

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	1
List of Tables	1
1 Question 1	2
Answer	2
2 Question 2	5
Answer	5
3 Question 3	8
Answer	8
4 Reference	10
Appendices	11
A C Code for the Implementation	11
A.1 Q1	11
A.2 Q2	15
A.3 Q3	19

List of Figures

1 Semaphore has been posted by this program	3
2 Semaphore has been posted by this program	6
3 Semaphore has been posted by this program	9

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()`.

```

GTKTerm - /dev/ttyACM0 230400-8-N-1
File Edit Log Configuration Control signals View Help
Task 1 completed at 9 ms and Timer interrupt data: 9
Task 1 completed at 19 ms and Timer interrupt data: 19
Task 1 completed at 29 ms and Timer interrupt data: 29
Task 1 completed at 39 ms and Timer interrupt data: 39
Task 1 completed at 49 ms and Timer interrupt data: 49
Task 1 completed at 59 ms and Timer interrupt data: 59
Task 1 completed at 69 ms and Timer interrupt data: 69
Task 1 completed at 79 ms and Timer interrupt data: 79
Task 1 completed at 89 ms and Timer interrupt data: 89
Task 1 completed at 99 ms and Timer interrupt data: 99
Task 1 completed at 109 ms and Timer interrupt data: 109
Task 1 completed at 119 ms and Timer interrupt data: 119
Task 1 completed at 129 ms and Timer interrupt data: 129
Task 1 completed at 139 ms and Timer interrupt data: 139
Task 1 completed at 149 ms and Timer interrupt data: 149
Task 1 completed at 159 ms and Timer interrupt data: 159
Task 1 completed at 169 ms and Timer interrupt data: 169
Task 1 completed at 179 ms and Timer interrupt data: 179
Task 1 completed at 189 ms and Timer interrupt data: 189
Task 1 completed at 199 ms and Timer interrupt data: 199
Task 1 completed at 209 ms and Timer interrupt data: 209
Task 1 completed at 219 ms and Timer interrupt data: 219
Task 1 completed at 229 ms and Timer interrupt data: 229
Task 1 completed at 239 ms and Timer interrupt data: 239
Task 1 completed at 249 ms and Timer interrupt data: 249
Task 1 completed at 259 ms and Timer interrupt data: 259
/dev/ttyACM0 230400-8-N-1
DTR RTS CTS CD DSR RI

```

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.

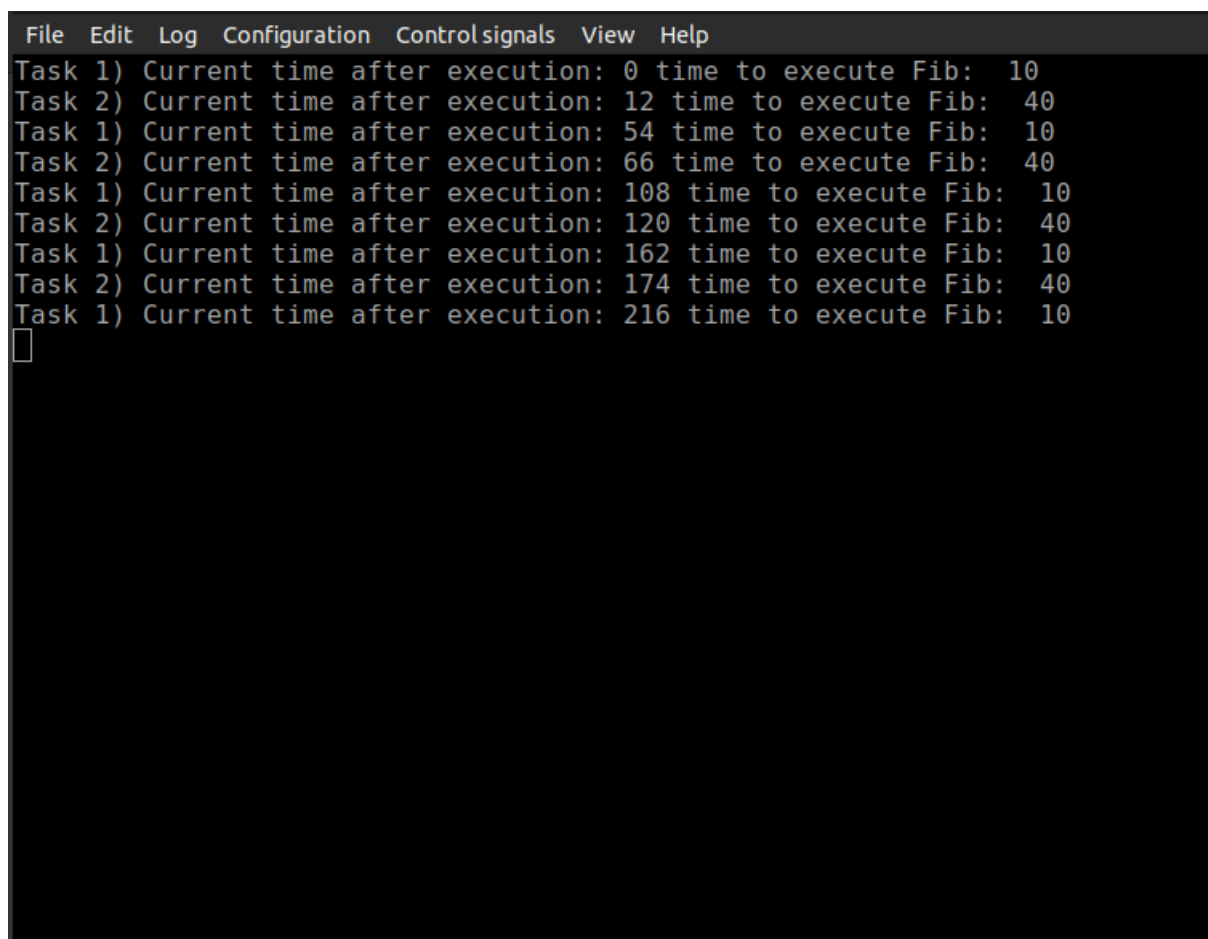
Detailed Code Analysis:

1. The `fibonacci()` 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 `fibonacci()` 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().



```

File Edit Log Configuration Control signals View Help
Task 1) Current time after execution: 0 time to execute Fib: 10
Task 2) Current time after execution: 12 time to execute Fib: 40
Task 1) Current time after execution: 54 time to execute Fib: 10
Task 2) Current time after execution: 66 time to execute Fib: 40
Task 1) Current time after execution: 108 time to execute Fib: 10
Task 2) Current time after execution: 120 time to execute Fib: 40
Task 1) Current time after execution: 162 time to execute Fib: 10
Task 2) Current time after execution: 174 time to execute Fib: 40
Task 1) Current time after execution: 216 time to execute Fib: 10

```

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 'fibonacci()' 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 'fibonacci()' 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 'fibonacci()' 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 2) and completes at 40 ms, with an execution duration of 11 ms.

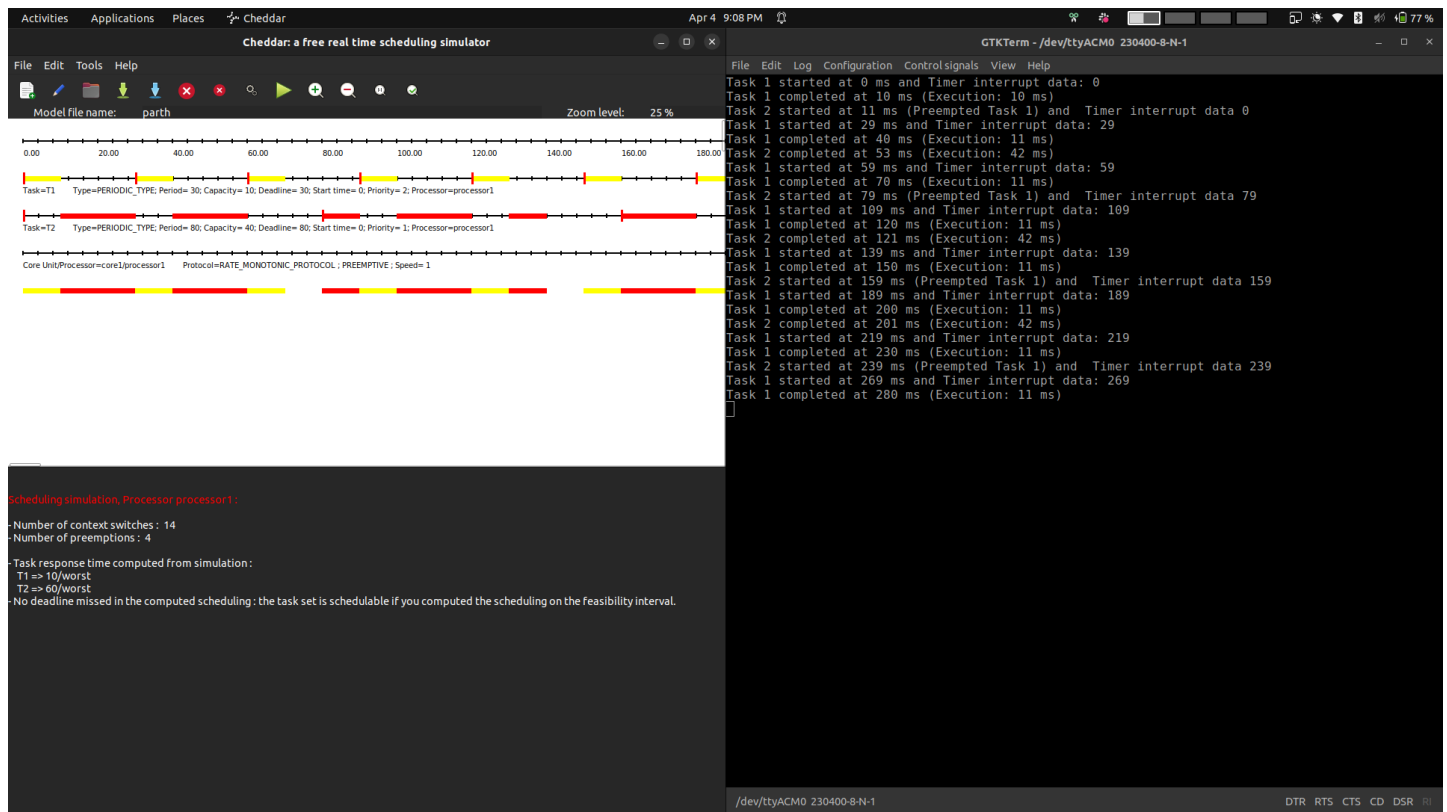


Figure 3: Semaphore has been posted by this program

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 Reference

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

```
10
11 #include <stdint.h>
12 #include <stdbool.h>
13 #include "main.h"
14 #include "drivers/pinout.h"
15 #include "utils/uartstdio.h"
16
17
18 // TivaWare includes
19 #include "driverlib/sysctl.h"
20 #include "driverlib/debug.h"
21 #include "driverlib/rom_map.h"
22 #include "driverlib/rom.h"
23 #include "driverlib/timer.h"
24 #include "driverlib/inc/hw_memmap.h"
25 #include "driverlib/inc/hw_ints.h"
26
27 // FreeRTOS includes
28 #include "FreeRTOSConfig.h"
29 #include "FreeRTOS.h"
30 #include <timers.h>
31 #include <semphr.h>
32 #include "task.h"
33 #include "queue.h"
34 #include "limits.h"
35
36
37 #define FIB_LIMIT_FOR_32_BIT 47
38 #define TIME_TO_RUN 240 //ms
39
40 SemaphoreHandle_t task1SyncSemaphore;
41 TaskHandle_t Task1_handle;
42 double Hz = 100;
43 uint32_t ulPeriod;
44
45
46
47 void Timer0Isr(void)
48 {
49     TickType_t xCurrentTick = xTaskGetTickCount();
50     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
51     ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
52
53     xTaskNotifyFromISR(Task1_handle, xCurrentTick, eSetValueWithOverwrite, &
xHigherPriorityTaskWoken);
54     portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
55
56 }
57
58
59
60 // Process 1
61 void xTask1(void * pvParameters)
```

```
62 {
63
64
65     const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
66     BaseType_t xResult;
67     uint32_t ulNotifiedValue;
68
69     while(1){
70
71         xResult = xTaskNotifyWait( pdFALSE,
72                                   /* Don't clear bits on entry. */
73                                   ULONG_MAX,
74                                   /* Clear all bits on exit. */
75                                   &ulNotifiedValue, /* Stores the notified value. */
76                                   xMaxBlockTime );
77
78         if( xResult == pdPASS )
79         {
80
81             TickType_t xCurrentTick = xTaskGetTickCount();
82             UARTprintf("Task 1 completed at %d ms and Timer interrupt data: %d\n",
83 xCurrentTick, ulNotifiedValue);
84
85         }
86     }
87
88
89
90 // Main function
91 int main(void)
92 {
93     // Initialize system clock to 120 MHz
94     uint32_t output_clock_rate_hz;
95     output_clock_rate_hz = ROM_SysCtlClockFreqSet(
96         (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
97          SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
98         SYSTEM_CLOCK);
99     ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
100
101
102     // Initialize the GPIO pins for the Launchpad
103     PinoutSet(false, false);
104     UARTStdioConfig(0, 230400, SYSTEM_CLOCK);
105
106     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
107     ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC); // 32 bits Timer
108     TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr); // Registering isr
109
110
111     ulPeriod = (SYSTEM_CLOCK / Hz);
112     ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);
113
114     ROM_TimerEnable(TIMER0_BASE, TIMER_A);
115     ROM_IntEnable(INT_TIMER0A);
116     ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
```

```
117
118     task1SyncSemaphore = xSemaphoreCreateBinary();
119
120
121     xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 2, &Task1_handle);
122
123     vTaskStartScheduler();
124
125     return (0);
126 }
127
128
129 /* ASSERT() Error function
130  *
131  * failed ASSERTS() from driverlib/debug.h are executed in this function
132  */
133 void __error__(char *pcFilename, uint32_t ui32Line)
134 {
135     // Place a breakpoint here to capture errors until logging routine is finished
136     while (1)
137     {
138     }
139 }
```

A.2 Q2

main.c

```
15
16 #include <stdint.h>
17 #include <stdbool.h>
18 #include "main.h"
19 #include "drivers/pinout.h"
20 #include "utils/uartstdio.h"
21
22
23 // TivaWare includes
24 #include "driverlib/sysctl.h"
25 #include "driverlib/debug.h"
26 #include "driverlib/rom_map.h"
27 #include "driverlib/rom.h"
28 #include "driverlib/timer.h"
29 #include "driverlib/inc/hw_memmap.h"
30 #include "driverlib/inc/hw_ints.h"
31
32 // FreeRTOS includes
33 #include "FreeRTOSConfig.h"
34 #include "FreeRTOS.h"
35 #include <timers.h>
36 #include <semphr.h>
37 #include "task.h"
38 #include "queue.h"
39
40
41 #define FIB_LIMIT_FOR_32_BIT 47
42 #define TIME_TO_RUN 200 //ms
43
44 unsigned long int ulPeriod;
45 unsigned int Hz = 1; // frequency in Hz
46
47 SemaphoreHandle_t task1SyncSemaphore, task2SyncSemaphore;
48 TickType_t startTimeTick;
49
50
51 void fiboncacci(int ms){
52     TickType_t xStartTick = xTaskGetTickCount();
53     TickType_t xCurrentTick = xTaskGetTickCount();
54     uint32_t fib = 1, fib_a = 1, fib_b = 1;
55     uint32_t i;
56     while((xCurrentTick - xStartTick) < pdMS_TO_TICKS(ms)){
57         for (i = 0; i < FIB_LIMIT_FOR_32_BIT; i++){
58             fib_a = fib_b;
59             fib_b = fib;
60             fib = fib_a + fib_b;
61         }
62         xCurrentTick = xTaskGetTickCount();
63     }
64
65
66 }
67
```

```

68
69 // Process 1
70 void xTask1(void * pvParameters)
71 {
72     TickType_t xLastWakeTime;
73     xLastWakeTime = xTaskGetTickCount();
74
75     while((xLastWakeTime - startTimeTick) < TIME_TO_RUN){
76         if (xSemaphoreTake(task1SyncSemaphore, portMAX_DELAY) == pdTRUE)
77         {
78             TickType_t xCurrentTick = xTaskGetTickCount();
79             fiboncacci(10);
80             TickType_t xFibTime = xTaskGetTickCount();
81             UARTprintf("Task 1) Current time after execution: %d time to execute Fib:
%d \n", xCurrentTick, (xFibTime - xCurrentTick));
82             xLastWakeTime = xCurrentTick;
83             xSemaphoreGive(task2SyncSemaphore);
84         }
85     }
86 }
87
88
89 // Process 2
90 void xTask2(void *pvParameters)
91 {
92     TickType_t xLastWakeTime;
93     xLastWakeTime = xTaskGetTickCount();
94
95     while ((xLastWakeTime - startTimeTick) < TIME_TO_RUN)
96     {
97
98
99         if (xSemaphoreTake(task2SyncSemaphore, portMAX_DELAY) == pdTRUE)
100         {
101             TickType_t xCurrentTick = xTaskGetTickCount();
102             fiboncacci(40);
103             TickType_t xFibTime = xTaskGetTickCount();
104             UARTprintf("Task 1) Current time after execution: %d time to execute Fib:
%d \n", xCurrentTick, (xFibTime - xCurrentTick));
105             xLastWakeTime = xCurrentTick;
106             xSemaphoreGive(task1SyncSemaphore);
107         }
108     }
109 }
110
111
112 // Main function
113 int main(void)
114 {
115     // Initialize system clock to 120 MHz
116     uint32_t output_clock_rate_hz;
117     output_clock_rate_hz = ROM_SysCtlClockFreqSet(
118         (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
119          SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
120         SYSTEM_CLOCK);
121     ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
122

```

```
123
124 // Initialize the GPIO pins for the Launchpad
125 PinoutSet(false, false);
126 UARTStdioConfig(0, 230400, SYSTEM_CLOCK);
127
128
129
130 task1SyncSemaphore = xSemaphoreCreateBinary();
131 task2SyncSemaphore = xSemaphoreCreateBinary();
132
133
134 xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
135 xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
136
137 xSemaphoreGive(task1SyncSemaphore);
138 startTimeTick = xTaskGetTickCount();
139
140 vTaskStartScheduler();
141
142 return (0);
143 }
144
145
146 /* ASSERT() Error function
147 *
148 * failed ASSERTS() from driverlib/debug.h are executed in this function
149 */
150 void __error__(char *pcFilename, uint32_t ui32Line)
151 {
152     // Place a breakpoint here to capture errors until logging routine is finished
153     while (1)
154     {
155     }
156 }
```

A.3 Q3

main.c

```
108
109 #include <stdint.h>
110 #include <stdbool.h>
111 #include "main.h"
112 #include "drivers/pinout.h"
113 #include "utils/uartstdio.h"
114
115
116 // TivaWare includes
117 #include "driverlib/sysctl.h"
118 #include "driverlib/debug.h"
119 #include "driverlib/rom_map.h"
120 #include "driverlib/rom.h"
121 #include "driverlib/timer.h"
122 #include "driverlib/inc/hw_memmap.h"
123 #include "driverlib/inc/hw_ints.h"
124
125 // FreeRTOS includes
126 #include "FreeRTOSConfig.h"
127 #include "FreeRTOS.h"
128 #include <timers.h>
129 #include <semphr.h>
130 #include "task.h"
131 #include "queue.h"
132 #include "limits.h"
133
134
135 #define FIB_LIMIT_FOR_32_BIT 47
136 #define TIME_TO_RUN 240 //ms
137
138 SemaphoreHandle_t task1SyncSemaphore, task2SyncSemaphore;
139 TickType_t startTimeTick;
140 TaskHandle_t Task1_handle, Task2_handle;
141 uint32_t counter = 0;
142 double Hz = 100;
143 uint32_t ulPeriod;
144
145
146 void fiboncacci(int ms){
147     TickType_t xStartTick = xTaskGetTickCount();
148     TickType_t xCurrentTick = xTaskGetTickCount();
149     uint32_t fib = 1, fib_a = 1, fib_b = 1;
150     uint32_t i;
151     while((xCurrentTick - xStartTick) < (pdMS_TO_TICKS(ms) - 1)){
152         for (i = 0; i < FIB_LIMIT_FOR_32_BIT; i++){
153             if((xCurrentTick - xStartTick) >= pdMS_TO_TICKS(ms) - 1) break;
154             fib_a = fib_b;
155             fib_b = fib;
156             fib = fib_a + fib_b;
157         }
158         xCurrentTick = xTaskGetTickCount();
159     }
160 }
```

```

161
162
163
164
165 void Timer0Isr(void)
166 {
167     TickType_t xCurrentTick = xTaskGetTickCount();
168     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
169     ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
170     counter ++;
171     if (counter % 3 == 0){
172         xTaskNotifyFromISR(Task1_handle, xCurrentTick, eSetValueWithOverwrite, &
xHigherPriorityTaskWoken);
173         portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
174     }
175     else if(counter % 8 == 0){
176         xTaskNotifyFromISR(Task2_handle, xCurrentTick, eSetValueWithOverwrite, &
xHigherPriorityTaskWoken);
177         portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
178         counter = 0;
179     }
180 }
181
182
183
184 // Process 1
185 void xTask1(void * pvParameters)
186 {
187     TickType_t xLastWakeTime;
188     xLastWakeTime = xTaskGetTickCount();
189     const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
190     BaseType_t xResult;
191     uint32_t ulNotifiedValue;
192
193     while((xLastWakeTime - startTimeTick) < TIME_TO_RUN){
194
195         xResult = xTaskNotifyWait( pdFALSE,
196                                     /* Don't clear bits on entry. */
197                                     ULONG_MAX,
198                                     /* Clear all bits on exit. */
199                                     &ulNotifiedValue, /* Stores the notified value. */
200                                     xMaxBlockTime );
201
202         if( xResult == pdPASS )
203         {
204             // xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
205             TickType_t xCurrentTick = xTaskGetTickCount();
206             UARTprintf("Task 1 started at %d ms and Timer interrupt data: %d\n",
xCurrentTick, ulNotifiedValue);
207             fiboncacci(10);
208             TickType_t xFibTime = xTaskGetTickCount();
209             UARTprintf("Task 1 completed at %d ms (Execution: %d ms)\n",
xFibTime, (xFibTime - xCurrentTick));
210             xLastWakeTime = xCurrentTick;
211             // xSemaphoreGive(task1SyncSemaphore);
212         }
213     }
214 }

```

```

215 }
216
217
218
219 // Process 2
220 void xTask2(void *pvParameters)
221 {
222     TickType_t xLastWakeTime;
223     xLastWakeTime = xTaskGetTickCount();
224     const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
225     BaseType_t xResult;
226     uint32_t ulNotifiedValue;
227
228     while ((xLastWakeTime - startTimeTick) < TIME_TO_RUN)
229     {
230
231         xResult = xTaskNotifyWait( pdFALSE,
232                                     /* Don't clear bits on entry. */
233                                     ULONG_MAX,
234                                     /* Clear all bits on exit. */
235                                     &ulNotifiedValue, /* Stores the notified value. */
236                                     xMaxBlockTime );
237
238         if( xResult == pdPASS )
239         {
240             // xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
241             TickType_t xCurrentTick = xTaskGetTickCount();
242             UARTprintf("Task 2 started at %d ms (Preempted Task 1) and Timer
interrupt data %d\n", xCurrentTick, ulNotifiedValue);
243             fibonacc(40);
244             TickType_t xFibTime = xTaskGetTickCount();
245             UARTprintf("Task 2 completed at %d ms (Execution: %d ms)\n",
246                         xFibTime, (xFibTime - xCurrentTick));
247             xLastWakeTime = xCurrentTick;
248             // xSemaphoreGive(task1SyncSemaphore);
249         }
250     }
251 }
252
253
254
255 // Main function
256 int main(void)
257 {
258     // Initialize system clock to 120 MHz
259     uint32_t output_clock_rate_hz;
260     output_clock_rate_hz = ROM_SysCtlClockFreqSet(
261                                     (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
262                                     SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
263                                     SYSTEM_CLOCK);
264     ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
265
266
267     // Initialize the GPIO pins for the Launchpad
268     PinoutSet(false, false);
269     UARTStdioConfig(0, 230400, SYSTEM_CLOCK);

```

```
270
271     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
272     ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);    // 32 bits Timer
273     TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr);    // Registering isr
274
275
276     ulPeriod = (SYSTEM_CLOCK / Hz);
277     ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 1);
278
279     ROM_TimerEnable(TIMER0_BASE, TIMER_A);
280     ROM_IntEnable(INT_TIMER0A);
281     ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
282
283     task1SyncSemaphore = xSemaphoreCreateBinary();
284     task2SyncSemaphore = xSemaphoreCreateBinary();
285
286
287     xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 2, &Task1_handle);
288     xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, &Task2_handle);
289     startTimeTick = xTaskGetTickCount();
290     xTaskNotifyFromISR(Task1_handle, startTimeTick, eSetValueWithOverwrite, NULL);
291     xTaskNotifyFromISR(Task2_handle, startTimeTick, eSetValueWithOverwrite, NULL);
292
293     vTaskStartScheduler();
294
295     return (0);
296 }
297
298
299 /* ASSERT() Error function
300 *
301 * failed ASSERTS() from driverlib/debug.h are executed in this function
302 */
303 void __error__(char *pcFilename, uint32_t ui32Line)
304 {
305     // Place a breakpoint here to capture errors until logging routine is finished
306     while (1)
307     {
308     }
309 }
```