**CG2271 Real Time Operating Systems**

**Lab 6 – Introduction to RTOS**

1. Introduction

   FreeRTOS is an industrial grade real-time operating system that runs on a very wide variety of microcontrollers. It is open-source, so many variants of FreeRTOS exist.

   A paid, professional version of FreeRTOS is available called SafeRTOS. which is certified to be used on critical applications like medical equipment and cars. SafeRTOS has more features for safety-critical applications, but is otherwise almost identical to FreeRTOS.

   In today's lab we will be looking at FreeRTOS and some of its features. We will in particular look at how to set up a FreeRTOS project in MCUXpresso, how to configure FreeRTOS, and how to write a FreeRTOS application.

2. Submissions

   Answer all questions in the enclosed answer book, and as always rename it to Lab6-LabGpBxxSubGpyy.docx, e.g. Lab6-LabGpB01SubGp10.docx. Submit the completed docx file on Canvas by Friday 17 October 2359 hours. This lab is worth 20 marks.

3. Running Periodic Tasks without an RTOS

   Many embedded applications require tasks to be performed in fixed periods. For example, it is common for GPS units to return readings at a 10 Hz frequency; i.e. one reading every 100 milliseconds (ms).

   Our application must then read the GPS every 100 ms and process the reading.

   In this part we will look at writing periodic tasks without an RTOS:

   i. Create a new C/C++ project called "lab6p1", then delete the contents of lab6p1.c.
   ii. Open the provided busywait.c and copy the contents over. This program will:
      a. Print "Hello!" on the Terminal every 300 ms.
      b. Toggle the red LED every 600 ms.
      c. Toggle the green LED every 900 ms.

d. Toggle the blue LED every 1200 ms.

iii. Compile the code and run the code

iv. Open the serial terminal to see the Hello! being printed every 300ms.

**Question 1.** (3 MARKS) The function performing all these tasks is called "manageTasks". Explain briefly how this function works.

Note that because we are using busy-waiting, the timings are not very accurate, but are representative of the periods we want.

**Question 2.** (2 MARKS) Modify the program so that:

i. We print "Hello!" every 210 ms.

ii. We toggle the red LED every 322 ms.

iii. We toggle the green LED every 333 ms.

iv. We toggle the blue LED every 425 ms.

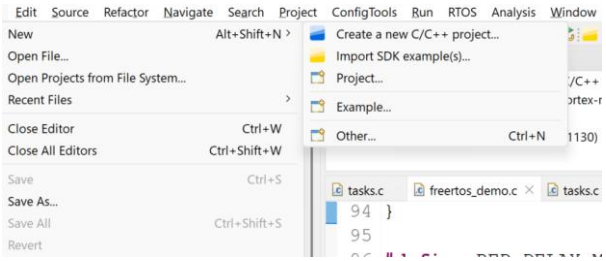
The minimum delay for delayMS is 5 ms. Do not use any delay below that. Describe how you modified the code to meet the requirements above.

(Also: Don't spend too much time on this question).

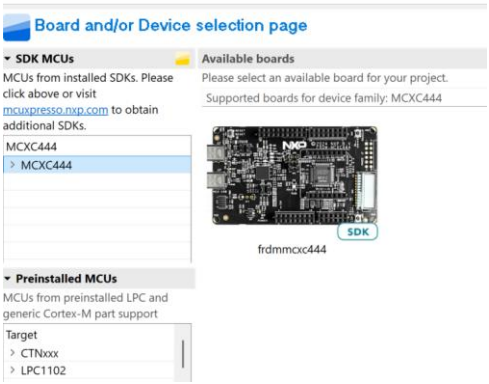
4. <u>Setting up a FreeRTOS Project</u>

We will now solve the same problem using FreeRTOS, and take this opportunity to learn how to set up a FreeRTOS project on MCUXpresso.

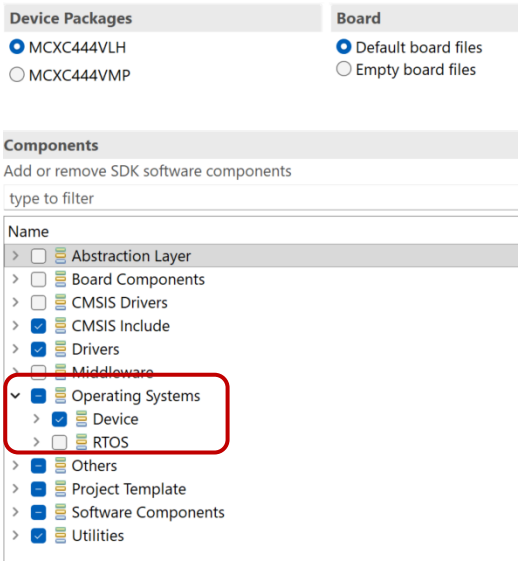
i. Click File->New->Create a new C/C++ project, as before.



ii. Select the MCXC444 CPU on the left panel and the FRDM-MCXC444 board on the right panel, and click Next.




iii. Name your project lab6p2, and go to the Components panel at the bottom of the screen, and click the “>” next to “Operating Systems”:

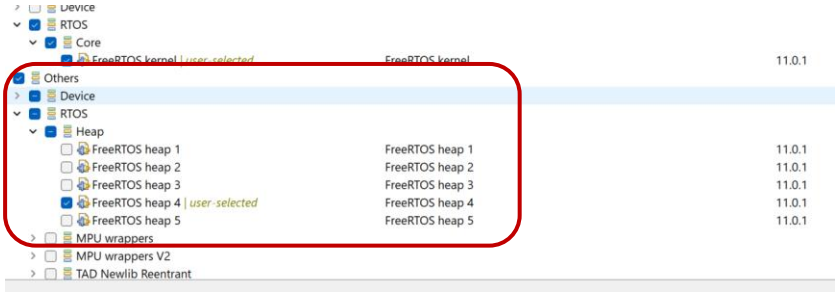



Click to select “RTOS”. This will automatically unselect “Device” as only one of the two options can be chosen in any one project.

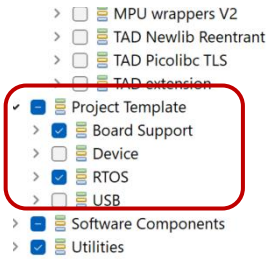
iv. **This step is very important – failure to follow this step will result in your program hanging.**

Now on the next line, click on the ">" next to Others to expand it, then expand the RTOS tree, then the Heap tree. The default heap is FreeRTOS heap 5.

**Unselect FreeRTOS heap 5, and select FreeRTOS heap 4.** Make sure that no other FreeRTOS heap is selected.




v. Now go down to "Project Template" and expand it. Ensure that RTOS is selected.




vi. Now click Next. As usual on the next screen ensure that Redirect printf/scanf to UART is selected, then click Finish to create the project.

vii. Open the lab6p2.c file, and remove its contents. Open the freertos.c file given to you, and copy its contents over to the empty lab6p2.c file.

Real-time operating systems like FreeRTOS are easily customizable. We will look at how to customize our FreeRTOS installation. Note that this has to be done once for every project, so it might help for you to create a standard customization and just paste it in.

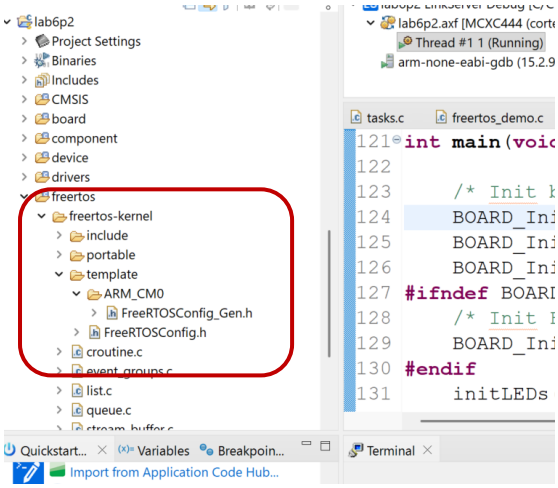
To customize our FreeRTOS:

i. In the Project Explorer window on the left, click on the > next to lab6p2 to expand its tree, if its not already expanded.

ii. Expand the tree for freertos.

iii. Under freertos, expand the tree for freertos-kernel.

iv. Now expand the tree for template, then ARM CM0. Your Project Explorer should now look like this:




v. Double-click on FreeRTOSConfig.Gen.h to open this header file. You will see many parameters that you can use to customize FreeRTOS:

```
#define configUSE_PREEMPTION                    1
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      (SystemCoreClock)
#define configTICK_RATE_HZ                      ((TickType_t)200)
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                ((unsigned short)90)
#define configMAX_TASK_NAME_LEN                 20
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       1
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               8
```

To see a full explanation of these parameters, please refer to Chapter 7 of the FreeRTOS Reference Manual which can be found at https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf

One interesting aspect of FreeRTOS (and many RTOS) is that you can select what components to include and what to leave out. If you scroll down further in FreeRTOSConfig_Gen.h, you will see:

```
/* Optional functions - most linkers will remove unused functions anyway. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTaskGetCurrentTaskHandle       1
#define INCLUDE_uxTaskGetStackHighWaterMark     0
#define INCLUDE_xTaskGetIdleTaskHandle          0
#define INCLUDE_eTaskGetState                   0
#define INCLUDE_xTimerPendFunctionCall          1
#define INCLUDE_xTaskAbortDelay                 0
#define INCLUDE_xTaskGetHandle                  0
#define INCLUDE_xTaskResumeFromISR              1
```

If you want to remove a function, set its value to 0. To include, set it to 1.

For CG2271, we will modify one parameter:

- Look for configUSE_TIME_SLICING and change the value from 0 to 1. By default in FreeRTOS a task will continue to run until a higher priority task is ready and pre-empts it (if configUSE_PREEMPTION is set to 1) or the current task gives us CPU using vTaskDelay or other such calls. Setting configUSE_TIME_SLICING to 1 turns on time slicing so that the CPU is handed over to other tasks even if the current tasks is still running.

Your file should now look like this:

```
#define configUSE_PREEMPTION                    1
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      (SystemCoreClock)
#define configTICK_RATE_HZ                      ((TickType_t)200)
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                ((unsigned short)180) // Was 90
#define configMAX_TASK_NAME_LEN                 20
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       1
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               8
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  1 // Was 0
#define configUSE_NEWLIB_REENTRANT              0
```

We can now build and debug our program as usual. Note that the LEDs are now flashing at a similar rate to our earlier program. Please refer to the Lab Lecture to see how this program was written, and answer the question below.

**Question 3.** (1 MARK) Modify lab6p2.c so that:

i. We print "Hello!" every 210 ms.
ii. We toggle the red LED every 322 ms.
iii. We toggle the green LED every 333 ms.
iv. We toggle the blue LED every 425 ms.

Describe how you modified the code to meet the requirements above. Did you have an easier time modifying lab6p2.c than lab6p1.c? Explain briefly why.

5. Exploring Race Conditions and Mutexes in FreeRTOS

I. Create a new C/C++ project, calling it "cg2271race". Ensure that you set it up as a FreeRTOS project (see above). Check that you have:
   a. Selected the RTOS Operating System.
   b. Selected the freertos_heap4 heap (important!). Ensure that no other heap has been selected.
   c. Selected the RTOS Project Template.
   d. At this point DO NOT change configUSE_TIME_SLICING. Leave it as 0.x`
II. Open the cg2271race.c file and delete all its contents.
III. Open the provided race.c file and copy its contents over to cg2271race.c
IV. Build your program and click Debug.
V. Open the UART terminal, click Run->Continue, and answer the following question:

**Question 4a.** (1 MARK) What is the expected result of sum? What is the actual value

Now open FreeRTOSConfig_Gen.h, and change configUSE_TIME_SLICING to 1. Terminate the current program, build, and debug again. Open the UART terminal, then click Run->Continue.

**Question 4b.** (2 MARKS) What is the actual value of sum now? Explain why you get correct results when time slicing is turned off, and the wrong results when it is turned on.

**Question 5.** (2 MARKS) What is the advantage of using time-slicing?

We will now solve our race condition by using mutexes. In FreeRTOS, a mutex is a binary semaphore whose starting value is 1, which is also its maximum value. Mutexes work like this:

i. The first task that executes xSemaphoreTake on the mutex will decrement it to 0.
ii. A second task executing xSemaphoreTake on the mutex that is now 0 will block.

iii. When the first task executes xSemaphoreGive, it will unblock the second task.

iv. When the second task executes xSemaphoreGive, the mutex will increment back to 1.

We will now see how to use mutexes to prevent race conditions.

At the top of your cg2271race.c file just after #include "task.h", add in the following #include to bring in the relevant header file.

```
#include "semphr.h"
```

i. Just above addOneTask, add in this mutex declaration:

```
SemaphoreHandle_t mutex;
```

ii. Now go to main, and create the mutex by adding this line BEFORE the loop that creates the tasks:

```
mutex = xSemaphoreCreateMutex();
```

iii. Go to the addOneTask task, and add `if(xSemaphoreTake(mutex, portMAX_DELAY)==pdTRUE)` before the sum++, and `xSemaphoreGive(mutex);` Your code should look like this. Note that sum++ and xSemaphoreGive(mutex) occur within the if block (circled):

```
static void addOneTask(void *p) {
    for(int i=0; i<COUNT; i++) {
        if(xSemaphoreTake(mutex, portMAX_DELAY) == pdTRUE) {
            sum++;
            xSemaphoreGive(mutex);
        }
    }
    vTaskSuspend(NULL);
}
```

This is important because xSemaphoreTake will only block until portMAX_DELAY. While this is a very large number ($2^{32} - 1$ on the MCXC444), it can still timeout, in which case xSemahporeTake will return pdFALSE. So we much check that xSemaphoreTake returns pdTRUE to ensure that it didn't timeout.

iv. Now build your project, click Debug, open the Terminal window, click Run->Continue, and answer this question:

**Question 6.** (2 MARKS)

Is the answer now correct? Explain why.

6. Using Semaphores

Currently our printTask calls vTaskDelay to wait for 10 seconds before printing the value of sum:

```
static void printTask(void *p) {
    while(1) {
        // Delay before printing
        PRINTF("Waiting ten seconds before printing results\r\n");
        vTaskDelay(pdMS_TO_TICKS(10000));
        PRINTF("Expected result: %d Actual: %d\r\n",
                NUM_TASKS * COUNT, sum);
         vTaskSuspend(NULL);
    }
}
```

We will now modify our program to use binary semaphores so that:

i. printTask will no longer wait for 10 seconds before printing the result.
ii. printTask will now print the sub-total each time a task finishes updating sum.

We will use binary semaphores to do this. In FreeRTOS a binary semaphore is similar to a mutex except that its starting value is 0 instead of 1. Just like a mutex, the maximum value of a binary semaphore is 1. **Note: The starting value of a binary semaphore can be set in most operating systems, but not in FreeRTOS. In FreeRTOS the starting value is always 0.**

Binary semaphores are used to coordinate between tasks in this way:

i. Task A calls xSemaphoreTake on the binary semaphore. If no task has called xSemaphoreGive yet, task A will block.
ii. Task B calls xSemaphoreGive, causing A to unblock.

Note that if task B had called xSemaphoreGive first, this will increment the semaphore to 1, and task A will not block when it calls xSemaphoreTake.

Let's now modify our cg2271race.c file:

i. We need to #include semphr.h, which we did so earlier.
ii. Create a new semaphore called "coord". Just under our mutex semaphore, add this line:

```
SemaphoreHandle_t coord;
```

iii. We want printTask to wait on the coord sema:

a. Delete the line that says PRINTF("Waiting ten seconds ...
b. Delete the line that says vTaskDelay(pdMS_TO_TICKS(10000));
c. Add this line:

```
if(xSemaphoreTake(coord, portMAX_DELAY) == pdTRUE) {
}
```

d. Move the PRINTF("Expected result: ... line to within the if(xSemaphoreTake(... if block.
e. Delete the vTaskSuspend(NULL); line.

Your final printTask should look like this. Notice how the PRINTF is inside the xSemaphoreTake's if block:

```
static void printTask(void *p) {
    while(1) {
        // Delay before printing
        if(xSemaphoreTake(coord, portMAX_DELAY) == pdTRUE) {
            PRINTF("Expected result: %d Actual: %d\r\n", NUM_TASKS * COUNT, sum);
        }
    }
}
```

Again this is important to ensure that xSemaphoreTake did not timeout.

iv. We will now modify addOneTask to call xSemaphoreGive once it finishes adding to sum. Go to your addOneTask, and immediately after the for loop and before the vTaskSuspend(NULL), add this line:

```
xSemaphoreGive(coord);
```

Your addOneTask should now look like this:

```
static void addOneTask(void *p) {
    for(int i=0; i<COUNT; i++) {
        if(xSemaphoreTake(mutex, portMAX_DELAY) == pdTRUE) {
            sum++;
            xSemaphoreGive(mutex);
        }
    }

    xSemaphoreGive(coord);
    vTaskSuspend(NULL);
}
```

v. Finally modify main to create the semaphore. Immediately after the call to create the mutex (mutex = xSemaphoreCreateMutex(); ) and BEFORE the for loop to create the tasks, add in the following line:

```
coord = xSemaphoreCreateBinary();
```

Your main should now look like this:

```
mutex = xSemaphoreCreateMutex();
coord = xSemaphoreCreateBinary();

for(int i=0; i<NUM_TASKS; i++) {
    sprintf(taskName, "addone_%d", i);
    xTaskCreate(addOneTask, taskName, configMINIMAL_STACK_SIZE+100, NULL, 2, NULL);
}
```

Build the code, click Debug, then open the UART Terminal, and click Run->Continue. You will now see printTask print at the end of each task:

```
Race Demo
Expected result: 500000 Actual: 392181
Expected result: 500000 Actual: 468298
Expected result: 500000 Actual: 478654
Expected result: 500000 Actual: 492610
Expected result: 500000 Actual: 500000
```

7. Using Semaphores in ISRs

We very often want a task to wait until an interrupt has been triggered. Coming back to our GPS example at the start of this lab, we want to make navigation calculations only when there is a GPS reading. One way of doing this is to have a variable that is set to true when a reading has come in, and set back to false when the reading has been processed.

You are provided with a file called isr_poll.c, which has SW3 (the switch at the bottom of the board closest to the LCD) set up to trigger an interrupt when it is pressed. Every 5 presses of this button will toggle the red LED. Independently it will also print "Hello!" to the UART terminal every 500 ms.

i. Create a new FreeRTOS project called "pollisr". Ensure that:
   a. You choose RTOS as the Operating System.
   b. Under Others->RTOS->Heap you choose FreeRTOS heap 4 ONLY (none of the other heaps should be chosen).
   c. Ensure that Project Template->RTOS is selected.
   d. DO NOT set configUSE_TIME_SLICING to 1. Leave it at its default value of 0.

ii. Delete the contents of pollisr.c, and paste the contents of isr_poll.c in.

iii. Build and Debug your project. Open the UART terminal, then click Run->Resume to resume execution.

iv. Press the button. You should see "ISR triggered" followed by the current count value. The red LED will toggle every 5 presses.

Notice that the "Hello!" message never appears.

If we examine blinkLEDTask, we immediately see a problem:

```
static void blinkLEDTask(void *p) {
    PRINTF("BlinkLED Task Started\r\n");
    while(1) {
        if(blink) {
            // Toggle the RED LED
            toggleLED(RED);
            blink = 0;
        }
    }
}
```

The issue is that blinkLEDTask and helloTask have the same priority level (priority 1), and because blinkLEDTask never gives up control of the CPU, helloTask never gets to run.

**Question 7.** (2 MARKS) Would setting configUSE_TIME_SLICING to 1 help? Why or why not?

**Question 8.** (1 MARK) Change helloTask's priority to 2 by locating this line:

```
// Create the hello task
xTaskCreate(helloTask, "hello",
        configMINIMAL_STACK_SIZE+100, NULL, 1, NULL);
```

And changing the priority from 1 to 2:

```
// Create the hello task
xTaskCreate(helloTask, "hello",
        configMINIMAL_STACK_SIZE+100, NULL, 2, NULL);
```

Does helloTask run now? Explain why or why not.

**Change helloTask's priority back to 1**. This will now prevent helloTask from running again.

Add "else vTaskDelay(pdMS_TO_TICKS(250));" to the if statement in blinkLEDTask, so that blinkLEDTask now looks like this:

```
static void blinkLEDTask(void *p) {
    PRINTF("BlinkLED Task Started\r\n");
    while(1) {
        if(blink) {
            // Toggle the RED LED
            toggleLED(RED);
            blink = 0;
        }
        else
            vTaskDelay(pdMS_TO_TICKS(250));
    }
}
```

Build your code and verify that it works correctly now.

**Question 9.** (2 MARKS)

Why does adding the vTaskDelay to blinkLEDTask allow helloTask to run?

Let's now use semaphores instead of polling:

i. Immediately after #include "task.h", add this line to include the header files for semaphores.

```
#include "semphr.h"
```

ii. Create a semaphore called sema just above PORTA_IRQHandler:

```
SemaphoreHandle_t sema;
```

iii. In PORTA_IRQHandler we have this line that sets blink to 1 when count is a multiple of 5:

```
if(PORTA->ISFR & (1 << SW_PIN)) {
    if((count % 5) == 0) {
        blink = 1;
    }
}
```

Within blinkLEDTask we have:

```
while(1) {
    if(blink) {
        // Toggle the RED LED
        toggleLED(RED);
        blink = 0;
    }
    else
        xTaskDelay(pdMS_TO_TICKS(250));
}
```

This task tests if blink is 1, and if it is, it toggles the red LED. Otherwise it waits for 250 ms and tries again.

We need to use sema in place of blink to coordinate between PORTA_IRQHandler and blinkLEDTask. We modify PORtA_IRQHandler using the following steps:

i. Move the line that clears PORTA->ISFR to just above the if((count % 5) == 0) line. This because we will trigger a context switch and the ISFR bit may never get cleared if it remained in its original place.

ii. Replace the blink = 1 with these lines:

```
BaseType_t hpw = pdFALSE;
xSemaphoreGiveFromISR(sema, &hpw);
portYIELD_FROM_ISR(hpw);
```

Notice that in an ISR we must call xSemaphoreGiveFromISR instead of xSemaphoreGive. We declare a variable of type BaseType_t and set it to pd_FALSE.

xSemaphoreGiveFromISR takes two arguments; the semaphore itself, and hpw. The ISR will unblock any task taking the semaphore, and if that task has a higher priority that the task that was interrupted, hpw will be set to pdTRUE.

We then call portYIELD_FROM_ISR with hpw to let FreeRTOS decide if it should perform a context switch to the higher priority task (hpw == pdTRUE) or not (hpw == pdFALSE).

Our full PORTA_IRQHandler will now look like this with the changes circled.

```
void PORTA_IRQHandler() {
    static int count=0;

    count++;
    PRINTF("ISR triggered. count = %d\r\n", count);
    NVIC_ClearPendingIRQ(PORTA_IRQn);

    if(PORTA->ISFR & (1 << SW_PIN)) {
        PORTA->ISFR |= (1 << SW_PIN);
        if((count % 5) == 0) {
            BaseType_t hpw = pdFALSE;
            xSemaphoreGiveFromISR(sema, &hpw);
            portYIELD_FROM_ISR(hpw);
        }
    }
}
```

We now modify blinkLEDTask to wait on the semaphore instead of on the blink variable.

i. Locate this block in blinkLEDTask:

```
if(blink) {
    // Toggle the RED LED
    toggleLED(RED);
    blink = 0;
}
else
    xTaskDelay(pdMS_TO_TICKS(250));
```

ii. Replace it with this:

```
if(xSemaphoreTake(sema, portMAX_DELAY) == pdTRUE)
    toggleLED(RED);
```

Now blinkLEDTask will wait on sema until the ISR gives, or until the xSemaphoreTake times out. So we must check that this function returns pdTRUE before we toggle the LED to ensure that it did not timeout.

Finally we need to create the ISR.

i. Go to main and locate the line where we set blink to 0.
ii. Delete this line and replace it with:

```
sema = xSemaphoreCreateBinary();
```

Now build your application, click Debug, then click Run->Resume. Your red LED will toggle every 5 times that you press SW3.

**Question 10.** (2 MARKS) Name one advantage that using semaphores has over our earlier solution of using the global variable blink.

8. Lab Demo

There is no demo for this lab.