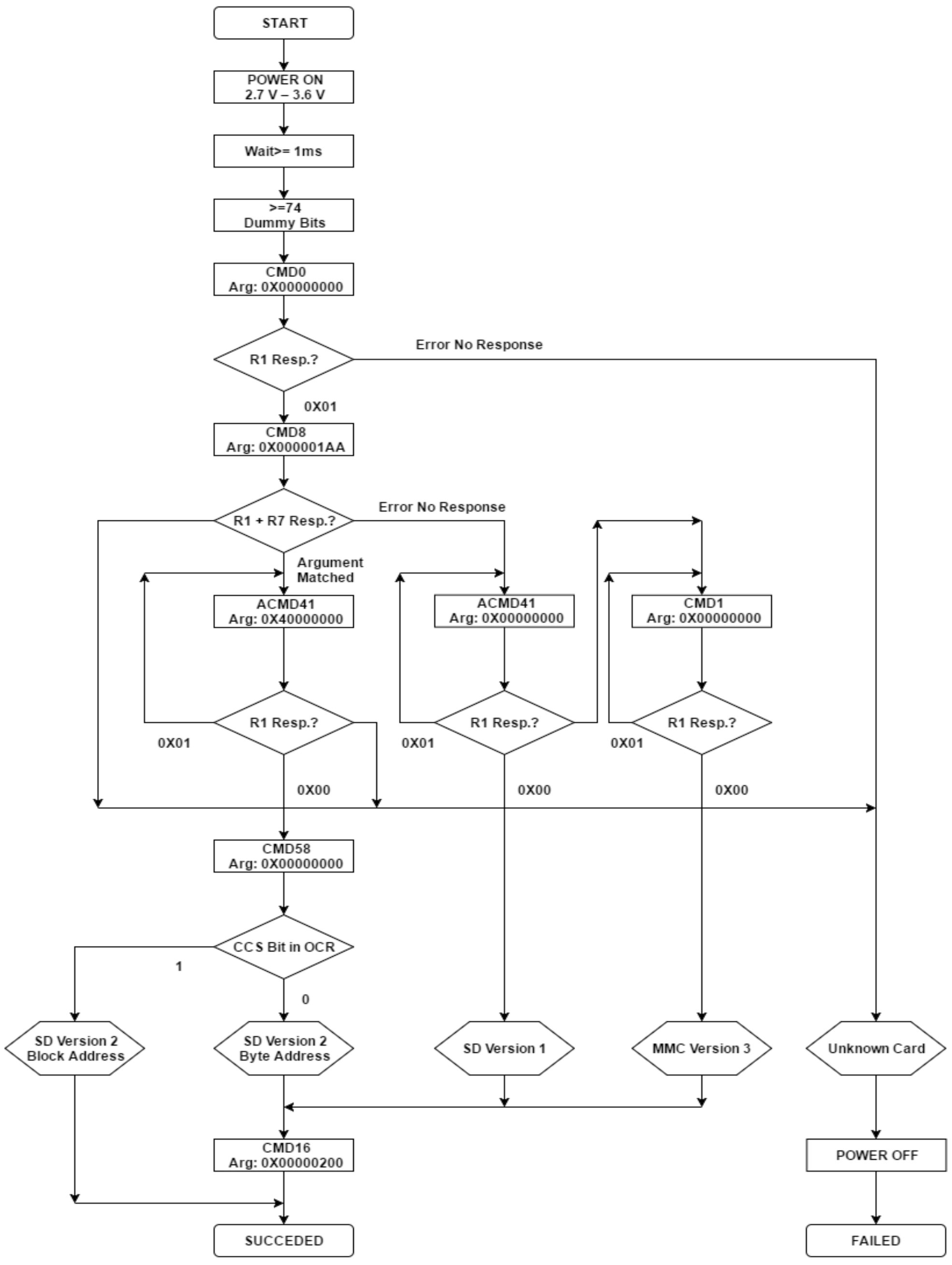


Figure 8 SD Card Initialization Sequence[13]

Writing data to the SD card involved sending the command 24 and then wait for a valid response from the card for more than 16 clock cycles. Then data packet is sent to the card which included data token, 512 bytes of data block and 2 bytes of CRC. In our application, single block write is performed at address 1234h every time the scheduler runs task2.

2.4.3 ADC Driver

ADC drivers available from Free RTOS were used to configure the on-board ADC module for temperature measurement. The library function to use the ADC driver when task 1 is running was developed. The working of the ADC driver can be explained in brief as follows. ADC channel zero has been used in this application. Temperature measurement is an example of a periodic task in our application. The trigger is generated by the processor via a ADCProcessorTrigger function. For each sample a sequence number is defined and is updated after every sample. Also, a priority is set for each sample. The sequence to convert the analog input values can be listed as follows:

- Select the ADC channel to be used.
- Assign the source of trigger and priority of a sample source.
- Configure step of the sample sequencer.
- Enable the ADC channel to start sampling.

2.5 Software Design:

2.5.1 Creation of a task:

Before creating a task, memory locations must be reserved to hold the data of each task to be executed. In our program, an array of structures was created and it contained pointer to each task and includes variables such as task ID, worst case execution time, period of the task, deadline of the task, number of times the task has occurred and about the status of the task if it's ready to execute or its running.

Instruction for task creation: xTaskCreate (task1," task name",stack size ,null ,priority ,null)

The task gets created in the main function. The function used for the creation of the task assigns values for all the variables defined in the structure. The periodic, aperiodic and sporadic tasks are created in the main function i.e. the first time the program executes. When the task gets created it is considered as ready to be executed and thus its variables are set accordingly. Also, parameters like the task ID, execution time required and deadline are decided while it is being created and are passed as parameters to the function.

2.5.2 Real Time Clock:

The real-time clock on board is used to keep track of the period. It plays a vital role while designing a scheduler. The TM4C Series TM4C123G comes with 5 clock sources; two external, two internal and one programmable. The system tick is generated by the 32-kHz clock source which we enable in the main function. Using this clock source, we generate an interrupt at a rate which can be determined by the programmer. This interrupt is used to check the status of the various tasks and is also the time at which the scheduler decides the priority of the tasks. The RTC is initialized in the main function. While initializing the RTC, we decide the number of system ticks we want to have per second. The system ticks are the interrupts; these interrupts are handled by the interrupt handler for the RTC. The number of ticks per second is decided as two raised to the power of the value we have given. Note that the time periods and deadlines of the tasks given can be decided by the programmer to be either the number of system ticks or to a set amount of time given in seconds. Thus, in this method the scheduler will be called every time a system tick occurs and not at every clock cycle. The implication of this is that a task can be called or interrupted only when a system tick has occurred. Thus, we need to ensure that the number of ticks per second is not too low for tasks to fail to execute.

2.5.3 Schedule-Ability Testing:

The schedule-ability test function determines whether any set of tasks which are to be executed can complete without any of them missing their deadlines. A task might miss its deadline when an aperiodic or user defined task is executed. If the tasks can be executed without missing their deadlines no special action is taken but If not the users task is rejected. The program can also be modified so that the least important or the lowest priority periodic is rejected instead.
The function works by calculating a value which determines whether the tasks can be executed or not. In the case of only periodic tasks, what occurs is that the worst-case execution time of the task

is divided by its period, which is also its deadline. This value is then added with the values obtained for all the other periodic tasks. If aperiodic tasks are also present a slight modification is made. In this case, we calculate the time required to complete the remainder of the task (for cases when an aperiodic task occurs during the middle of a periodic task) and divide it by the value obtained by multiplying the deadline with the number of times the task has executed minus the total number of ticks done. This is repeated and added for all the periodic tasks. For the aperiodic task, we follow the method mentioned for only periodic tasks and add it to the obtained value. It is represented in equation format as shown below. [7]

For Periodic Tasks, only: $\frac{\text{WCET for periodic tasks}}{Period\ for\ periodic\ tasks}$

For Both Tasks: $\frac{\text{WCET}-\text{Time Done}+1\text{ for periodic tasks}}{(\text{Period}*\text{Iteration})-\text{Ticks}}+\frac{\text{WCET for periodic tasks}}{Deadline}$

WCET = Worst Case Execution Time or just Execution Time
In either case, if the final value of sum is greater than one it indicates that the tasks cannot be executed without one or the other missing its deadline. When this case occurs, we discard one task. Here we discard the aperiodic task. This is done by setting a variable which acts as a flag; this causes the aperiodic task to get discarded in the external interrupt handler. Note that if sum is greater than one for only periodic tasks the program, in its present format, will not run.

2.5.4 Ready Queue:

A maximum of N lists is created for N tasks and tasks are assigned priority per EDF or LST algorithm and tasks are put in the ready queue per their priority. Example: if three tasks are created and three lists, list1, list2 and list3, are created, then the task with least slack time is added to list3 in ready queue and that with largest slack is added to list1.
If slack/deadline is equal, then it is added to tail in same list. Task in highest priority list in ready queue is taken and given to processor for execution.

2.5.5 Scheduler:

The scheduler is the heart of our program. It decides which tasks to run and determines status of the task. In the function, the priority of the tasks is determined. Using a while loop, the scheduler then performs a series of checks for each of the tasks present to determine the course of action to be followed. It starts from the task given the task ID of zero through the last task. The first step performed is to check whether the task has completed its process for the given iteration. If it has completed all its steps, the task is reset. The second step is to check whether a task is ready to be called. This is performed only for periodic tasks. This task is performed second to allow for the fact a task may be called again immediately after it has finished executing. In this case, we activate the task.

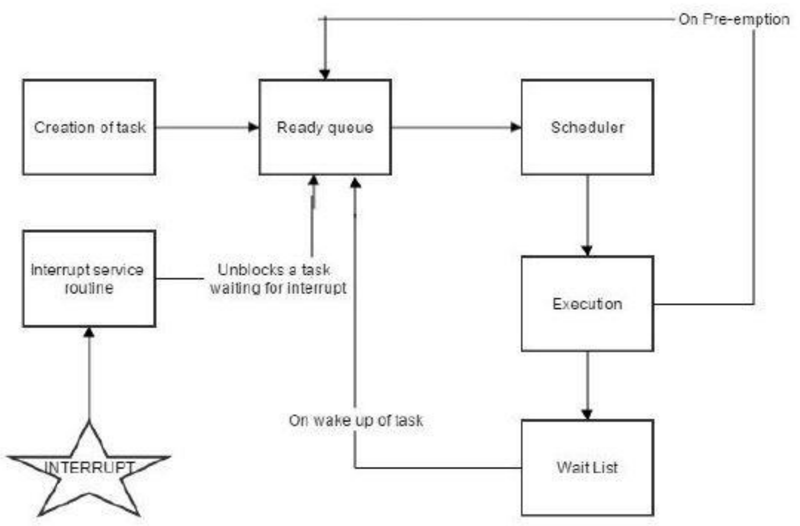


Figure 9 Block Diagram for Scheduling

The priority of each task is determined individually and then compared with each other to determine which task to execute. For both periodic and aperiodic tasks, the priority is determined only if the task is ready to execute; otherwise it is ignored for that task. For the first periodic task, the difference between the deadline and the system tick is compared with a variable kept at a random high number. If the value is lower than the variable, the variable's value changes to the calculated difference. This process is then repeated for the other periodic tasks. This way, the task with the least difference will be given the highest priority. Note that at this instance the term deadline refers to the system tick time at which the task should be completed and not the period it has to complete.

The formula used to make this calculation is given below:
Periodic Task: $Least=((Period*Iteration) -SysTick)\ if<Least$

Aperiodic Task: $Least = ((Period+Time\ at\ Task\ Call) -SysTick)\ if<Least$
The highest priority task is determined by above calculation. If no task has priority (occurs if no task is ready), the program enters the main function and remains idle until a task is ready to execute. If any task has highest priority, the program flow moves to that task's function. Before the task is called, its ticks-completed is incremented and its run variable is set. The task is called using the pointer variable created in the structure.

LST: All the tasks in ready queue is prioritised in rearrange function. A 'for' n cycle loop for n task is being executed so as new task arrives it is compared to all task in ready queue lists. Then it enters if condition to check the slack time and gets inserted in appropriate priority list. if the slack of the new task is greater than slack in the list i then it is compared with tasks present with list 'i-1'. if slack is equal then it is added to tail in list i.
If less than it is added to list 'i+1'. this process continues till it satisfies the for condition. Then a task from highest priority list is taken and given to processor for execution.

Slack time = deadline – remaining execution time - current time
In Free RTOS slack is calculated by:
Slack time = (wakeup time + period) – WCET – tickcount

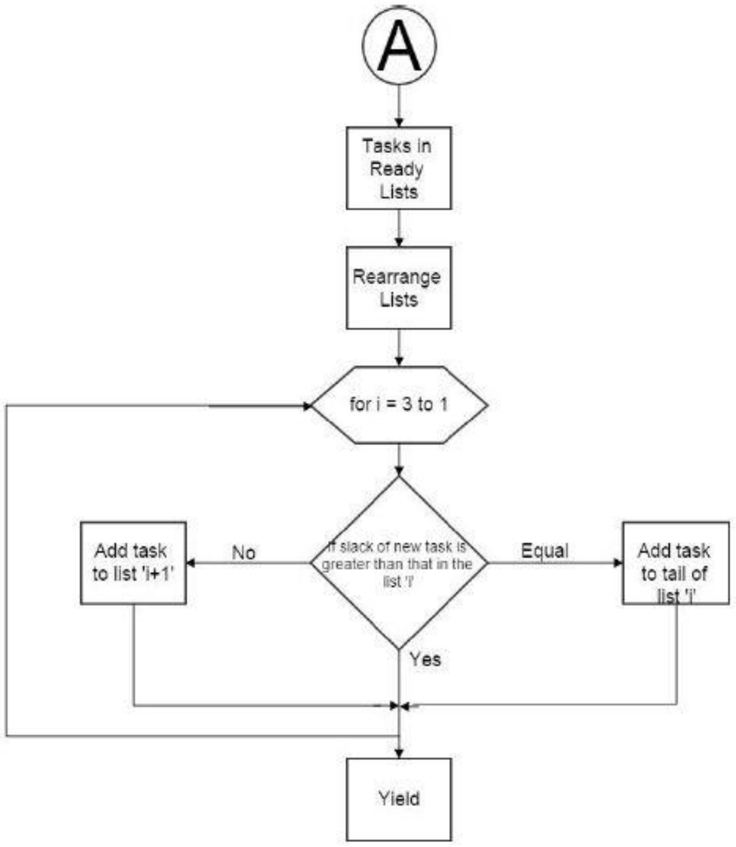


Figure 10 Flowchart for LST Algorithm

EDF: All the tasks in ready queue is prioritised in rearrange function. A ’for’ n cycle loop for n task is being executed so as new task arrives it is compared to all task in ready queue lists. Then it enters if condition to check the Earliest Deadline and gets inserted in appropriate priority list. if the deadline of the new task is greater than slack in the list i then it is compared with tasks present with list ‘i-1’. if deadline is equal then it is added to tail in list i. If deadline is less it is added to list ‘i+1’. This process continues till it satisfies the for condition. Then a task from highest priority list is taken and given to processor for execution.

In Free RTOS deadline is calculated by:
Deadline = (wakeup time +period)

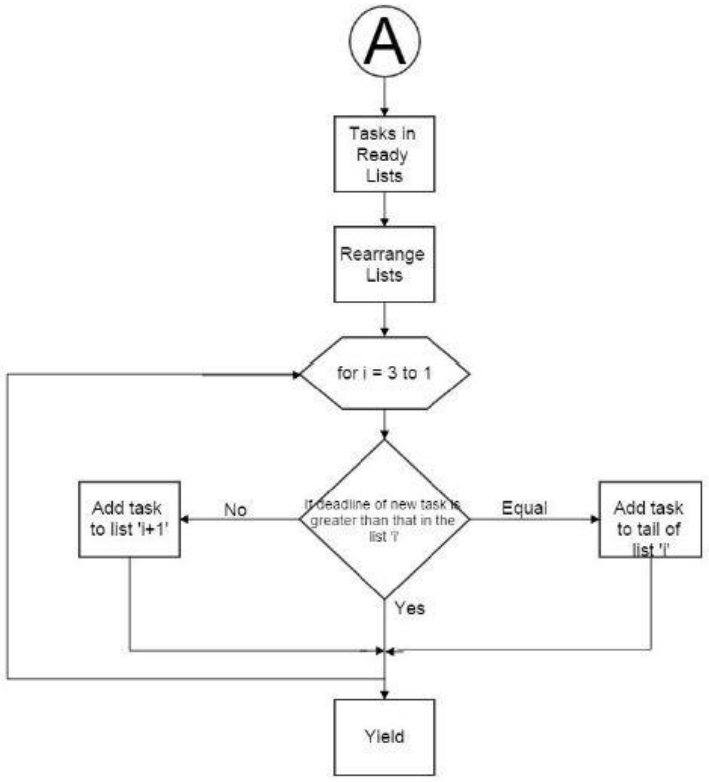

A
Tasks in Ready Lists
Rearrange Lists
for i = 3 to 1
If deadline of new task is greater than that in the list 'i'
No
Add task to list 'i+1'
Equal
Add task to tail of list 'i'
Yes
Yield

Figure 11 Flowchart for EDF algorithm

2.5.6 Execution

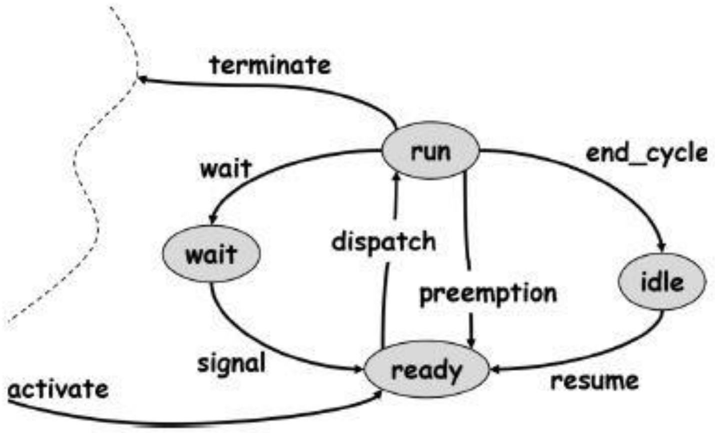


Figure 12 Execution states and switching of tasks[2]

A task from ready queue list is taken and is being executed in processor. On completion of task it is put into wait list until it woken again i.e. on wake-up time task in wait list is added to ready queue. If running task in processor is pre-empted by another task, then task is added to ready queue list. If no task is present to execute then processor calls **idle** function so that at all times a processor is busy.

2.5.7 Interrupt and Interrupt service routine

Binary Semaphores used for Synchronization
A Binary Semaphore can be used to unblock a task each time an interrupt occurs, effectively synchronizing the task with the interrupt. This allows most the interrupt event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR. The interrupt processing is said to have been ‘deferred’ to a ‘handler’ task.
If the interrupt processing is particularly time critical then the handler task priority can be set to ensure the handler task always pre-empts the other tasks in the system. It will then be the task that the ISR returns to when the ISR itself has completed executing. This has the effect of ensuring the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself.

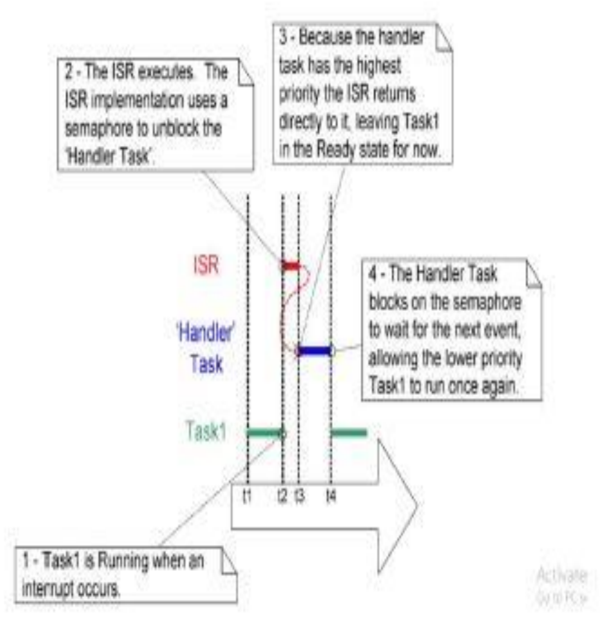

2 - The ISR executes. The ISR implementation uses a semaphore to unblock the 'Handler Task'.
3 - Because the handler task has the highest priority the ISR returns directly to it, leaving Task1 in the Ready state for now.
ISR
'Handler' Task
4 - The Handler Task blocks on the semaphore to wait for the next event, allowing the lower priority Task1 to run once again.
Task1
t1
t2
t3
t4
1 - Task1 is Running when an interrupt occurs.

Figure 13 Working of an Interrupt[2]

In this interrupt synchronization scenario, the semaphore can be conceptually thought of as a queue that has a length of one. The queue can contain a maximum of one item at any time so is always either empty or full (hence binary). By calling xSemaphoreTake () the handler task effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty. When the event occurs the ISR simply uses the xSemaphoreGiveFromISR () function to place a token (the semaphore) into the queue, making the queue full. This causing the handler task to exit the Blocked state and remove the token, leaving the queue empty once more.

Once the handler task has completed its processing it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event.

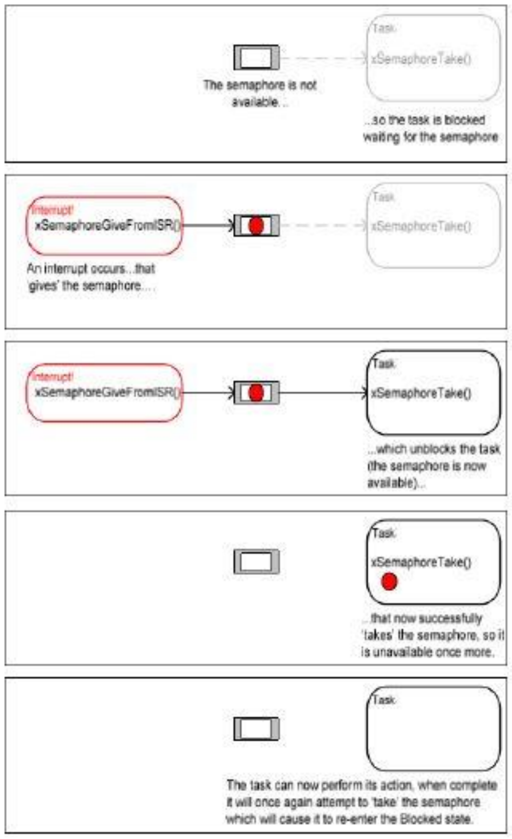
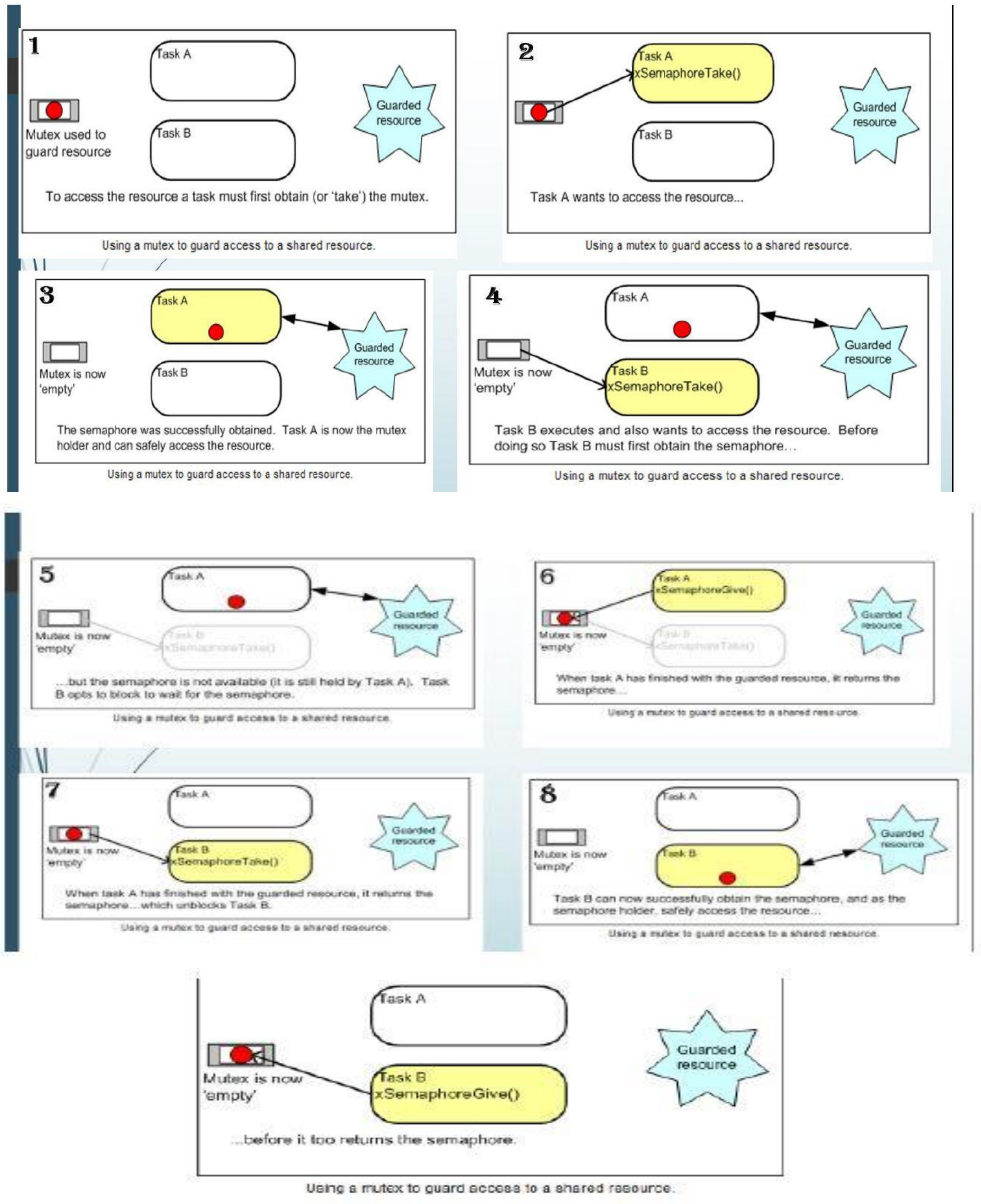

Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
An interrupt occurs...that 'gives' the semaphore....
...which unblocks the task (the semaphore is now available)...
...that now successfully 'takes' the semaphore, so it is unavailable once more.
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

Figure 14 Using binary semaphore to synchronize task with an interrupt [2]

2.5.8 Mutex and Semaphore:

Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

When used for mutual exclusion the mutex acts like a token that is used to guard a resource. When a task wishes to access the resource, it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.



To access the resource a task must first obtain (or 'take') the mutex.

Task A wants to access the resource...

The semaphore was successfully obtained. Task A is now the mutex holder and can safely access the resource.

Task B executes and also wants to access the resource. Before doing so Task B must first obtain the semaphore...

...but the semaphore is not available (it is still held by Task A). Task B opts to block to wait for the semaphore.

When task A has finished with the guarded resource, it returns the semaphore...

When task A has finished with the guarded resource, it returns the semaphore...which unblocks Task B.

Task B can now successfully obtain the semaphore, and as the semaphore holder, safely access the resource...

...before it too returns the semaphore.

Using a mutex to guard access to a shared resource.

Figure 15 Mutex and Semaphore Concepts [2]

2.5.9 Task Functions:

Each of the tasks created has a function to depict periodic, aperiodic and sporadic function. Since our project is to implement the scheduler of an RTOS, we have created time critical real-time applications to show CPU utilisation and performance.

When the task is called, its execution time required and the time it has already executed are both stored in local variables. The main function of the task is kept inside an infinite while loop. This ensures that the task keeps performing its duty until an interrupt occurs. When the RTC or external interrupts occurs the variable to call the scheduler is set. Thus, inside the while loop of the task, we need to have mechanism which activates when this variable is set. Depending on the task states, the serial monitor displays whether the task is active or "running", is ready to run or "waiting", or has stopped running to allow a higher priority task to execute or "blocked". Separate functions are used for the display of the above information and they are executed once per system tick.

2.5.10 Main Function:

As most the work is done by the scheduler and the schedule-ability test functions. By separating the various duties of the program into separate function, we have kept the program simple to understand as each module does one task only. In the main function, we enable the RTC clock source. We then initialize the RTC of the board; its working has been described earlier. The periodic tasks are created at this stage. Just like the tasks, the main function contains an infinite while loop. The reason is that when no tasks are ready, the program flow will return to the main function; the loop ensures that the program doesn't stop. Inside the loop, we call the scheduler if the appropriate RTC interrupt has occurred. If a task is ready to be executed the program flow will move to that task.

2.5.11 Applications:

The real-time tasks are classified into three main categories: periodic, sporadic, and aperiodic tasks.

Periodic Task: A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually demarcated by clock interrupts. For this reason, periodic tasks are sometimes referred to as clock-driven tasks. The fixed time interval after which a task repeats is called the period of the task [9]. We will be demonstrating this by using a temperature sensor which measures temperature and displays on serial monitor after certain interval time.

Sporadic Task: A sporadic task is one that recurs at random instants. A sporadic task Ti can be represented by a three tuple:
Ti = (ei, gi, di)
Where ei is the worst-case execution time of an instance of the task, gi denotes the minimum separation between two consecutive instances of the task, di is the relative deadline. The minimum separation (gi) between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before gi time units have elapsed [9], which will be demonstrated by displaying the number of times periodic function is executed, whenever it is prime number on serial monitor. Prime number function is sporadic in nature.

Aperiodic Task: An aperiodic task is in many ways like a sporadic task. An aperiodic task can arise at random instants. However, in case of an aperiodic task, the minimum separation gi between two consecutive instances can be 0. That is, two or more instances of an aperiodic task might occur at the same time instant [9], which is being demonstrated by Morse code application i.e. Morse code is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be

directly understood by a skilled listener or observer without special equipment. Each Morse code symbol represents either a text character (letter or numeral) or a pro-sign and is represented by a unique sequence of dots and dashes [10].
Example of a Morse code: S is represented in Morse code as · · ·
We have implemented Morse code application by interfacing three switches as interrupts. One switch for indicating dot, second for dash and the third switch to indicate end of input. The output will be displayed on serial monitor.
For implementing the different tasks the shared resource is seven segment display and the UART or printing any message on the serial terminal. This resource is controlled by Mutex and for synchronization between the interrupts and the task, a binary semaphore is used.
vSemaphoreCreateBinary(share);
share= xSemaphoreCreateMutex ();
Here share is a semaphore handler. Semaphore and Mutex are created using the functions shown above.

2.5.12 Test Methodology:

For the testing of our project, we have used the serial monitor, LED present on the Tiva TM4C123GXL board and seven segment display to indicate the task which is being executed and 3 push buttons (two present on the board and other interfaced externally on breadboard) for Morse code application. The LED blinks to indicate the task being run. The serial monitor on the Energia HyperTerminal also gives information on the states of the other tasks. Each periodic task is indicated whether it is ready, has been blocked, is inactive or is executing. By allowing us to view the different states of the tasks, we could identify and rectify any cases where a task was missing its deadline or was not being executed correctly. A multiple of test cases were used to ensure the correct working of the code.

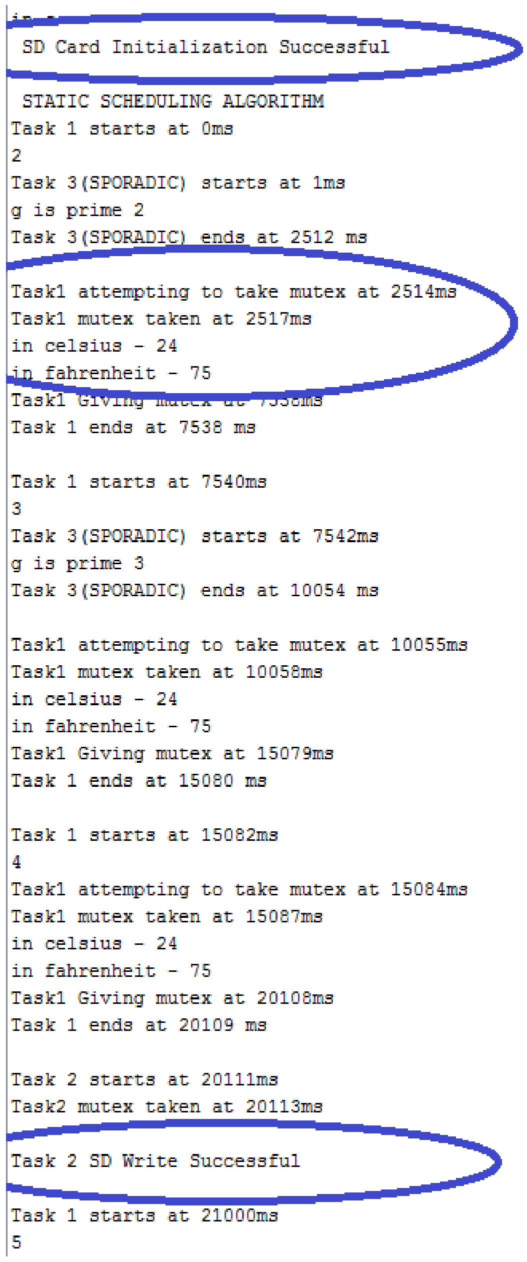
The two scheduling algorithms, EDF and LST are run on the Tiva TM4C123GXL board and the following tests are performed:

1. Few Tasks are dependent of each other and rest are independent of each other and are Schedulable per the condition for schedulability.
2. All Tasks are Schedulable, with different Execution Times and different Deadlines:

Here the tasks will have definite priorities. By priority we mean that tasks which have lower slack will be required to execute earlier than other tasks. A variety of cases can be achieved by varying the deadline and execution times. By doing so tasks with various processor utilization are scheduled and the performance is compared between the two scheduling algorithms.

3 RESULTS AND ANALYSIS:

The initialization function involves configuring the UART module and initializing the SD card reader properly. This sequence remains same for all algorithms.



```
 SD Card Initialization Successful

 STATIC SCHEDULING ALGORITHM
Task 1 starts at 0ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2
Task 3(SPORADIC) ends at 2512 ms

Task1 attempting to take mutex at 2514ms
Task1 mutex taken at 2517ms
in celsius - 24
in fahrenheit - 75
Task1 Giving mutex at 7538ms
Task 1 ends at 7538 ms

Task 1 starts at 7540ms
3
Task 3(SPORADIC) starts at 7542ms
g is prime 3
Task 3(SPORADIC) ends at 10054 ms

Task1 attempting to take mutex at 10055ms
Task1 mutex taken at 10058ms
in celsius - 24
in fahrenheit - 75
Task1 Giving mutex at 15079ms
Task 1 ends at 15080 ms

Task 1 starts at 15082ms
4
Task1 attempting to take mutex at 15084ms
Task1 mutex taken at 15087ms
in celsius - 24
in fahrenheit - 75
Task1 Giving mutex at 20108ms
Task 1 ends at 20109 ms

Task 2 starts at 20111ms
Task2 mutex taken at 20113ms

Task 2 SD Write Successful

Task 1 starts at 21000ms
5
```

Figure 16 Temperature sensor and SD card functionality

Test Case 1:

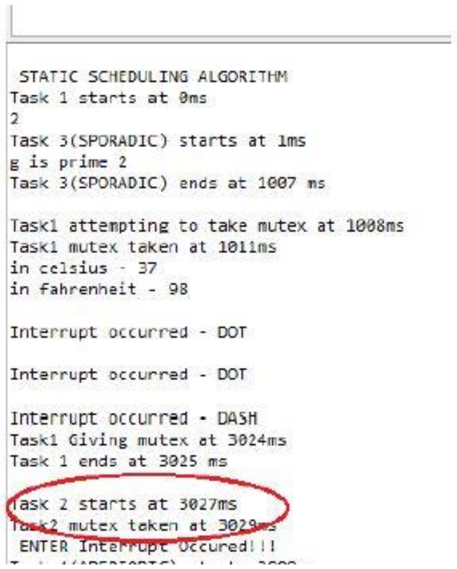
Test Case	Execution Time	Deadline
Task 1	2	4
Task 2	2	6
Task 3	1	4
Task 4	1	4

Table 3.1

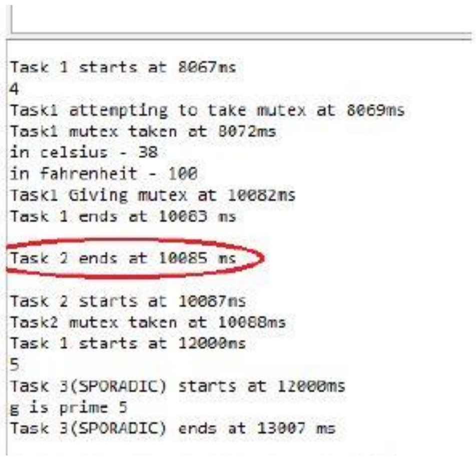
CPU Utilization depends on the arrival of sporadic and aperiodic tasks. If both arrive within 6 seconds CPU utilization becomes 99%.

Static:

The tasks were scheduled based on the static priority assigned at the time of their creation. A low priority task executes only when all the higher priority tasks are in the wait list.



```
 STATIC SCHEDULING ALGORITHM
Task 1 starts at 0ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2
Task 3(SPORADIC) ends at 1007 ms

Task1 attempting to take mutex at 1008ms
Task1 mutex taken at 1011ms
in celsius - 37
in fahrenheit - 98

Interrupt occurred - DOT

Interrupt occurred - DOT

Interrupt occurred - DASH
Task1 Giving mutex at 3024ms
Task 1 ends at 3025 ms

Task 2 starts at 3027ms
Task2 mutex taken at 3029ms
 ENTER Interrupt Occured!!!
```

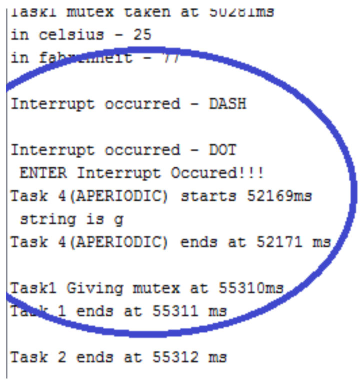


```
Task 1 starts at 8067ms
4
Task1 attempting to take mutex at 8069ms
Task1 mutex taken at 8072ms
in celsius - 38
in fahrenheit - 100
Task1 Giving mutex at 10082ms
Task 1 ends at 10083 ms

Task 2 ends at 10085 ms

Task 2 starts at 10087ms
Task2 mutex taken at 10088ms
Task 1 starts at 12000ms
5
Task 3(SPORADIC) starts at 12000ms
g is prime 5
Task 3(SPORADIC) ends at 13007 ms
```

Figure 17 Screenshots of Static Algorithm

For the first burst of execution (i.e. 18 seconds), Task2 starts at 3027 ms but ends at 10085ms. But, the deadline of Task2 is 6000ms. Task2 has missed its deadline. Thus, it is observed that in case of a static scheduler the lower priority task is starved and hence misses deadline.

The case when the on board, pushbuttons are pressed to give user input and its execution in terms of static scheduling can be seen below.

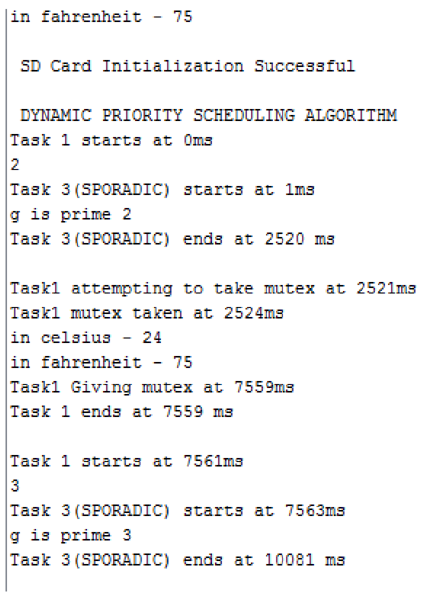


```
Task1 mutex taken at 50281ms
in celsius - 25
in fahrenheit - 77

Interrupt occurred - DASH

Interrupt occurred - DOT
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 52169ms
 string is g
Task 4(APERIODIC) ends at 52171 ms

Task1 Giving mutex at 55310ms
Task 1 ends at 55311 ms

Task 2 ends at 55312 ms
```

Figure 18 Interrupt occurrence Static Algorithm

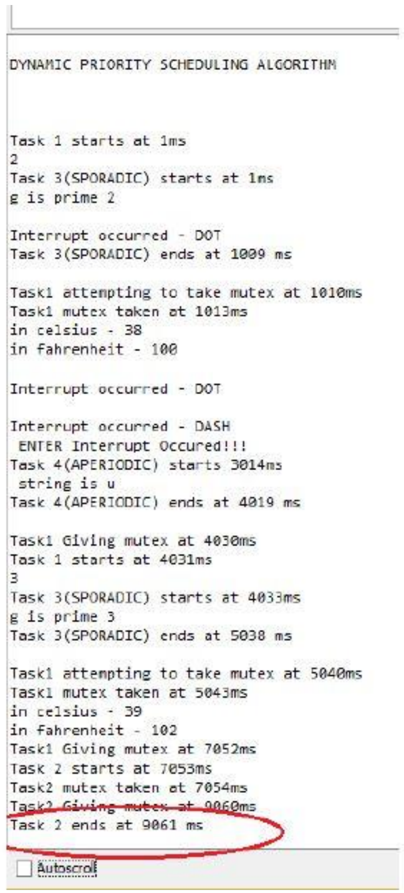
EDF:
The tasks were scheduled based on the execution time and their deadlines. The periodic task is pre-empted at moments where another task with lesser deadline arrives such that none of the tasks miss their deadlines. The tasks finish their execution and are inserted into a wait list until it wakes up again. The tasks which are pre-empted are rearranged in the ready queue such that a higher priority task can execute. Aperiodic and sporadic tasks are executed along with periodic tasks but those tasks are given higher priority than periodic tasks to process interrupt service.

The initial sequence of execution of the EDF can be as seen below



```
in fahrenheit - 75

 SD Card Initialization Successful

 DYNAMIC PRIORITY SCHEDULING ALGORITHM
Task 1 starts at 0ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2
Task 3(SPORADIC) ends at 2520 ms

Task1 attempting to take mutex at 2521ms
Task1 mutex taken at 2524ms
in celsius - 24
in fahrenheit - 75
Task1 Giving mutex at 7559ms
Task 1 ends at 7559 ms

Task 1 starts at 7561ms
3
Task 3(SPORADIC) starts at 7563ms
g is prime 3
Task 3(SPORADIC) ends at 10081 ms
```

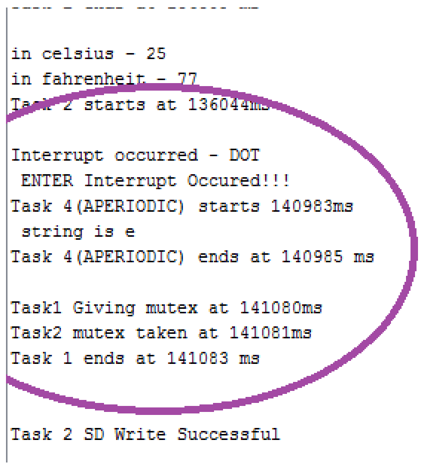
Figure 19 Screenshot 1 of EDF Algorithm



```
DYNAMIC PRIORITY SCHEDULING ALGORITHM


Task 1 starts at 1ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2

Interrupt occurred - DOT
Task 3(SPORADIC) ends at 1009 ms

Task1 attempting to take mutex at 1010ms
Task1 mutex taken at 1013ms
in celsius - 38
in fahrenheit - 100

Interrupt occurred - DOT

Interrupt occurred - DASH
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 3014ms
 string is u
Task 4(APERIODIC) ends at 4019 ms

Task1 Giving mutex at 4030ms
Task 1 starts at 4031ms
3
Task 3(SPORADIC) starts at 4033ms
g is prime 3
Task 3(SPORADIC) ends at 5038 ms

Task1 attempting to take mutex at 5040ms
Task1 mutex taken at 5043ms
in celsius - 39
in fahrenheit - 102
Task1 Giving mutex at 7052ms
Task 2 starts at 7053ms
Task2 mutex taken at 7054ms
Task2 Giving mutex at 9060ms
Task 2 ends at 9061 ms
```

Figure 20 Screenshot 2 of EDF Algorithm

Within the first burst of execution task 2 misses' deadline of seconds.
The case when the on board, pushbuttons are pressed to give user input and its execution in terms of EDF scheduling can be seen below.



```
in celsius - 25
in fahrenheit - 77
Task 2 starts at 136044ms

Interrupt occurred - DOT
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 140983ms
 string is e
Task 4(APERIODIC) ends at 140985 ms

Task1 Giving mutex at 141080ms
Task2 mutex taken at 141081ms
Task 1 ends at 141083 ms


Task 2 SD Write Successful
```

Figure 21 Interrupt occurrence EDF algorithm

LST:
The tasks were scheduled based on the slack times. The periodic task is pre-empted at moments where another task with lesser slack time arrives such that none of the tasks miss their deadlines. The tasks finish their execution and are inserted into a wait list until it wakes up again. The tasks which are pre-empted are rearranged in the ready queue such that a higher priority task can execute. In cases where one of tasks finishes execution early another task maybe scheduled to guarantee highest possible utilization of the processor. Aperiodic and sporadic tasks are executed along with periodic tasks but those tasks are given higher priority than periodic tasks to process interrupt service.

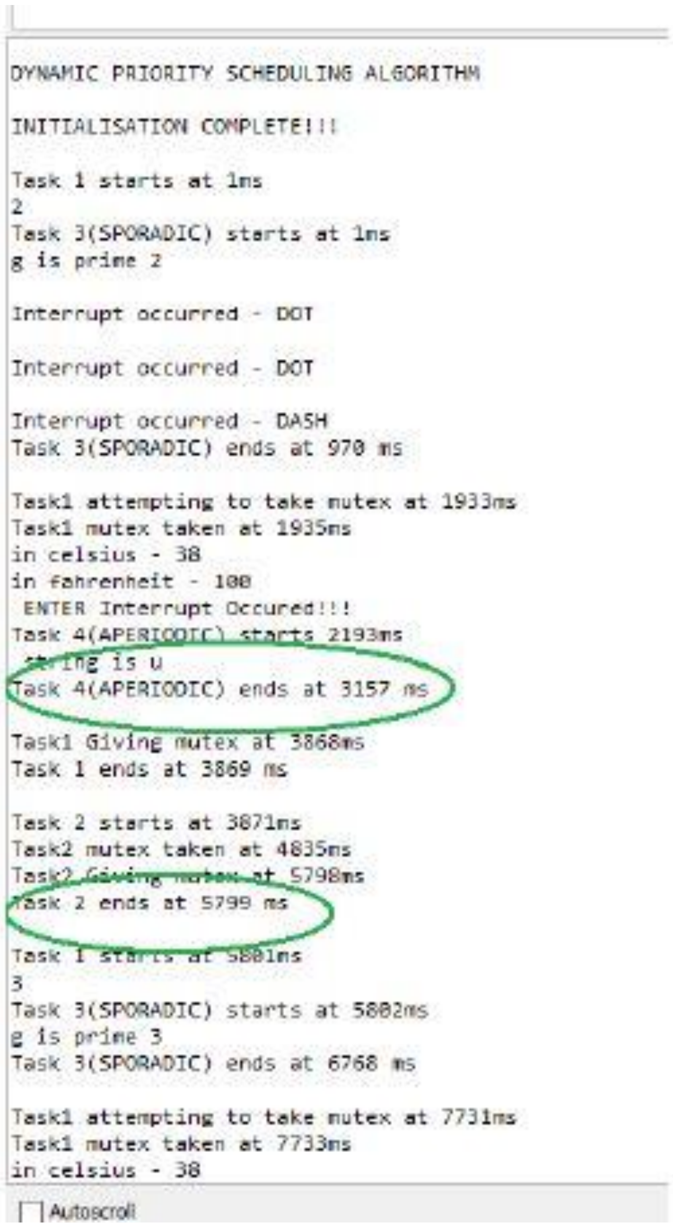
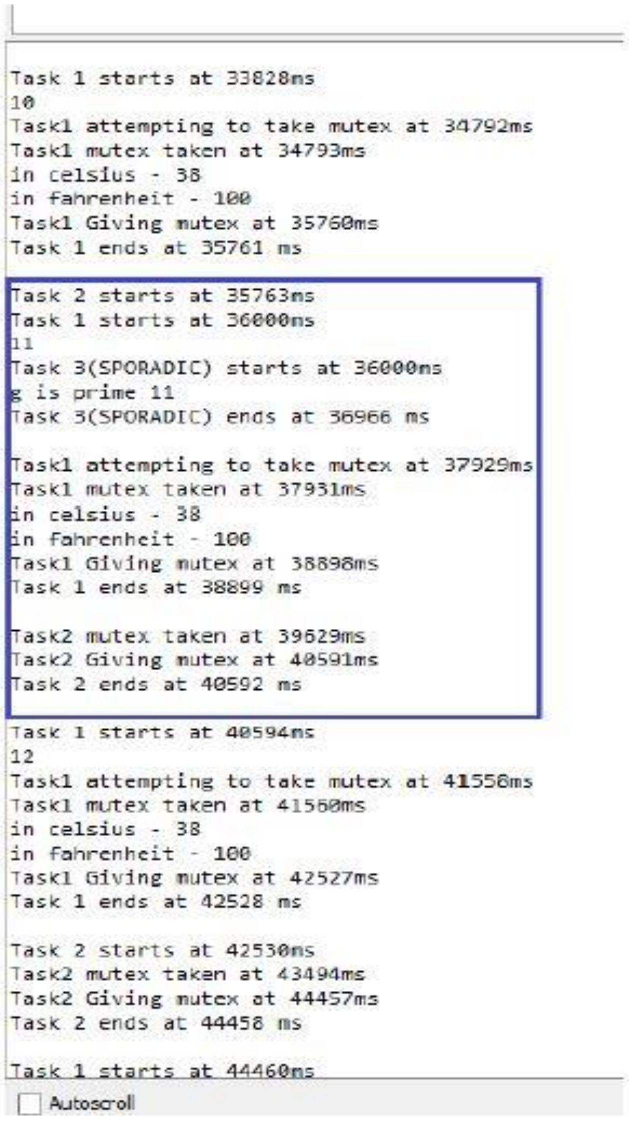

```
DYNAMIC PRIORITY SCHEDULING ALGORITHM

INITIALISATION COMPLETE!!!

Task 1 starts at 1ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2

Interrupt occurred - DOT

Interrupt occurred - DOT

Interrupt occurred - DASH
Task 3(SPORADIC) ends at 970 ms

Task1 attempting to take mutex at 1933ms
Task1 mutex taken at 1935ms
in celsius - 38
in fahrenheit - 100
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 2193ms
 string is u
Task 4(APERIODIC) ends at 3157 ms

Task1 Giving mutex at 3868ms
Task 1 ends at 3869 ms

Task 2 starts at 3871ms
Task2 mutex taken at 4835ms
Task2 Giving mutex at 5798ms
Task 2 ends at 5799 ms

Task 1 starts at 5801ms
3
Task 3(SPORADIC) starts at 5802ms
g is prime 3
Task 3(SPORADIC) ends at 6768 ms

Task1 attempting to take mutex at 7731ms
Task1 mutex taken at 7733ms
in celsius - 38
```

Figure 22 Screenshot 1 of LST Algorithm



```
Task 1 starts at 33828ms
10
Task1 attempting to take mutex at 34792ms
Task1 mutex taken at 34793ms
in celsius - 38
in Fahrenheit - 100
Task1 Giving mutex at 35760ms
Task 1 ends at 35761 ms

Task 2 starts at 35763ms
Task 1 starts at 36000ms
11
Task 3(SPORADIC) starts at 36000ms
g is prime 11
Task 3(SPORADIC) ends at 36966 ms

Task1 attempting to take mutex at 37929ms
Task1 mutex taken at 37931ms
in celsius - 38
in Fahrenheit - 100
Task1 Giving mutex at 38898ms
Task 1 ends at 38899 ms

Task2 mutex taken at 39629ms
Task2 Giving mutex at 40591ms
Task 2 ends at 40592 ms
Task 1 starts at 40594ms
12
Task1 attempting to take mutex at 41556ms
Task1 mutex taken at 41560ms
in celsius - 38
in Fahrenheit - 100
Task1 Giving mutex at 42527ms
Task 1 ends at 42528 ms

Task 2 starts at 42530ms
Task2 mutex taken at 43494ms
Task2 Giving mutex at 44457ms
Task 2 ends at 44458 ms

Task 1 starts at 44460ms
```

Figure 23 Screenshot 2 of LST algorithm

Within the first burst of execution (i.e. 6 seconds), Task2 starts executing at 3891 ms and mutex is taken. It completes execution at 5799ms. Thus, Task2 meets its deadline of 6000ms.aperiodic and sporadic tasks are being executed. The instant where task 1 and task 2 clash to acquire same resource is also shown using a blue box.

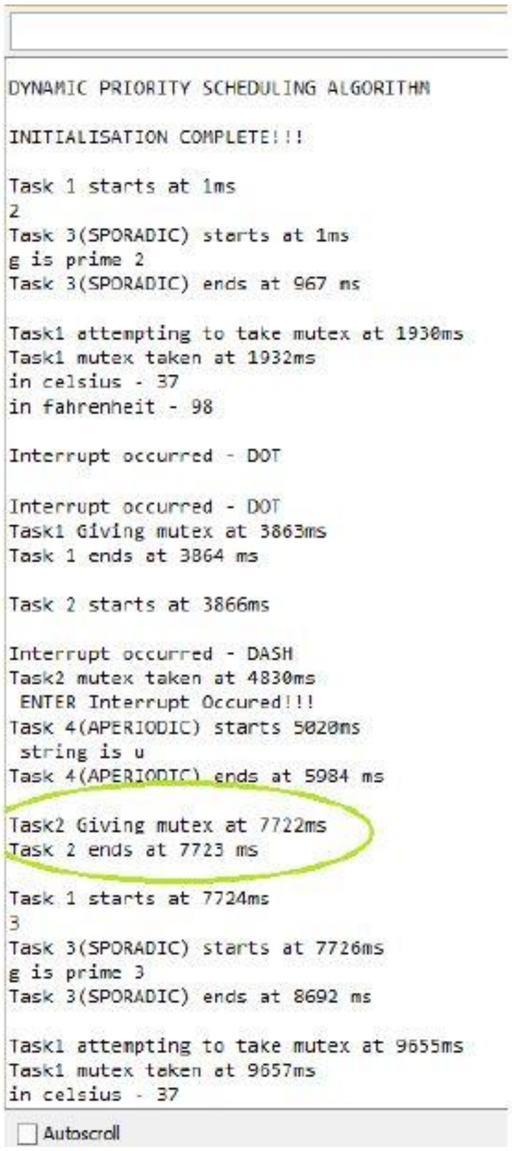
For aperiodic task- morse code '.._' i.e. dot-dot-dash is being given as input to board using push buttons and character 'u' which is being displayed.

Test Case 2:

Test Case	Execution Time	Deadline
Task 1	3	6
Task 2	3	9
Task 3	1	2
Task 4	1	2

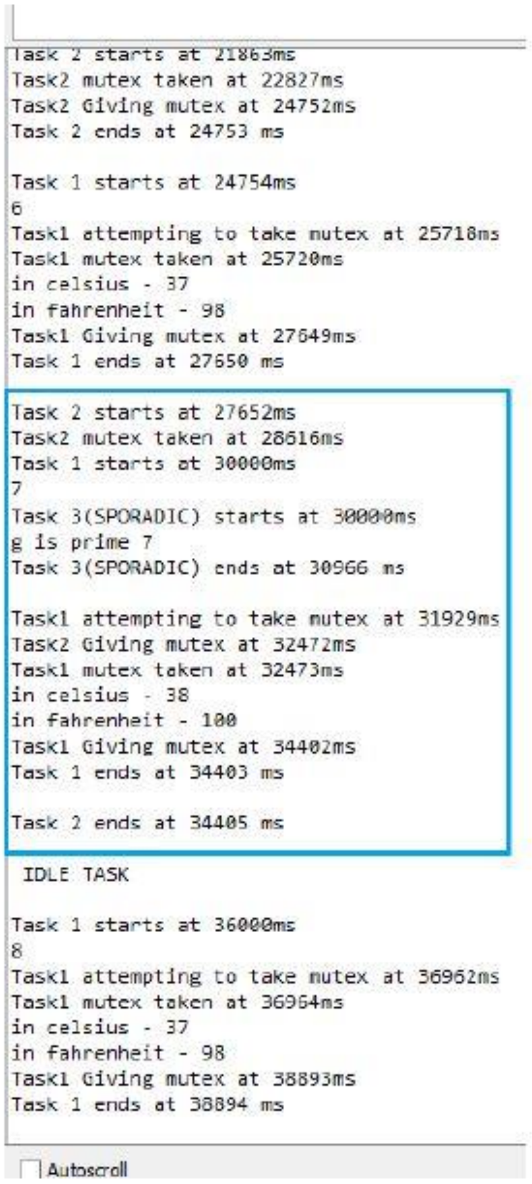
Table 3.2

CPU Utilization depends on the arrival of sporadic and aperiodic tasks. If both arrive within 6 seconds CPU utilization becomes 88.88%



```
DYNAMIC PRIORITY SCHEDULING ALGORITHM

INITIALISATION COMPLETE!!!

Task 1 starts at 1ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2
Task 3(SPORADIC) ends at 967 ms

Task1 attempting to take mutex at 1930ms
Task1 mutex taken at 1932ms
in celsius - 37
in fahrenheit - 98

Interrupt occurred - DOT

Interrupt occurred - DOT
Task1 Giving mutex at 3863ms
Task 1 ends at 3864 ms

Task 2 starts at 3866ms

Interrupt occurred - DASH
Task2 mutex taken at 4830ms
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 5020ms
 string is u
Task 4(APERIODIC) ends at 5984 ms

Task2 Giving mutex at 7722ms
Task 2 ends at 7723 ms

Task 1 starts at 7724ms
3
Task 3(SPORADIC) starts at 7726ms
g is prime 3
Task 3(SPORADIC) ends at 8692 ms

Task1 attempting to take mutex at 9655ms
Task1 mutex taken at 9657ms
in celsius - 37
```

Figure 24 Screenshot 3 of LST Algorithm



```
Task 2 starts at 21863ms
Task2 mutex taken at 22827ms
Task2 Giving mutex at 24752ms
Task 2 ends at 24753 ms

Task 1 starts at 24754ms
6
Task1 attempting to take mutex at 25718ms
Task1 mutex taken at 25720ms
in celsius - 37
in fahrenheit - 98
Task1 Giving mutex at 27649ms
Task 1 ends at 27650 ms

Task 2 starts at 27652ms
Task2 mutex taken at 28616ms
Task 1 starts at 30000ms
7
Task 3(SPORADIC) starts at 30000ms
g is prime 7
Task 3(SPORADIC) ends at 30966 ms

Task1 attempting to take mutex at 31929ms
Task2 Giving mutex at 32472ms
Task1 mutex taken at 32473ms
in celsius - 38
in fahrenheit - 100
Task1 Giving mutex at 34402ms
Task 1 ends at 34403 ms

Task 2 ends at 34405 ms
 IDLE TASK

Task 1 starts at 36000ms
8
Task1 attempting to take mutex at 36962ms
Task1 mutex taken at 36964ms
in celsius - 37
in fahrenheit - 98
Task1 Giving mutex at 38893ms
Task 1 ends at 38894 ms
```

Figure 25 Screenshot 4 of LST algorithm

A second case was used to test LST with CPU Utilization of 99.9%. Within the first burst of execution (i.e. 9 seconds), Task2 starts executing at 3866 ms and is pre-empted at 7723ms by interrupt service i.e. aperiodic task. But after interrupt servicing, task 2 completes execution at 5066ms. Thus, Task2 meets its deadline of 9000ms. Similarly, like case 1, aperiodic task and sporadic tasks are being executed successfully.

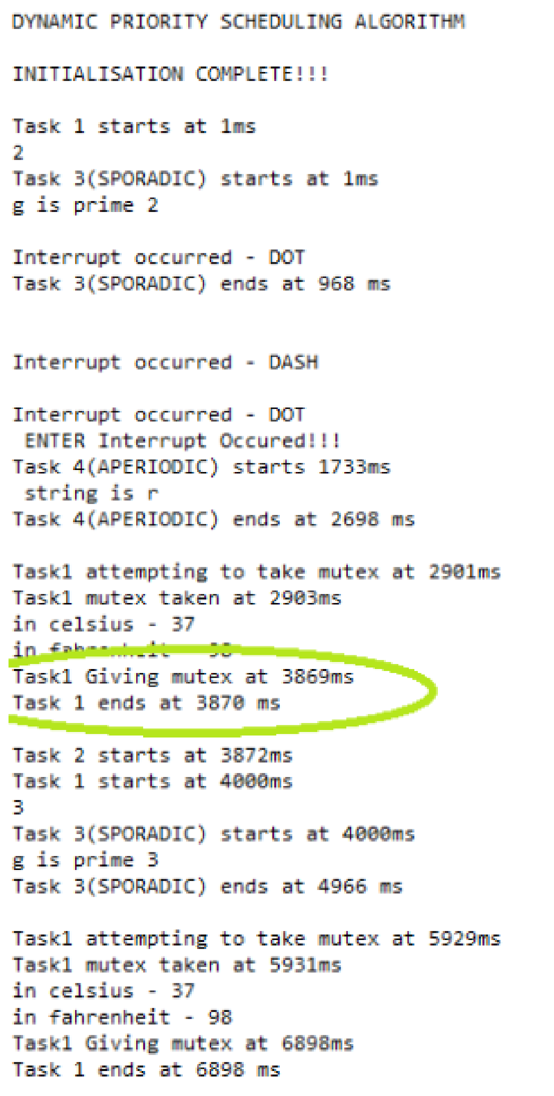
Test Case 3:

Test Case	Execution Time	Deadline
Task 1	2	4
Task 2	3	18
Task 3	1	6
Task 4	1	4

Table 3.3

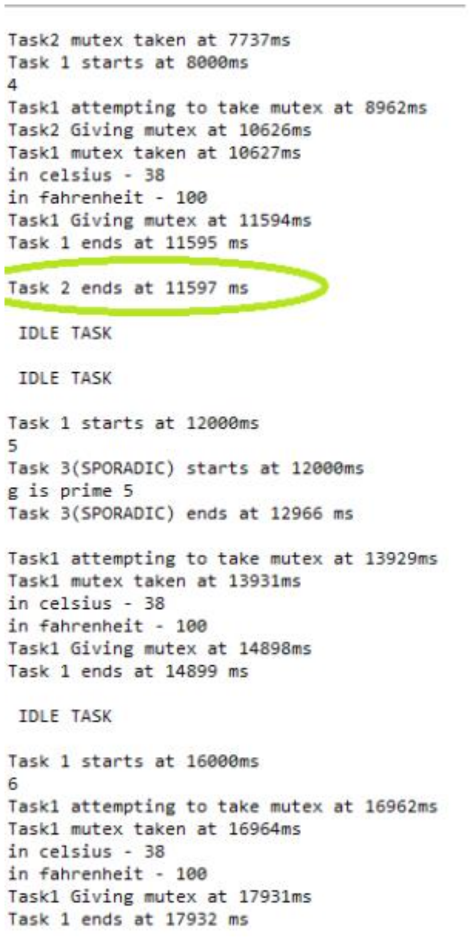
CPU Utilization is 67% for test case 3.

A third case was used to test LST with lower CPU Utilization of 83.3%. Within the first burst of execution (i.e. 18 seconds), Task2 starts executing at 3870 ms and pre–empted by task1 at 4000 ms but task 2 completes execution at 11997ms. Thus, Task2 meets its deadline of 18000ms. Similarly, like case 1 and case 2, aperiodic task and sporadic tasks are being executed successfully. Screenshots of LST algorithm are as follows;



```
DYNAMIC PRIORITY SCHEDULING ALGORITHM

INITIALISATION COMPLETE!!!

Task 1 starts at 1ms
2
Task 3(SPORADIC) starts at 1ms
g is prime 2

Interrupt occurred - DOT
Task 3(SPORADIC) ends at 968 ms


Interrupt occurred - DASH

Interrupt occurred - DOT
 ENTER Interrupt Occured!!!
Task 4(APERIODIC) starts 1733ms
 string is r
Task 4(APERIODIC) ends at 2698 ms

Task1 attempting to take mutex at 2901ms
Task1 mutex taken at 2903ms
in celsius - 37
in fahrenheit - 98
Task1 Giving mutex at 3869ms
Task 1 ends at 3870 ms

Task 2 starts at 3872ms
Task 1 starts at 4000ms
3
Task 3(SPORADIC) starts at 4000ms
g is prime 3
Task 3(SPORADIC) ends at 4966 ms

Task1 attempting to take mutex at 5929ms
Task1 mutex taken at 5931ms
in celsius - 37
in fahrenheit - 98
Task1 Giving mutex at 6898ms
Task 1 ends at 6898 ms
```

Figure 26 Screenshot 6 of LST Algorithm



```
Task2 mutex taken at 7737ms
Task 1 starts at 8000ms
4
Task1 attempting to take mutex at 8962ms
Task2 Giving mutex at 10626ms
Task1 mutex taken at 10627ms
in celsius - 38
in fahrenheit - 100
Task1 Giving mutex at 11594ms
Task 1 ends at 11595 ms

Task 2 ends at 11597 ms

 IDLE TASK

 IDLE TASK

Task 1 starts at 12000ms
5
Task 3(SPORADIC) starts at 12000ms
g is prime 5
Task 3(SPORADIC) ends at 12966 ms

Task1 attempting to take mutex at 13929ms
Task1 mutex taken at 13931ms
in celsius - 38
in fahrenheit - 100
Task1 Giving mutex at 14898ms
Task 1 ends at 14899 ms

 IDLE TASK

Task 1 starts at 16000ms
6
Task1 attempting to take mutex at 16962ms
Task1 mutex taken at 16964ms
in celsius - 38
in fahrenheit - 100
Task1 Giving mutex at 17931ms
Task 1 ends at 17932 ms
```

Figure 27 Screenshot 7 of LST Algorithm

4 CHALLENGES FACED:

4.1Operating System:

Choosing an operating system to work on was one of the challenges. We tried to develop the scheduling algorithm on Sys/Bios, a new operating system introduced by TI. Due to lack of good documentation, we were lagging in our timeline for the project. Hence, we decided to work with Free-RTOS.

4.2 Understanding and Implementing:

As we were new to operating system concepts, understanding and implementing all the theoretical concepts into out design was another big challenge. It consumed a lot of time to understand the flow of the scheduling algorithm in general.

4.3 Execution time and Deadline:

Choosing appropriate execution time and deadline was a challenge too. An algorithm can be scheduled only if the CPU utilization is less than 100% which considers the execution time and deadline for computation.

4.4 SD card:

Using Driver libraries instead of register level coding was a hard task. As register level coding was not supported in the OS development, we had to use driver libraries. It turned out to be an issue while setting the frequency for SD card initialization.

5 CONCLUSION:

In this project, various applications such as periodic, aperiodic and sporadic were implemented successfully. Two periodic tasks were implemented, one with an on-board temperature sensor configured with ADC and another performing a SD card write of the time at which Task 2 was executed configured with SPI. Libraries for temperature sensor and SD card was written for the periodic tasks. The sporadic task was executed whenever the periodic task with a temperature sensor was executed prime number of times and aperiodic task which was interrupt driven was executed with a Morse code circuitry. The concepts of mutex and semaphore was implemented on the scheduler to demonstrate resource sharing and synchronization between tasks or interrupt and the task. The real-time applications were tested for different time criticalities. The performance of static priority scheduling, EDF scheduling and LST scheduling was analyzed and in the test cases considered, LST proved to perform better than the rest of the algorithms.

6 FUTURE SCOPE:

The energy consumption of the algorithm can be analysed using tools in CCS or Keil and develop an algorithm with increased performance and reduces power consumption.

Scheduler needs to be developed to support multicore processors along with memory utilization and CPU utilization.

7 ACKNOLEDGEMENTS:

We would like to extend our gratitude to Prof. Linden McClure for allowing us to be a part of this course which has been a great holistic learning experience comprising of hardware, software and firmware design knowledge. This experience resulted in the culmination of a successful final project.

We are also grateful to all the Teaching Assistants, Virag Doshi, Meher Jain, and Anish Churi, for their support and guidance throughout the duration of this semester and course.

We would also like to thank the authors of all the references cited below.

8 REFERENCES:

8.1 Papers:

1) Giorio C. Buttazzo, “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications”, Third Edition, Springer, 2011.
2) Mats Pettersson, “Basic Concepts for Real Time Operating Systems”, IAR Systems
3) “Design and Implementation of Deadline based EDF Algorithm on ARM LPC2148”, Proceedings of 2013 IEEE Conference on Information and Communication Technologies (ICT 2013)
4) M. Kaladevi, S. Sathiyabama, “A Comparative Study of Scheduling Algorithms for Real Time Task”, International Journal of Advances in Science and Technology, Vol. 1, No. 4, 2010.
5) “Least Slack Time Rate First: New Scheduling Algorithm for Multi-Processor Environment.”, CONFERENCE PAPER · JANUARY 2010

8.2 Books/Reports:

6) Philip A. Laplante, “Real Time Systems Design and Analysis”, Third Edition, Wiley India, 2005
7) Implementing Scheduling Algorithms”, Real-Time and Embedded Systems (M) Lecture 9, University of Glasgow
8) Introduction to Free-RTOS”, Deepak D'Souza,Department of Computer Science and Automation Indian Institute of Science, Bangalore.

8.3 Web pages:

9) Wikipedia, “Embedded System”, Available at http://en.wikipedia.org/wiki/Embedded_system

8.4 YouTube/any other reference:

10) RTOS Tutorial”, YouTube Lecture Series, Available at: https://www.youtube.com/watch?v=smQ2s0ldRGA
11) FreeRTOS Task and Queues Tutorials” by Millsinghion Available at: https://www.youtube.com/watch?v=8lIpI30Tj-g

8.5 Images

12) https://www.engineersgarage.com/contributions/anjali/How-to-display-font-on-seven-segment-display
13) http://elm-chan.org/docs/mmc/mmc_e.html
14) https://www.sparkfun.com/products/97
15) http://www.geeetech.com/wiki/index.php/Arduino_SD_card_Module
16) http://wiring.org.co/learning/tutorials/breadboard/

9 APPENDICES:

9.1 Appendix- Bill of Materials

Part Description	**Source**	**Cost (in $)**
TM4C123G	Texas Instruments	12
SD Card Reader	Arduino	7
Seven Segment Display		1
Resistors		2
Pushbutton		1
Miscellaneous		5
TOTAL		28

9.2 Appendix- Schematics

9.3 Appendix- Firmware Source Code

9.4 Appendix- Software Source Code

9.5 Appendix- Data Sheets and Application Notes