Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Homework 3

Submitted by

Parth Thakkar

Submitted on April 5, 2024

Contents

List of Figures

List of Tables

PDF is clickable

1 Question 1

Q: Create a user-defined interrupt handler for the timer ISR and a task for processing. The timer should be scheduled on a regular basis, and the interrupt handler should signal the processing task. To ensure that the timer is being triggered with the correct periodicity, pass the interrupt timing to the processing task.
Answer:
Code Flow and Meaning:

1. The code starts by including the necessary header files:
   - ¡stdint.h¿ and ¡stdbool.h¿ for standard integer types and boolean type.
   - "main.h" and "drivers/pinout.h" for project-specific configurations and pin definitions.
   - "utils/uartstdio.h" for UART communication functionality. Various TivaWare header files ("driverlib/...") for accessing TivaC hardware features.
   - Various TivaWare header files ("driverlib/...") for accessing TivaC hardware features.
   - FreeRTOS header files ("FreeRTOS.h", "task.h", "queue.h", etc.) for using FreeRTOS functionality.
2. The code declares global variables:
   - task1SyncSemaphore is a binary semaphore handle used for synchronization.
   - Task1_handle is a task handle for xTask1.
   - Hz is set to 100, representing the desired frequency for the timer interrupt.
   - ulPeriod is used to store the calculated timer period.
3. The Timer0Isr() function is the timer interrupt service routine (ISR)
   - It is called whenever the Timer0 interrupt occurs.
   - It retrieves the current tick count using xTaskGetTickCount().
   - It clears the timer interrupt flag using ROM_TimerIntClear().
   - It sends a task notification to Task1_handle using xTaskNotifyFromISR(), passing the current tick count as the notification value.
4. The xTask1() function is the task that waits for the timer notification:
   - It is created with a stack size of configMINIMAL_STACK_SIZE and a priority of 2.
   - It enters an infinite loop.
   - Inside the loop, it waits for a notification using xTaskNotifyWait() with a maximum block time of 5000 ms.
   - If a notification is received (xResult == pdPASS), it retrieves the current tick count and the notified value.
   - It prints a message using UARTprintf(), indicating the task completion time and the received timer interrupt data (tick count).
5. The main() function is the entry point of the program:
   - It initializes the system clock to 120 MHz using ROM_SysCtlClockFreqSet().
   - It initializes the GPIO pins for the LaunchPad using PinoutSet().
   - It configures the UART for stdio output at a baud rate of 230400 using UARTStdioConfig().
   - It enables and configures Timer0 as a periodic timer using ROM_SysCtlPeripheralEnable() and ROM_TimerConfigure().

- It registers the timer ISR Timer0Isr() using TimerIntRegister().
- It calculates the timer period based on the desired frequency (Hz) and the system clock rate.
- It loads the timer with the calculated period using ROM_TimerLoadSet() and enables the timer using ROM_TimerEnable().
- It enables the Timer0 interrupt using ROM_IntEnable() and ROM_TimerIntEnable().
- It creates a binary semaphore task1SyncSemaphore using xSemaphoreCreateBinary().
- It creates the task xTask1 using xTaskCreate().
- Finally, it starts the FreeRTOS scheduler using vTaskStartScheduler().

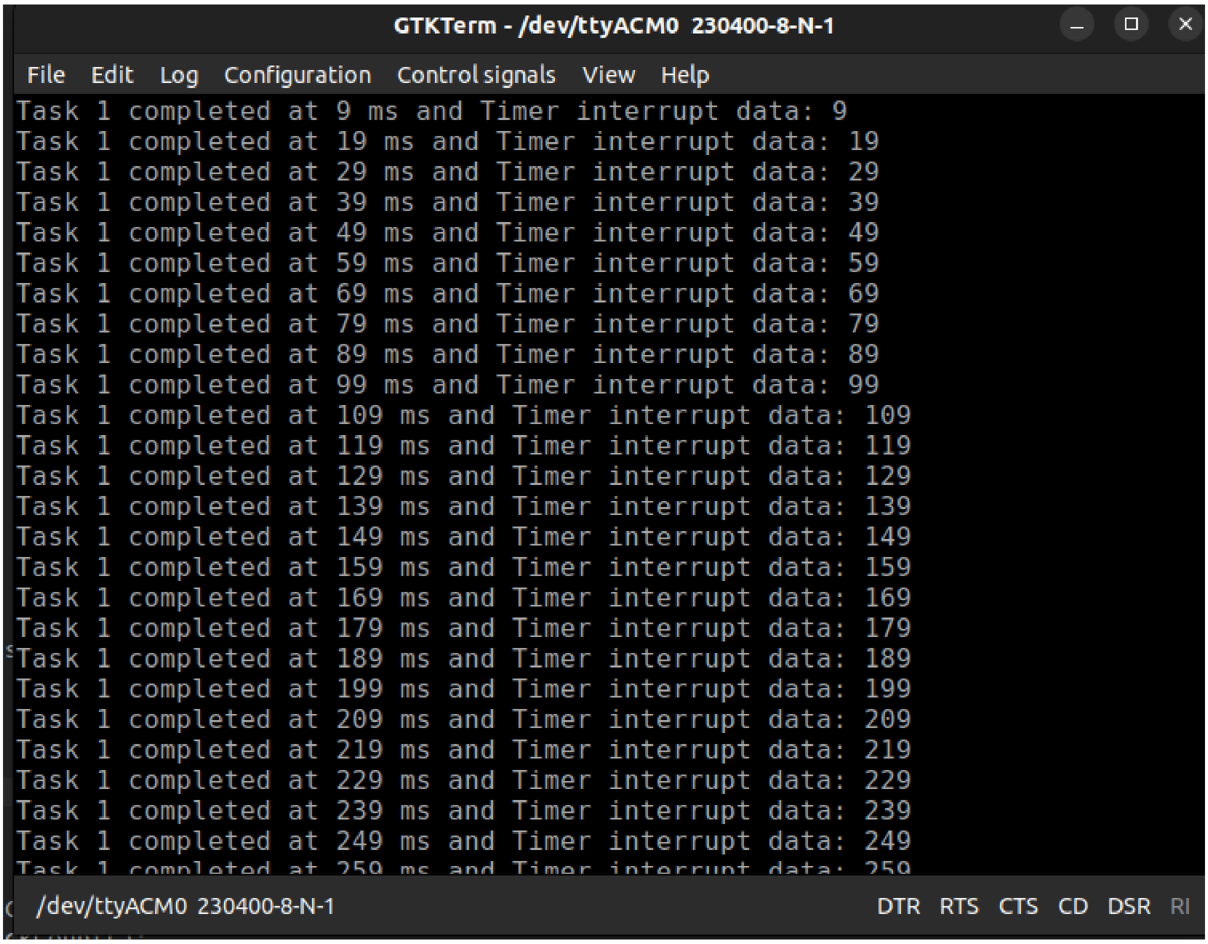

Figure 1: Semaphore has been posted by this program

Output Analysis:

1. The output of the code, as shown in the provided screenshot, displays the task completion messages printed by xTask1 at regular intervals. Each message includes the task completion time in milliseconds and the timer interrupt data (tick count) received through the notification.

Here's a detailed analysis of the output:

1. The first message indicates that Task 1 completed at 9 ms, and the timer interrupt data received is 9. This means that the first timer interrupt occurred at around 9 ms after the program started, and xTask1 received the notification with the tick count value of 9.

2. The subsequent messages show that Task 1 completes at regular intervals of approximately 10 ms, which corresponds to the timer frequency of 100 Hz. For example, the second message shows Task 1 completing at 19 ms with a timer interrupt data of 19, indicating that the second timer interrupt occurred 10 ms after the first one.

3. The tick count value in each message increments by 10 ms compared to the previous message. This confirms that the timer interrupt is occurring at the specified frequency of 100 Hz, and the tick count is accurately reflecting the time elapsed since the program started.

4. The output continues with Task 1 completing at 29 ms, 39 ms, 49 ms, and so on, with the corresponding timer interrupt data matching the completion time. This demonstrates the consistent and accurate synchronization between the timer interrupt and the task notification. The program keeps running indefinitely, with xTask1 receiving notifications and printing messages at regular intervals of 10 ms, until it is manually stopped or the desired runtime is reached.

The output verifies that the code is functioning as intended, with the timer interrupt triggering at the specified frequency of 100 Hz and xTask1 receiving notifications and printing the task completion time and timer interrupt data accordingly. This confirms the successful setup and synchronization between the timer interrupt and the task using FreeRTOS task notifications.

2 Question 2

Q: Create a pair of FreeRTOS tasks that signal each other. The first task performs some computation, signals the other task, and waits for a signal from that task. The second task repeats the same pattern so that they alternate. Each task should complete a defined amount of work, such as computing a specified number of Fibonacci values or some equivalent synthetic load. Do not use sleep functions as a load. Profile each task, by storing timestamps that can be printed at the end, with one task executing for 10 ms and the other for 40 ms. Run for at least 200 ms. Printing can be done using UARTprintf().

Answer:

The goal of this question is to create two FreeRTOS tasks that communicate with each other using signals (semaphores) and perform computations alternately. The tasks should follow a specific pattern of execution:

1. Task 1 performs a computation, signals Task 2, and then waits for a signal from Task 2.
2. Task 2, upon receiving the signal from Task 1, performs its computation, signals Task 1, and waits for a signal from Task 1.
3. This pattern repeats, with the tasks alternating their execution.

Each task should have a defined amount of work to complete, such as calculating a specific number of Fibonacci values or any other computational load. It's important to note that sleep functions should not be used as a load, as they would not represent actual computational work.

Profiling is required to measure the execution time of each task. Timestamps should be stored at appropriate points in the code to track the start and end times of each task's execution. The goal is to have Task 1 execute for approximately 10 milliseconds and Task 2 execute for approximately 40 milliseconds in each iteration.

The overall program should run for at least 200 milliseconds to allow for multiple iterations of the alternating task execution.

Finally, the stored timestamps and any relevant information should be printed using the UARTprintf() function, which sends the output to the UART (Universal Asynchronous Receiver/Transmitter) for display or logging purposes.

Detailed Code Analysis:

1. The fiboncacci() function is defined to calculate Fibonacci numbers for a specified duration in milliseconds. It uses a loop to calculate Fibonacci numbers up to FIB_LIMIT_FOR_32_BIT and keeps track of the elapsed time using xTaskGetTickCount().
2. The xTask1() function represents the first task:
   - It enters a loop that runs until the total runtime (TIME_TO_RUN) is reached.
   - Inside the loop, it waits for the task1SyncSemaphore using xSemaphoreTake().
   - Once the semaphore is obtained, it calls the fiboncacci() function to perform calculations for 10 ms.
   - After the calculations, it prints the current time and the time taken to execute the Fibonacci function using UARTprintf().
   - Finally, it gives the task2SyncSemaphore to signal Task 2. The xTask2() function represents the second task and follows a similar
3. pattern as xTask1():

- It waits for the task2SyncSemaphore.
- Upon receiving the semaphore, it performs Fibonacci calculations for 40 ms.
- It prints the execution time using UARTprintf().
- It gives the task1SyncSemaphore to signal Task 1.

4. The main() function serves as the entry point of the program:

   - It initializes the system clock, GPIO pins, and UART using the provided TivaWare functions.
   - It creates the semaphores task1SyncSemaphore and task2SyncSemaphore using xSemaphoreCreateBinary().
   - It creates xTask1 and xTask2 using xTaskCreate(), specifying their respective function names, stack sizes, and priorities.
   - The task1SyncSemaphore is given initially to start the alternating execution.
   - The startTimeTick is recorded to keep track of the total runtime.
   - Finally, the FreeRTOS scheduler is started using vTaskStartScheduler().

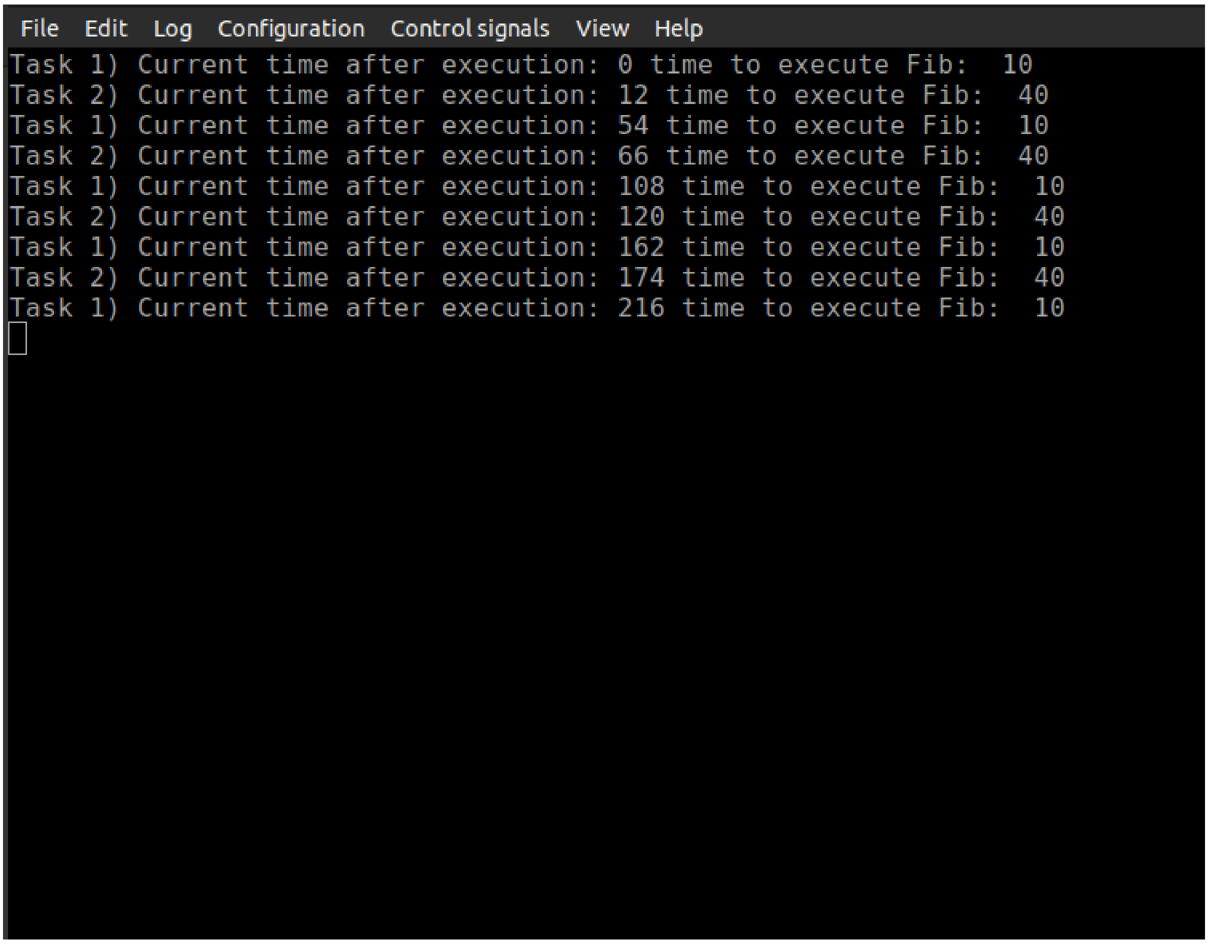

Figure 2: Semaphore has been posted by this program

Detailed Output Analysis:
The provided screenshot shows the output of the program, which demonstrates the alternating execution of Task 1 and Task 2 and their respective execution times.

1. The first line of the output indicates that Task 1 executed at time 0 and took 10 ms to execute the Fibonacci calculations. This is the initial execution of Task 1.

2. The second line shows that Task 2 executed again at time 12 and took 40 ms for the Fibonacci calculations. This implies that Task 2 executed between the first and second executions of Task 1, taking approximately 2 ms (12 - 10 = 2 ms) for its own execution and signaling.

3.

The subsequent lines follow the alternating pattern of Task 1 and Task 2 execution: Task 1 executes at time 54, taking 10 ms for Fibonacci calculations. Task 2 executes again at time 66, indicating that Task 2 executed in between and took approximately 2 ms (66 - 54 - 10 = 2 ms). This pattern continues, with Task 1 executing for 10 ms and Task 2 executing for 40 ms in each iteration. The output continues until the total runtime of 200 ms is reached, as specified by the TIME_TO_RUN constant. The output demonstrates the successful synchronization and alternating execution of Task 1 and Task 2 using semaphores. The timestamps printed show the precise execution times of each task and the time taken for the Fibonacci calculations.

The analysis reveals that Task 1 consistently executes for approximately 10 ms, while Task 2 executes for approximately 40 ms in each iteration. The small gaps between the executions of Task 1 (e.g., 2 ms) can be attributed to the time taken by Task 2 for its own execution and signaling.

Overall, the code provides a practical example of using FreeRTOS tasks, semaphores, and profiling techniques to create a synchronized and alternating execution pattern between two tasks. It showcases the ability to control the execution durations of each task and monitor their performance using timestamps.

The output confirms that the code achieves the desired behavior of alternating task execution, with Task 1 executing for approximately 10 ms and Task 2 executing for approximately 40 ms in each iteration, for a total runtime of at least 200 ms. The timestamps provide valuable insights into the timing and synchronization of the tasks, allowing for performance analysis and optimization if needed.

3 Question 3

Q: Modify the timer ISR to signal two tasks with different frequencies: one task every 30 ms and the other every 80 ms. Use your processing load from Q2 to run 10 ms of processing on the 30-ms task and 40 ms of processing on the 80-ms task. Produce logs that show you have done this.

Answer:

Code Flow and Meaning:

1. The ‘fiboncacci()‘ function calculates Fibonacci numbers for a specified duration in milliseconds. It uses a loop to calculate Fibonacci numbers up to ‘FIB_LIMIT_FOR_32_BIT‘ and breaks the loop if the specified duration is reached.
2. The ‘Timer0Isr()‘ function is the timer interrupt service routine (ISR):
   - It clears the timer interrupt flag.
   - It increments the ‘counter‘ variable.
   - If ’counter’ is divisible by 3, it sends a task notification to ‘Task1_handle‘ with the current tick count.
   - If ‘counter‘ is divisible by 8, it sends a task notification to ‘Task2_handle‘ with the current tick count and resets the ‘counter‘ to 0.
3. The ‘xTask1()‘ function represents the first task:
   - Upon receiving the notification, it prints the task starting time and the received timer interrupt data.
   - It waits for a task notification using ‘xTaskNotifyWait()‘ with a maximum block time of 5000 ms.
   - It calls the ‘fiboncacci()‘ function to perform calculations for 10 ms.
   - It prints the task completion time and the execution duration.
4. The ‘xTask2()‘ function represents the second task and follows a similar pattern as ‘xTask1()‘:
   - It waits for a task notification.
   - Upon receiving the notification, it prints the task starting time,
   - indicating that it preempted Task 1, and the received timer interrupt data.
   - It calls the ‘fiboncacci()‘ function to perform calculations for 40 ms.
   - It prints the task completion time and the execution duration.
5. The ‘main()‘ function initializes the system clock, GPIO pins, UART, and sets up the timer interrupt:
   - It configures Timer0 as a periodic timer with a frequency of 100 Hz.
   - It registers the timer ISR ‘Timer0Isr()‘.
   - It creates the tasks ‘xTask1‘ and ‘xTask2‘ with specified stack sizes and priorities.
   - It sends initial task notifications to both tasks with the starting tick count.
   - It starts the FreeRTOS scheduler.

Output Analysis: The provided output shows the execution timeline of Task 1 and Task 2 based on the timer interrupts and task notifications.

1. The output starts with Task 1 executing at 0 ms and completing at 10 ms, with an execution duration of 10 ms.
2. Task 1 starts again at 29 ms (preempted by Task 1) and completes at 40 ms, with an execution duration of 11 ms.

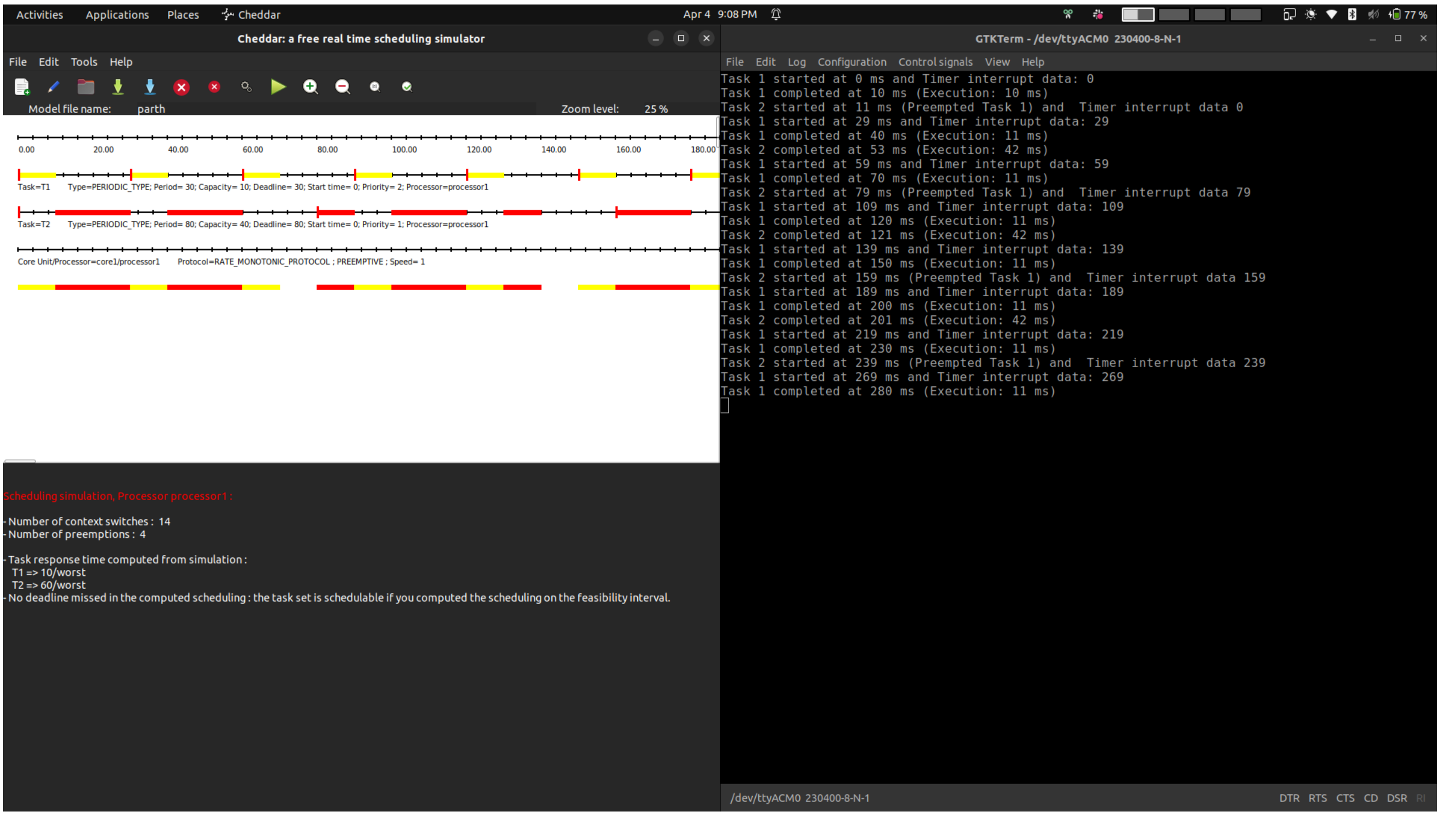

Figure 3: Semaphore has been posted by this program

3. Task 2 starts at 53 ms (preempting Task 1) and completes at 95 ms, with an execution duration of 42 ms.

4. The pattern continues, with Task 1 and Task 2 alternating their execution based on the timer interrupts.

5. Task 1 executes for approximately 10-11 ms each time, while Task 2 executes for approximately 42 ms each time.

6. The timer interrupt data shows the tick count at which the task notification was received.

7. The output demonstrates the successful synchronization and coordination of the tasks using timer interrupts and task notifications.

The code showcases the usage of FreeRTOS tasks, timer interrupts, and task notifications to achieve a synchronized execution pattern between two tasks. The tasks perform computations for specified durations and are triggered by periodic timer interrupts. The output verifies the expected behavior, with Task 1 executing for shorter durations and Task 2 executing for longer durations, while being coordinated by the timer interrupts.

4 Referance

1. REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS with LINUX and RTOS by Sam Siewert John Pratt

2. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment C. L. LIU AND JAMES W. LAYLAND

3. https://bears.ece.ucsb.edu/class/ece253/lect7.pdf

Appendices

A C Code for the Implementation

A.1 Q1

main.c

```c

#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"


// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"
#include "limits.h"


#define FIB_LIMIT_FOR_32_BIT 47
#define TIME_TO_RUN 240 //ms

SemaphoreHandle_t task1SyncSemaphore;
TaskHandle_t Task1_handle;
double Hz = 100;
uint32_t ulPeriod;



void Timer0Isr(void)
{
    TickType_t xCurrentTick = xTaskGetTickCount();
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);  // Clear the timer interrupt

        xTaskNotifyFromISR(Task1_handle, xCurrentTick, eSetValueWithOverwrite, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}



// Process 1
void xTask1(void * pvParameters)
```

```
{


    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
    BaseType_t xResult;
    uint32_t ulNotifiedValue;

    while(1){

        xResult = xTaskNotifyWait( pdFALSE,
                        /* Don't clear bits on entry. */
                        ULONG_MAX,
                        /* Clear all bits on exit. */
                        &ulNotifiedValue, /* Stores the notified value. */
                        xMaxBlockTime );

        if( xResult == pdPASS )
        {

            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 1 completed at %d ms and Timer interrupt data: %d\n",
xCurrentTick, ulNotifiedValue);

        }
    }
}



// Main function
int main(void)
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);


    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, 230400, SYSTEM_CLOCK);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);   // 32 bits Timer
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr);    // Registering  isr


    ulPeriod = (SYSTEM_CLOCK / Hz);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);

    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
```

```c

    task1SyncSemaphore = xSemaphoreCreateBinary();


    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 2, &Task1_handle);

    vTaskStartScheduler();

    return (0);
}


/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```

A.2 Q2

main.c

```c

#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"


// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"


#define FIB_LIMIT_FOR_32_BIT 47
#define TIME_TO_RUN 200 //ms

unsigned long int ulPeriod;
unsigned int Hz = 1;    // frequency in Hz

SemaphoreHandle_t task1SyncSemaphore, task2SyncSemaphore;
TickType_t startTimeTick;


void fiboncacci(int ms){
    TickType_t xStartTick = xTaskGetTickCount();
    TickType_t xCurrentTick = xTaskGetTickCount();
    uint32_t fib = 1, fib_a = 1, fib_b = 1;
    uint32_t i;
    while((xCurrentTick - xStartTick) < pdMS_TO_TICKS(ms)){
        for (i = 0; i < FIB_LIMIT_FOR_32_BIT; i++){
            fib_a = fib_b;
            fib_b = fib;
            fib = fib_a + fib_b;
        }
        xCurrentTick = xTaskGetTickCount();
    }


}

```

```c

// Process 1
void xTask1(void * pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();

    while((xLastWakeTime - startTimeTick) < TIME_TO_RUN){
        if (xSemaphoreTake(task1SyncSemaphore, portMAX_DELAY) == pdTRUE)
        {
            TickType_t xCurrentTick = xTaskGetTickCount();
            fiboncacci(10);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 1) Current time after execution: %d time to execute Fib: %d \n", xCurrentTick, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
            xSemaphoreGive(task2SyncSemaphore);
        }
    }
}


// Process 2
void xTask2(void *pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();

    while ((xLastWakeTime - startTimeTick) < TIME_TO_RUN)
    {


        if (xSemaphoreTake(task2SyncSemaphore, portMAX_DELAY) == pdTRUE)
        {
            TickType_t xCurrentTick = xTaskGetTickCount();
            fiboncacci(40);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 1) Current time after execution: %d time to execute Fib: %d \n", xCurrentTick, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
            xSemaphoreGive(task1SyncSemaphore);
        }
    }
}


// Main function
int main(void)
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);

```

```c

    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, 230400, SYSTEM_CLOCK);



    task1SyncSemaphore = xSemaphoreCreateBinary();
    task2SyncSemaphore = xSemaphoreCreateBinary();


    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    xSemaphoreGive(task1SyncSemaphore);
    startTimeTick = xTaskGetTickCount();

    vTaskStartScheduler();

    return (0);
}


/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```

A.3 Q3

main.c

```c

#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"


// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"
#include "limits.h"


#define FIB_LIMIT_FOR_32_BIT 47
#define TIME_TO_RUN 240 //ms

SemaphoreHandle_t task1SyncSemaphore, task2SyncSemaphore;
TickType_t startTimeTick;
TaskHandle_t Task1_handle, Task2_handle;
uint32_t counter = 0;
double Hz = 100;
uint32_t ulPeriod;


void fiboncacci(int ms){
    TickType_t xStartTick = xTaskGetTickCount();
    TickType_t xCurrentTick = xTaskGetTickCount();
    uint32_t fib = 1, fib_a = 1, fib_b = 1;
    uint32_t i;
    while((xCurrentTick - xStartTick) < (pdMS_TO_TICKS(ms) -1)){
        for (i = 0; i < FIB_LIMIT_FOR_32_BIT; i++){
            if(((xCurrentTick - xStartTick) >= pdMS_TO_TICKS(ms)-1)) break;
            fib_a = fib_b;
            fib_b = fib;
            fib = fib_a + fib_b;
        }
        xCurrentTick = xTaskGetTickCount();
    }
}
```

```c
void Timer0Isr(void)
{
    TickType_t xCurrentTick = xTaskGetTickCount();
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);  // Clear the timer interrupt
    counter ++;
    if (counter % 3 == 0){
        xTaskNotifyFromISR(Task1_handle, xCurrentTick, eSetValueWithOverwrite, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
    else if(counter % 8 == 0){
        xTaskNotifyFromISR(Task2_handle, xCurrentTick, eSetValueWithOverwrite, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        counter = 0;
    }
}



// Process 1
void xTask1(void * pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
    BaseType_t xResult;
    uint32_t ulNotifiedValue;

    while((xLastWakeTime - startTimeTick) < TIME_TO_RUN){

        xResult = xTaskNotifyWait( pdFALSE,
                                /* Don't clear bits on entry. */
                                ULONG_MAX,
                                /* Clear all bits on exit. */
                                &ulNotifiedValue, /* Stores the notified value. */
                                xMaxBlockTime );

        if( xResult == pdPASS )
        {
//              xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 1 started at %d ms and Timer interrupt data: %d\n", xCurrentTick, ulNotifiedValue);
            fiboncacci(10);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 1 completed at %d ms (Execution: %d ms)\n",
                        xFibTime, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
//              xSemaphoreGive(task1SyncSemaphore);
        }
    }
```

```
}



// Process 2
void xTask2(void *pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
    BaseType_t xResult;
    uint32_t ulNotifiedValue;

    while ((xLastWakeTime - startTimeTick) < TIME_TO_RUN)
    {

        xResult = xTaskNotifyWait( pdFALSE,
                               /* Don't clear bits on entry. */
                               ULONG_MAX,
                               /* Clear all bits on exit. */
                               &ulNotifiedValue, /* Stores the notified value. */
                               xMaxBlockTime );

        if( xResult == pdPASS )
        {
//              xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 2 started at %d ms (Preempted Task 1) and  Timer
interrupt data %d\n", xCurrentTick, ulNotifiedValue);
            fiboncacci(40);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 2 completed at %d ms (Execution: %d ms)\n",
                       xFibTime, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
//              xSemaphoreGive(task1SyncSemaphore);
        }
    }
}



// Main function
int main(void)
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);


    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, 230400, SYSTEM_CLOCK);
```

```c

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);   // 32 bits Timer
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr);    // Registering  isr


    ulPeriod = (SYSTEM_CLOCK / Hz);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);

    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    task1SyncSemaphore = xSemaphoreCreateBinary();
    task2SyncSemaphore = xSemaphoreCreateBinary();


    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 2, &Task1_handle);
    xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, &Task2_handle);
    startTimeTick = xTaskGetTickCount();
    xTaskNotifyFromISR(Task1_handle, startTimeTick, eSetValueWithOverwrite, NULL);
    xTaskNotifyFromISR(Task2_handle, startTimeTick, eSetValueWithOverwrite, NULL);

    vTaskStartScheduler();

    return (0);
}


/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```