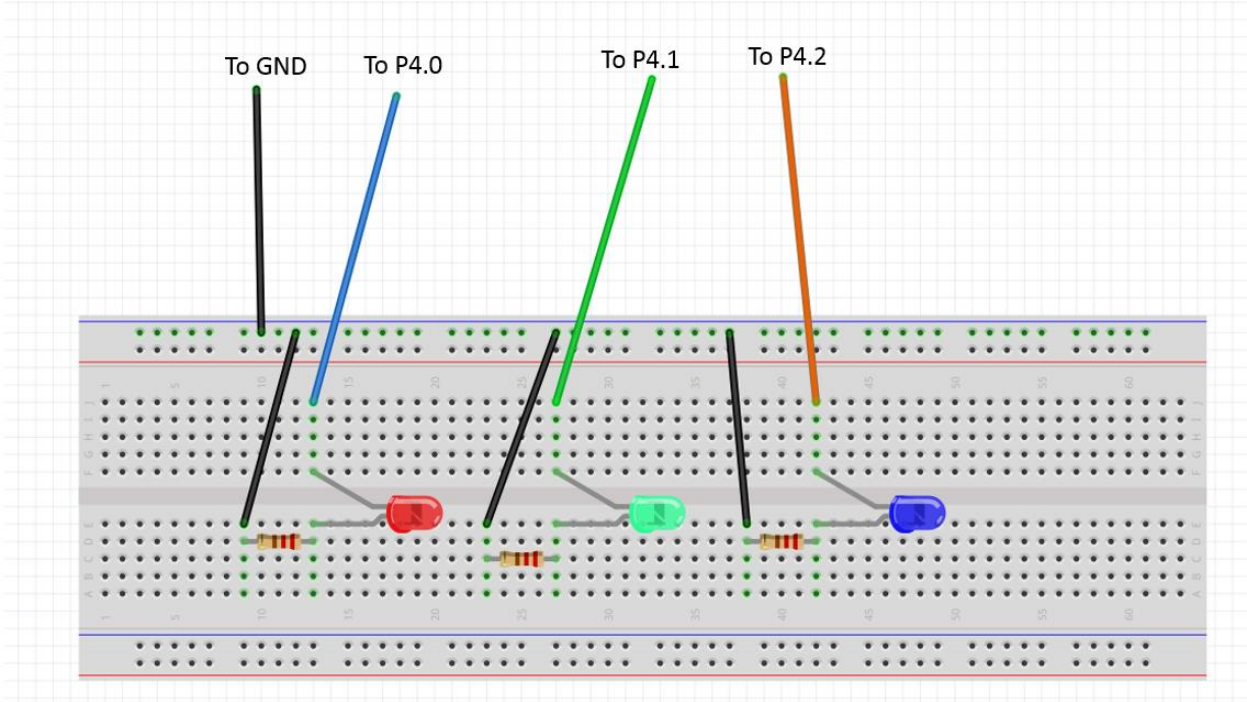
Lab Guide: FreeRTOS Mutex MSP432

Introduction:

Another important feature of most RTOS implementations is a mutex – mutual exclusion – system, a way to coordinate time between tasks/threads/processes, and also a way to protect shared resources like communication channels, printers, etc.

Constructing the Hardware

You'll use the same hardware for this lab as you used for the last lab.



Protecting Resources with Mutex/Semaphores

Mutual exclusion and semaphores are used for several purposes, but the two most important are synchronization and protection of shared resources. Let's look at the second case first, protecting a resource using a mutex. Let's start with the same code you used in the last lab. Then we'll add the code for this example (this code is on i-learn under the name FreeRTOSMSP432Mutex.txt):

```
/* Standard includes. */
#include <stdio.h>

/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
```

You'll add this library so that we can use semaphores

```
#include "semphr.h"

/* TI includes. */
#include "gpio.h"

static void vTaskFunction1( void *pvParameters );
static void vTaskFunction2( void *pvParameters );

static void prvSetupHardware( void );
```

You'll want to declare a global semaphore

```
SemaphoreHandle_t xMutex;

int main( void )
{
   /* See http://www.FreeRTOS.org/TI_MSP432_Free_RTOS_Demo.html for instructions. */

  /* Prepare the hardware to run this demo. */
   prvSetupHardware();
```

This actually fills in the data structures for your Mutex

```
   xMutex = xSemaphoreCreateMutex();
```

You'll use a random to trigger each of your two processes. This seeds the random number

```
   srand( 567 );
```

I didn't have to do it this way, but this just checks to see if the Mutex was created correctly.

```
    if( xMutex != NULL )
    {
       xTaskCreate( vTaskFunction1, "Task 1", 200, NULL, 1, NULL );
       xTaskCreate( vTaskFunction2, "Task 2", 200, NULL, 2, NULL );
       /* Start the scheduler. */
       vTaskStartScheduler();
    }

   /* If all is well, the scheduler will now be running, and the following
   line will never be reached.  If the following line does execute, then
   there was insufficient FreeRTOS heap memory available for the Idle and/or
   timer tasks to be created.  See the memory management section on the
   FreeRTOS web site for more details on the FreeRTOS heap
   http://www.freertos.org/a00111.html. */
   for( ;; );

            return 0;
}
/*-----------------------------------------------------------*/

void vTaskFunction1( void *pvParameters )
{
```

```
  /* As per most tasks, this task is implemented in an infinite loop. */
 for( ;; )
 {
```
Before accessing the LED, take the semaphore. If is not available wait for portMAX_DELAY time until it is ready.

```
    xSemaphoreTake( xMutex, portMAX_DELAY );
      GPIO_toggleOutputOnPin(
                GPIO_PORT_P4,
                GPIO_PIN0
                );
```
After toggling the LED, give back the Mutex. Now the other process can access the LED

```
   xSemaphoreGive( xMutex );
   /* Delay for a period.  This time we use a call to vTaskDelay() which
   puts the task into the Blocked state until the delay period has expired.
   The delay period is random */
   vTaskDelay( ( rand() & 0x1FF ) );
 }
}

void vTaskFunction2( void *pvParameters )
{


  /* As per most tasks, this task is implemented in an infinite loop. */
  for( ;; )
  {
```
Before accessing the LED, take the semaphore. If is not available wait for portMAX_DELAY time until it is ready.

```
     xSemaphoreTake( xMutex, portMAX_DELAY );
          GPIO_toggleOutputOnPin(
                    GPIO_PORT_P4,
                    GPIO_PIN0
                    );
```
After toggling the LED, give back the Mutex. Now the other process can access the LED

```
        xSemaphoreGive( xMutex );

   /* Delay for a period.  This time we use a call to vTaskDelay() which
   puts the task into the Blocked state until the delay period has expired.
   The delay period is random. */
   vTaskDelay( ( rand() & 0x1FF ) );
  }
}
```

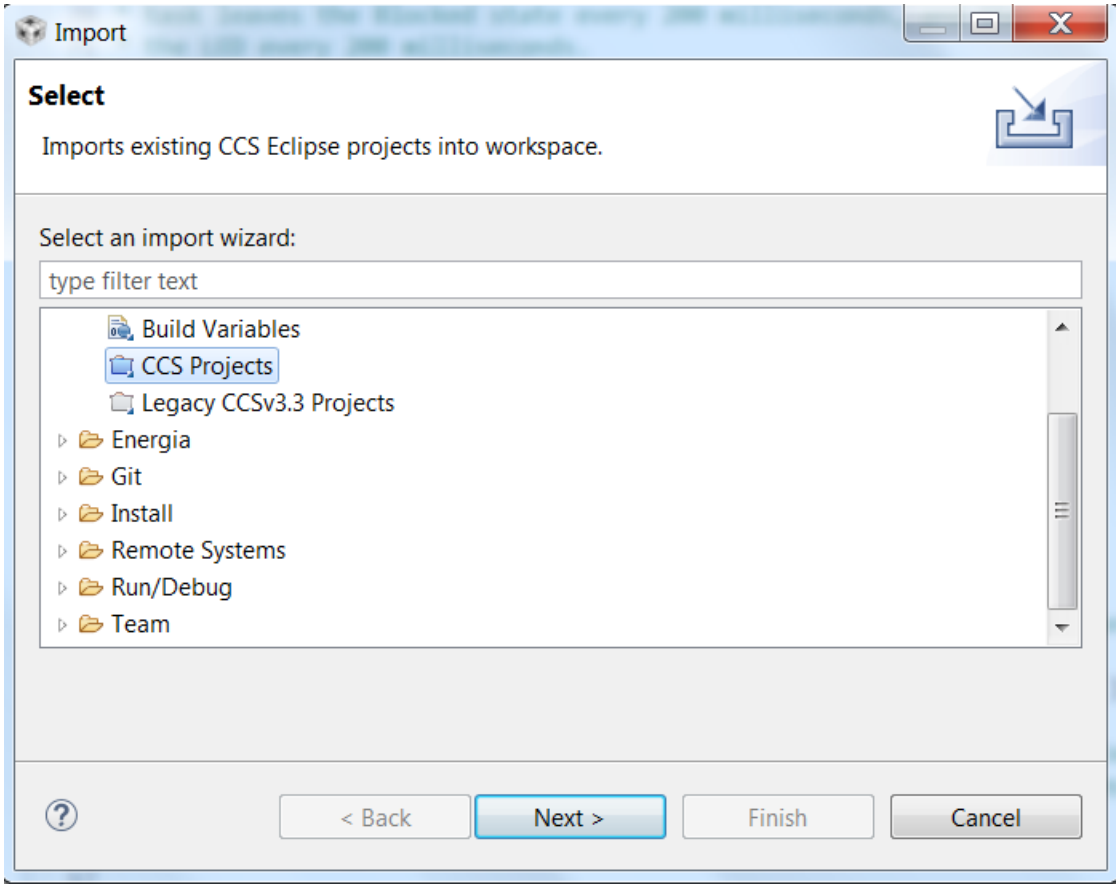
Lots of stuff for FreeRTOS

When you run this code you will see the LED connected to pin 4.0 flash at random intervals. The important aspect here is that neither process will get in the middle of toggling the LED and then be interrupted by the other process.

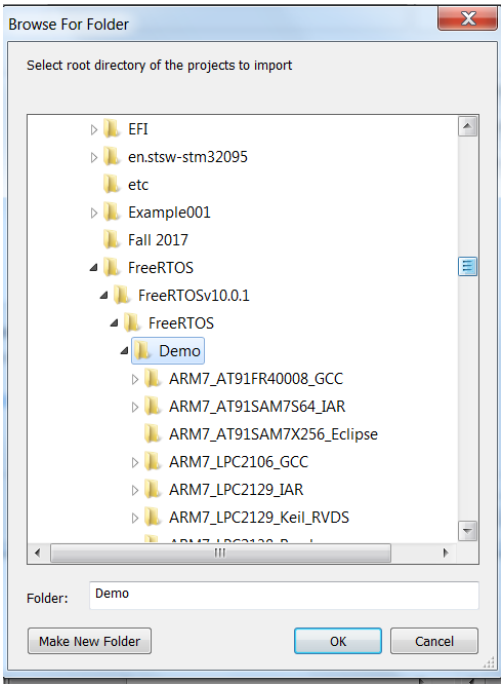
Synchronizing activities with Mutex/Semaphores

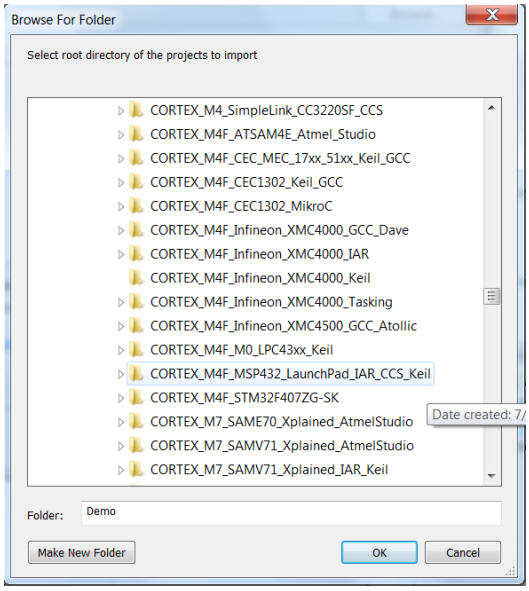
Now let's see an example where you will use Mutex to synchronize tasks. Let's start with the code from the RTOSDemo project. You may want to delete the project and re-import it. To do that follow these instructions:

To do this select Import->

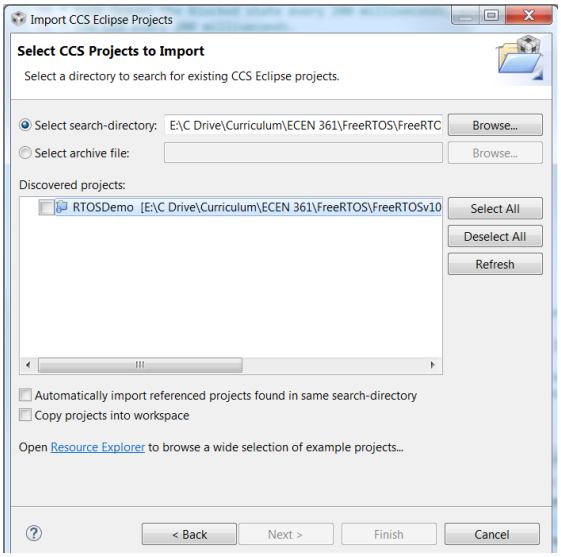


Hit Next. Now select the Browse button by the Select search-directory. Go to FreeRTOS/FreeRTOSv10.0.0.1/FreeRTOS/DEMO/CORTEX_M4F_MSP432_LaunchPad_IAR_CCS_KEIL, like this

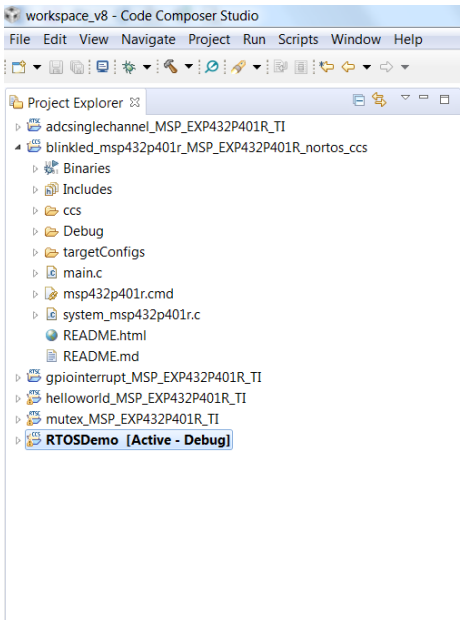




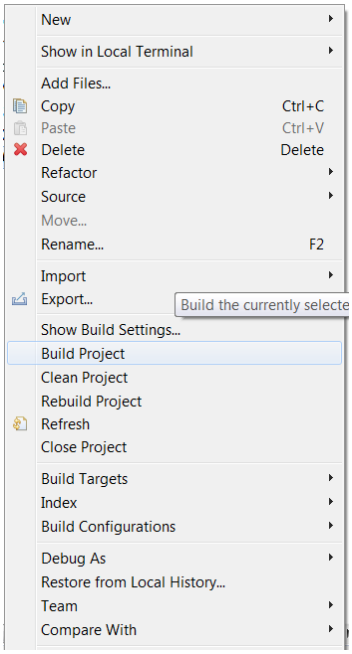
Then click OK.
Now you should see this selection:



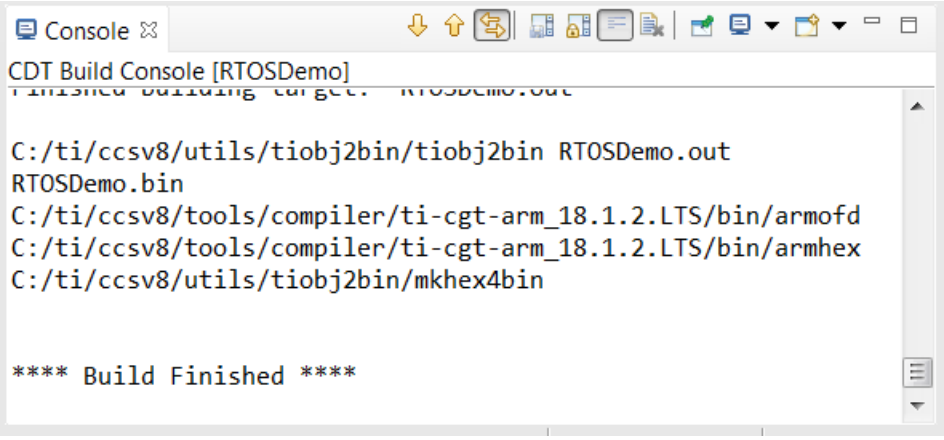
Make sure the RTOSDemo is selected, click both the two boxes at the bottom, then click Finish, and the Project should appear in your projects folder.



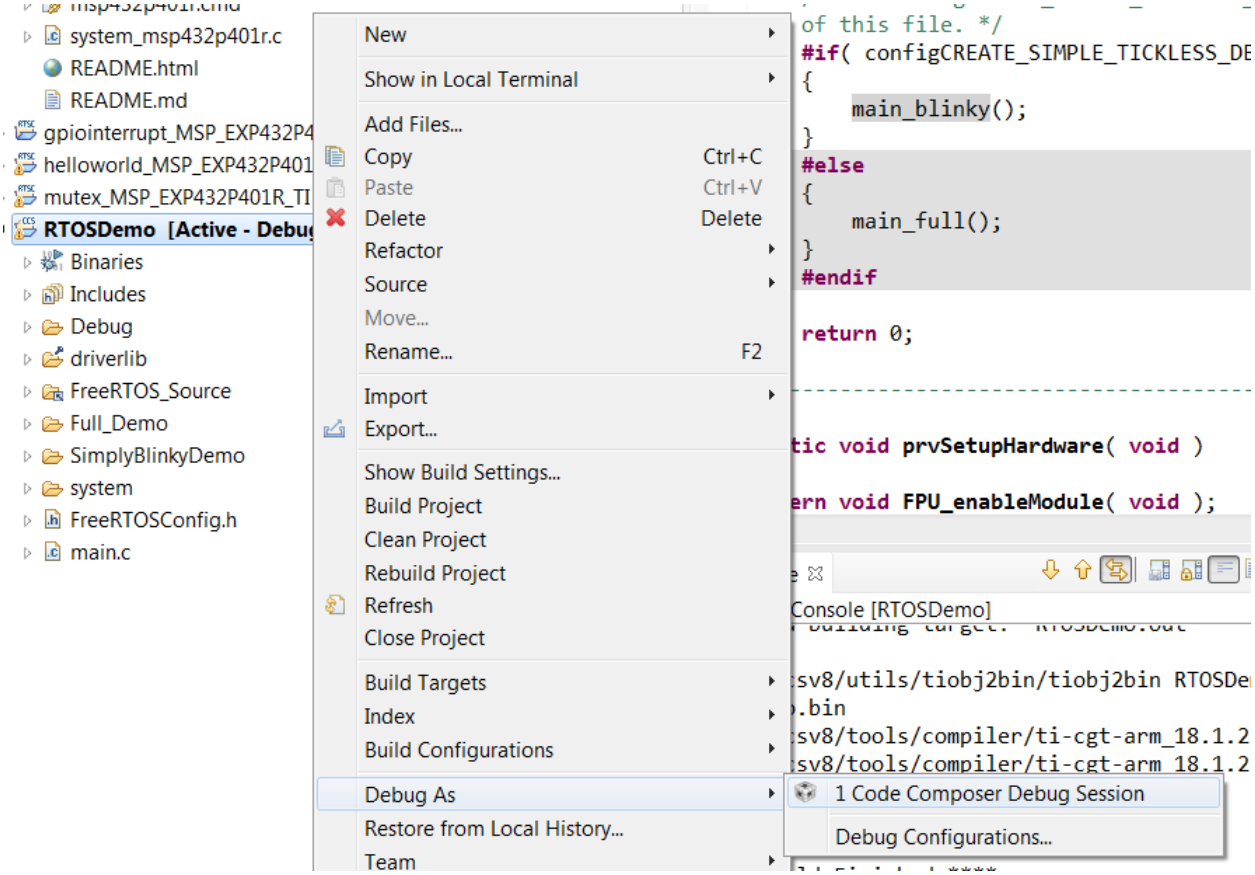
To make sure it installed correctly, go head and Build the project by right clicking on the project, then selecting Build Project:



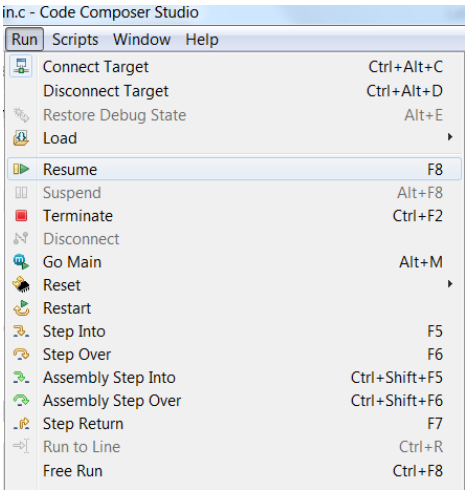
In the Build Console window you should see that the code has built.



To run the code you can right select the project, then select Debug as, then select Code Composer Debug Session.



If everything worked as it should, your code will now be downloaded to your processor. To enable it to run, click Run -> Resume.



A Red LED should be blinking on your MSP432 board. Now let's go to the main_blinky.c file in the SimplyBlinkyDemo file. You'll replace that file with the code on i-learn as FreeRTOSMSP432MutexSync.txt. Here is a description of that code:

```
/* Standard includes. */
#include <stdio.h>

/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"

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

/* NOTE: If an IAR build results in an undefined "reference to __write" linker
error then set the printf formatter project option to "tiny" and the scanf
formatter project option to "small". */

/*
 * Set up the hardware ready to run this demo.
 */
static void prvSetupHardware( void );

/*
 * main_blinky() is used when configCREATE_SIMPLE_TICKLESS_DEMO is set to 1.
 * main_full() is used when configCREATE_SIMPLE_TICKLESS_DEMO is set to 0.
 */
extern void main_blinky( void );
extern void main_full( void );

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

int main( void )
{
	/* See http://www.FreeRTOS.org/TI_MSP432_Free_RTOS_Demo.html for instructions. */

	/* Prepare the hardware to run this demo. */
	prvSetupHardware();

	/* The configCREATE_SIMPLE_TICKLESS_DEMO setting is described at the top
	of this file. */
	#if( configCREATE_SIMPLE_TICKLESS_DEMO == 1 )
	{
		main_blinky();
	}
	#else
	{
		main_full();
	}
	#endif

	return 0;
}
/*-----------------------------------------------------------*/

static void prvSetupHardware( void )
{
extern void FPU_enableModule( void );

	/* The clocks are not configured here, but inside main_full() and
	main_blinky() as the full demo uses a fast clock and the blinky demo uses
```

```
	a slow clock. */

	/* Stop the watchdog timer. */
	MAP_WDT_A_holdTimer();

	/* Ensure the FPU is enabled. */
	FPU_enableModule();

	/* Selecting P1.2 and P1.3 in UART mode and P1.0 as output (LED) */
	MAP_GPIO_setAsPeripheralModuleFunctionInputPin( GPIO_PORT_P1, GPIO_PIN2 | GPIO_PIN3,
GPIO_PRIMARY_MODULE_FUNCTION );
	MAP_GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
	MAP_GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
}
/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook( void )
{
	/* vApplicationMallocFailedHook() will only be called if
	configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
	function that will get called if a call to pvPortMalloc() fails.
	pvPortMalloc() is called internally by the kernel whenever a task, queue,
	timer or semaphore is created.  It is also called by various parts of the
	demo application.  If heap_1.c or heap_2.c are used, then the size of the
	heap available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
	FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
	to query the size of free heap space that remains (although it does not
	provide information on how the remaining heap might be fragmented). */
	taskDISABLE_INTERRUPTS();
	for( ;; );
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook( void )
{
	/* vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set
	to 1 in FreeRTOSConfig.h.  It will be called on each iteration of the idle
	task.  It is essential that code added to this hook function never attempts
	to block in any way (for example, call xQueueReceive() with a block time
	specified, or call vTaskDelay()).  If the application makes use of the
	vTaskDelete() API function (as this demo application does) then it is also
	important that vApplicationIdleHook() is permitted to return to its calling
	function, because it is the responsibility of the idle task to clean up
	memory allocated by the kernel to any task that has since been deleted. */
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName )
{
	( void ) pcTaskName;
	( void ) pxTask;

	/* Run time stack overflow checking is performed if
	configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.  This hook
	function is called if a stack overflow is detected. */
	taskDISABLE_INTERRUPTS();
```

```
	for( ;; );
}
/*-----------------------------------------------------------*/

void *malloc( size_t xSize )
{
	/* There should not be a heap defined, so trap any attempts to call
	malloc. */
	Interrupt_disableMaster();
	for( ;; );
}
/*-----------------------------------------------------------*/
```

When you compile and run this code the LED should flash for 200 msecs. Now you are ready to work the assignment.

Here is your assignment:

Now for your chance to create some code:

1. Using FreeRTOS to create a system that you can interrupt with Switch 1 on the MSP432 2. When you interrupt then flash the LED on Pin 4.0. Use shared memory to determine how long to flash the LED on. Initialize the memory to 200 msec. Protect the shared memory with a mutex. When an interrupt occurs increment the shared memory by 200 msec, then turn on the LED for the specified length of time. Use the 4.1 LED to indicate that the system is ready for another button press.

2. Submit your source code for the lab on i-learn.

3. Hook up the Logic analyzer to capture the signals to the LEDs. Capture a trace. Include it in the Lab report.