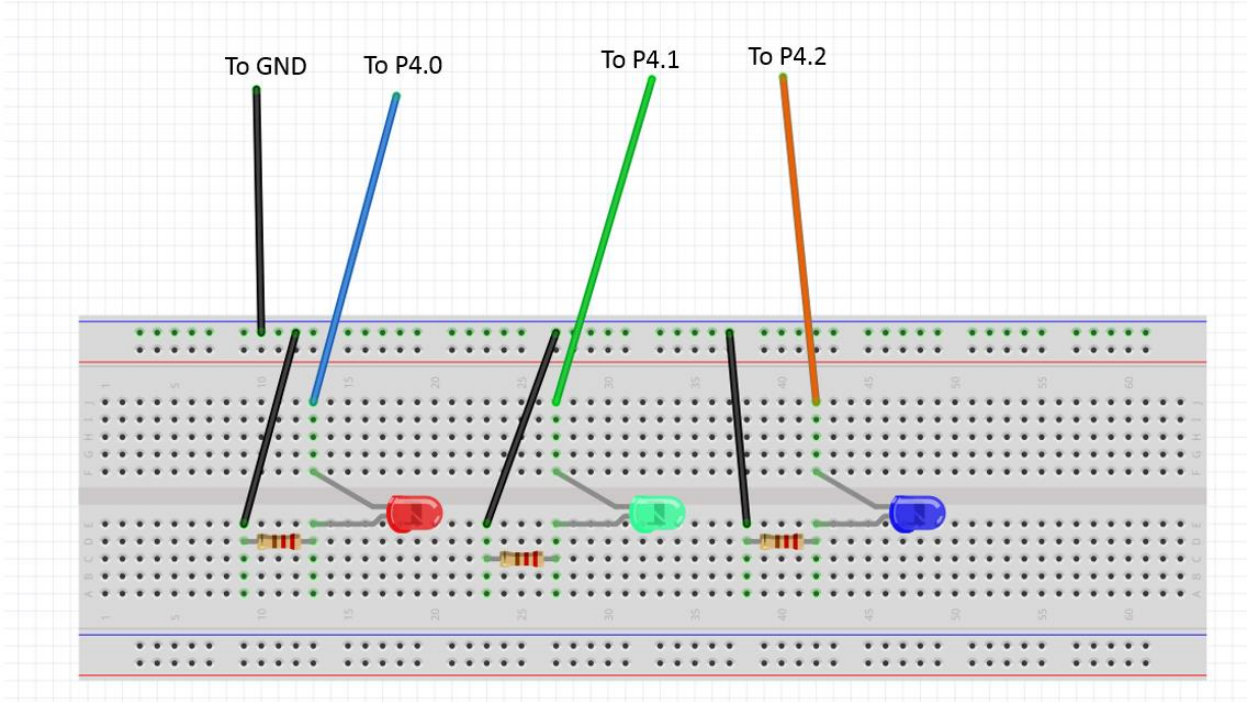
Lab Guide: FreeRTOS Queues on MSP432

Introduction:

An important feature of most RTOS implementations is a queue system, a way to communicate between tasks/threads/processes. A queue can also be a way to coordinate when activities happen, as a synchronization mechanism.

Constructing the Hardware

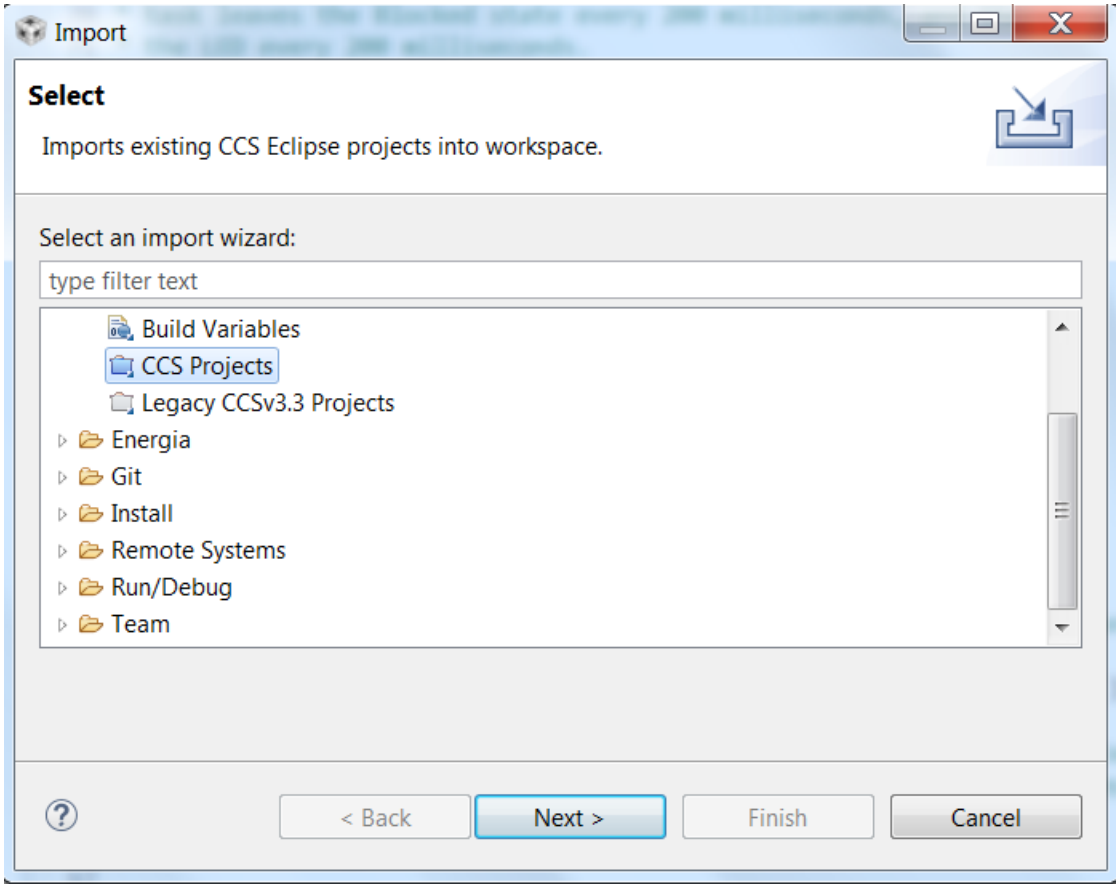
You'll use the same hardware for this lab as you used for the last lab.



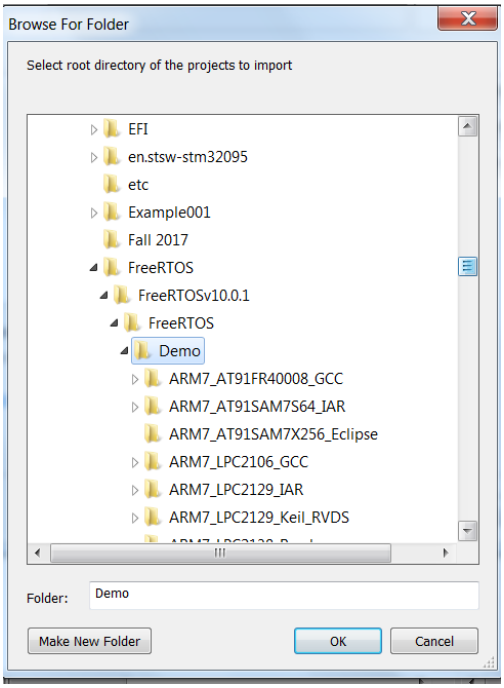
Communicating between processes using a queue

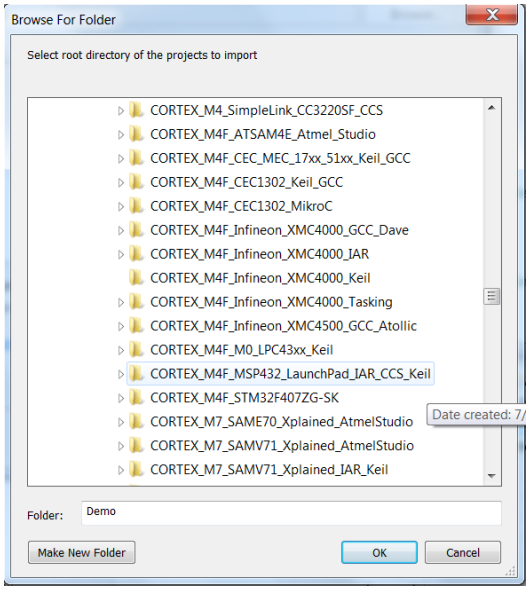
Queues are used for two purposes - synchronization and communicating between processes. Fortunately the FreeRTOS demo provides an example that illustrates both of these. To get this delete the RTOSDemo from your project space. Then reinstall the code using the following instructions:

Let's open the basic example provided in the FreeRTOS package. To do this select Import->

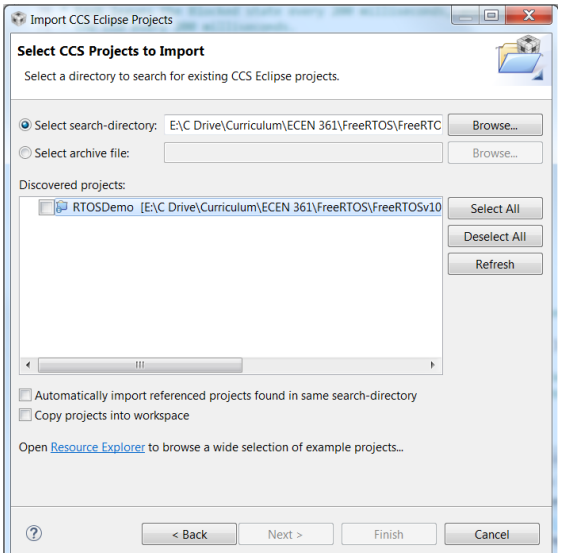


Hit Next. Now select the Browse button by the Select search-directory. Go to FreeRTOS/FreeRTOSv10.0.0.1/FreeRTOS/DEMO/CORTEX_M4F_MSP432_LaunchPad_IAR_CCS_KEIL, like this

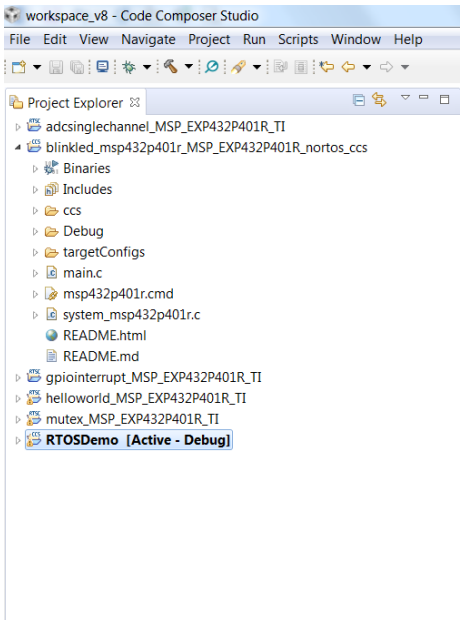




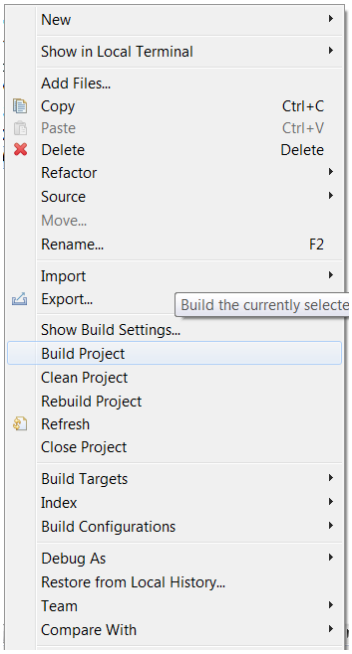
Then click OK.
Now you should see this selection:



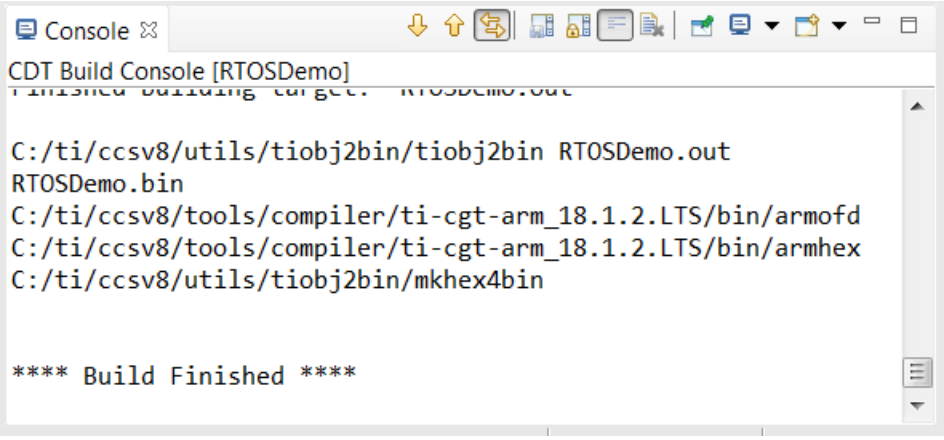
Make sure the RTOSDemo is selected, click both the two boxes at the bottom, then click Finish, and the Project should appear in your projects folder.



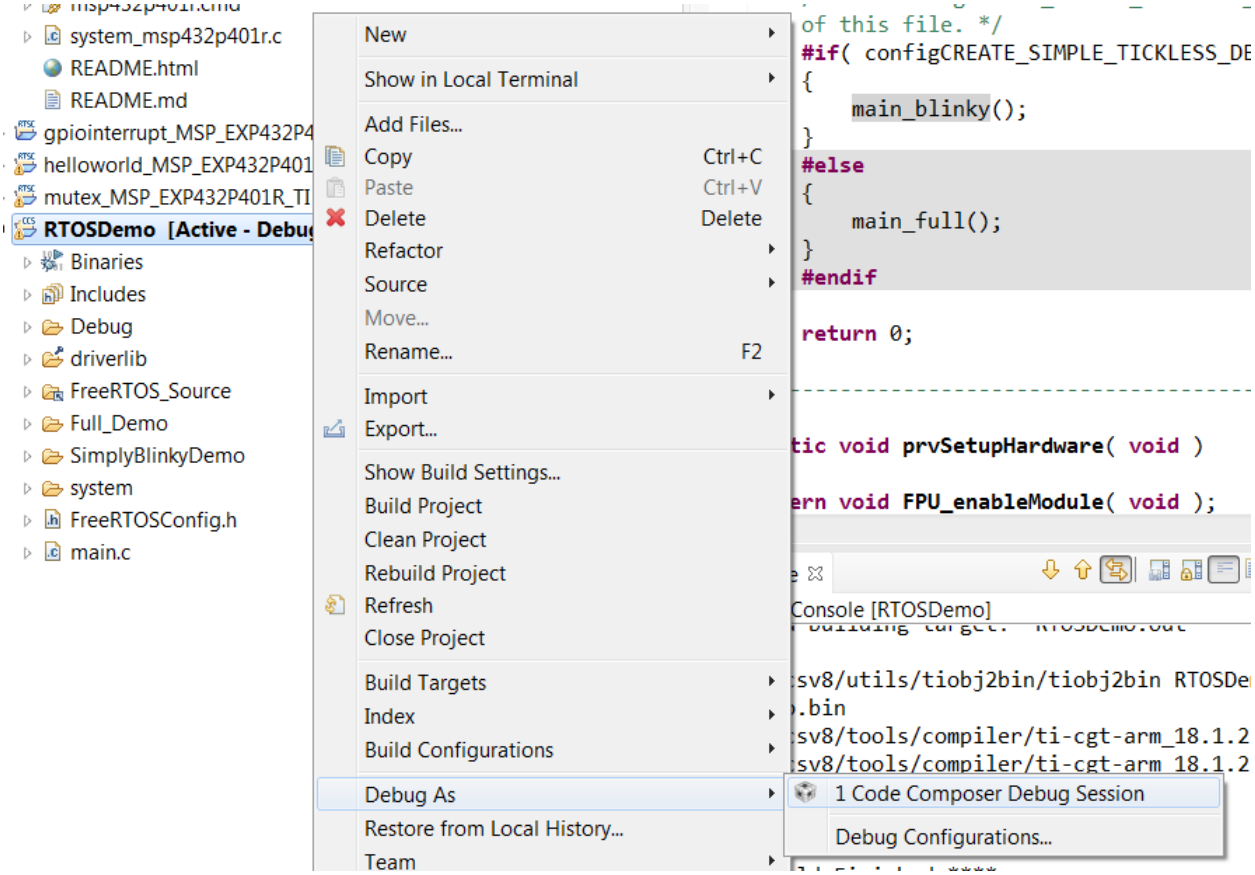
To make sure it installed correctly, go head and Build the project by right clicking on the project, then selecting Build Project:



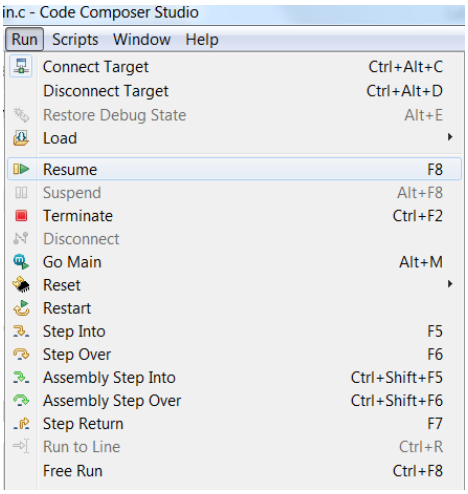
In the Build Console window you should see that the code has built.



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

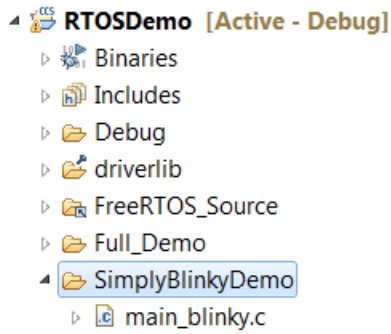


If everything worked as it should, your code will now be downloaded to your processor. To enable it to run, click Run -> Resume.



A Red LED should be blinking on your MSP432 board.

The code you just imported has two different code sets that can run at defaults. The first is the SimplyBlinkyDemo. This is the code you are running. Now go to the SimplyBlinkyDem and look at the main_blinky.c code.



This code is an example of how to set up a queue and communicate between two task using the queue. The code will create two tasks, a send task and a receive task. The receive task will be higher in priority. Then main will initialize the system, and then create the two tasks.

The higher of the two tasks, the receive task, will check the queue and block because there is nothing in the queue. The send task will then run and place something in the queue. It will then block for a set of time. When the task hits the delay function the receive task will be woken up by the item in the queue, and will run and toggle the LED.

Let's look in detail at the code:

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

Kernel includes.
```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
```

TI includes.
```
#include "gpio.h"
```

These two define the priority of both the send and the receive task.
```
#define mainQUEUE_RECEIVE_TASK_PRIORITY		( tskIDLE_PRIORITY + 2 )
#define  mainQUEUE_SEND_TASK_PRIORITY		( tskIDLE_PRIORITY + 1 )
```

The rate at which data is sent to the queue. The 200ms value is converted to ticks using the portTICK_PERIOD_MS constant.
```
#define mainQUEUE_SEND_FREQUENCY_MS			( pdMS_TO_TICKS( 1000UL ) )
```

The number of items the queue can hold. This is 1 as the receive task will remove items as they are added, meaning the send task should always find the queue empty.
```
#define mainQUEUE_LENGTH				( 1 )
```

Values passed to the two tasks just to check the task parameter functionality. These are just an example, they are not used in the actual code
```
#define mainQUEUE_SEND_PARAMETER			( 0x1111UL )
#define mainQUEUE_RECEIVE_PARAMETER			( 0x22UL )
```

Here are the pointers to the two functions that the tasks run as described in the comments at the top of this file.
```
static void prvQueueReceiveTask( void *pvParameters );
static void prvQueueSendTask( void *pvParameters );
```

```
 void main_blinky( void );

static void prvConfigureClocks( void );

This is an example function to create an interrupt to be serviced. This will not be used in this example
static void prvConfigureButton( void );

The queue used by both tasks.
static QueueHandle_t xQueue = NULL;

void main_blinky( void )
{
	prvConfigureClocks();

	prvConfigureButton();

	Create the queue.
	xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) );

	if( xQueue != NULL )
	{
		Start the two tasks
		xTaskCreate( prvQueueReceiveTask,	/* The function that implements the task. */
			"Rx",				/* The text name assigned to the task. */
			configMINIMAL_STACK_SIZE,	/* The size of the stack. */
			( void * ) mainQUEUE_RECEIVE_PARAMETER, /* The parameter passed */
			mainQUEUE_RECEIVE_TASK_PRIORITY, /* The priority. */
			NULL );			/* The task handle, NULL is passed. */

		xTaskCreate( prvQueueSendTask, "TX", configMINIMAL_STACK_SIZE, ( void * )
		mainQUEUE_SEND_PARAMETER, mainQUEUE_SEND_TASK_PRIORITY, NULL );

		Start the tasks and timer running.
		vTaskStartScheduler();
	}

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

static void prvQueueSendTask( void *pvParameters )
{
TickType_t xNextWakeTime;
const unsigned long ulValueToSend = 100UL;

	Check the task parameter is as expected. This just an example of how to use the parameter
	configASSERT( ( ( unsigned long ) pvParameters ) == mainQUEUE_SEND_PARAMETER );

	Initialise xNextWakeTime - this only needs to be done once.
	xNextWakeTime = xTaskGetTickCount();

	for( ;; )
	{
		Place this task in the blocked state until it is time to run again. The block time is specified in
		ticks, the constant used converts ticks to ms.  While in the Blocked state this task will not
		consume any CPU time.
```

```
                vTaskDelayUntil( &xNextWakeTime, mainQUEUE_SEND_FREQUENCY_MS );

                /* Send to the queue - causing the queue receive task to unblock and toggle the LED.  0 is used
                as the block time so the sending operation will not block - it shouldn't need to block as the
                queue should always be empty at this point in the code. */
                xQueueSend( xQueue, &ulValueToSend, 0U );
        }
}
/*-----------------------------------------------------------*/

static void prvQueueReceiveTask( void *pvParameters )
{
unsigned long ulReceivedValue;
static const TickType_t xShortBlock = pdMS_TO_TICKS( 50 );

        /* Check the task parameter is as expected. Again just another example */
        configASSERT( ( ( unsigned long ) pvParameters ) == mainQUEUE_RECEIVE_PARAMETER );

        for( ;; )
        {
                /* Wait until something arrives in the queue - this task will block indefinitely provided
                INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. */
                xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );

                /* To get here something must have been received from the queue, but is it the expected value?
                If it is, toggle the LED. */
                if( ulReceivedValue == 100UL )
                {
                        /* Blip the LED for a short while so as not to use too much power. */
                        configTOGGLE_LED();
                        vTaskDelay( xShortBlock );
                        configTOGGLE_LED();
                        ulReceivedValue = 0U;
                }
        }
}
```

Here is your assignment:

Now for your chance to create some code:

1. Just like on the mutex lab, using FreeRTOS to create a system that you can interrupt with the switch on the MSP432. When you interrupt then flash the LED on Pin 4.0. Only this time use a queue to determine how long to flash the LED on. Start with 200 msec and increment it 200 msec each time the button is pressed. Use the LED on Pin 4.1 to indicate that the system is ready for another button press. If the length of time for flashing the LED is longer than 2 seconds, then reset it to 200 msec.

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

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