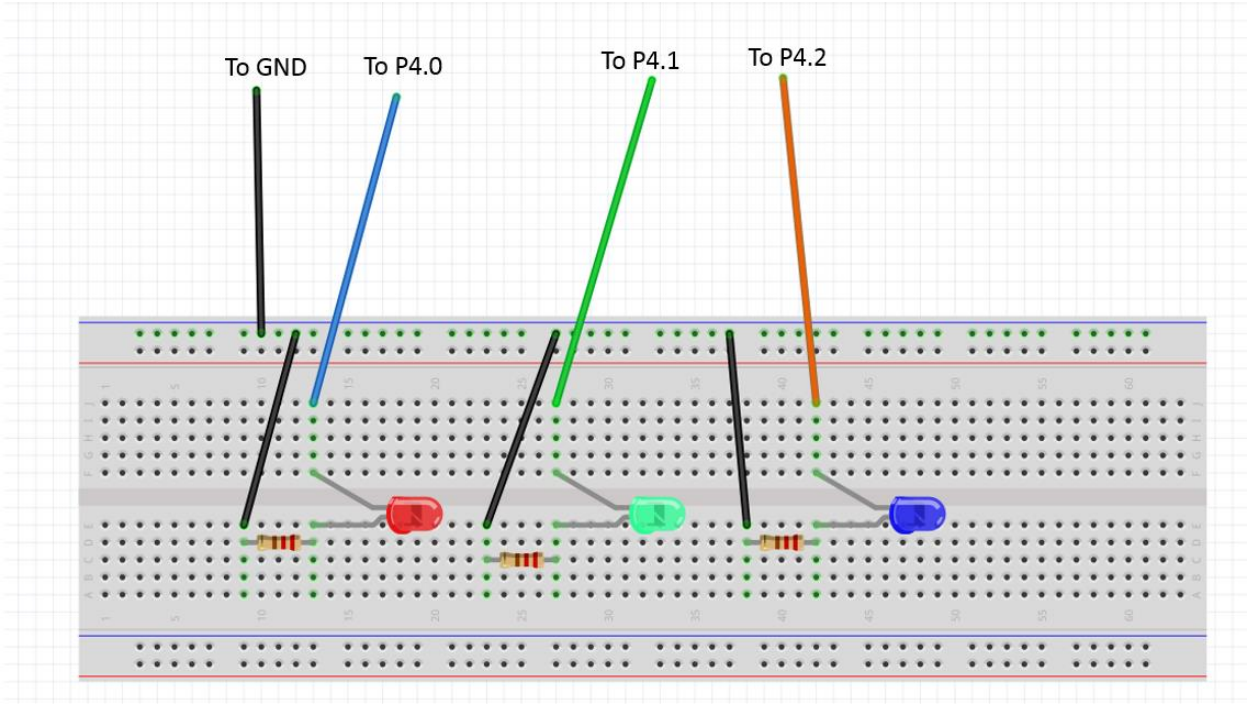
Lab Guide: FreeRTOS Introduction

Introduction

You have now written your own non-pre-emptive scheduler. The next step would be to write your own pre-emptive scheduler, but the jump in complexity is significant for two reasons; first, you would need to learn how to efficiently save the context (the stack pointer, the program counter, and the register state) and second, you'd need to use a timer interrupt to invoke the scheduler. Instead of focusing on these tasks, you're going to use a free version of an rtos that has the scheduler already written. It is called FreeRTOS, and is quite a powerful set of code. It implements not only a scheduler, but also resource management and inter-process communication as well. All of this runs on a number of different hardware implementations, but the key for you is that it runs on the Arduino. So you'll learn about these using this implementation.

Constructing the Hardware

For this lab construct the following LED circuit and connect it to the MSP432.

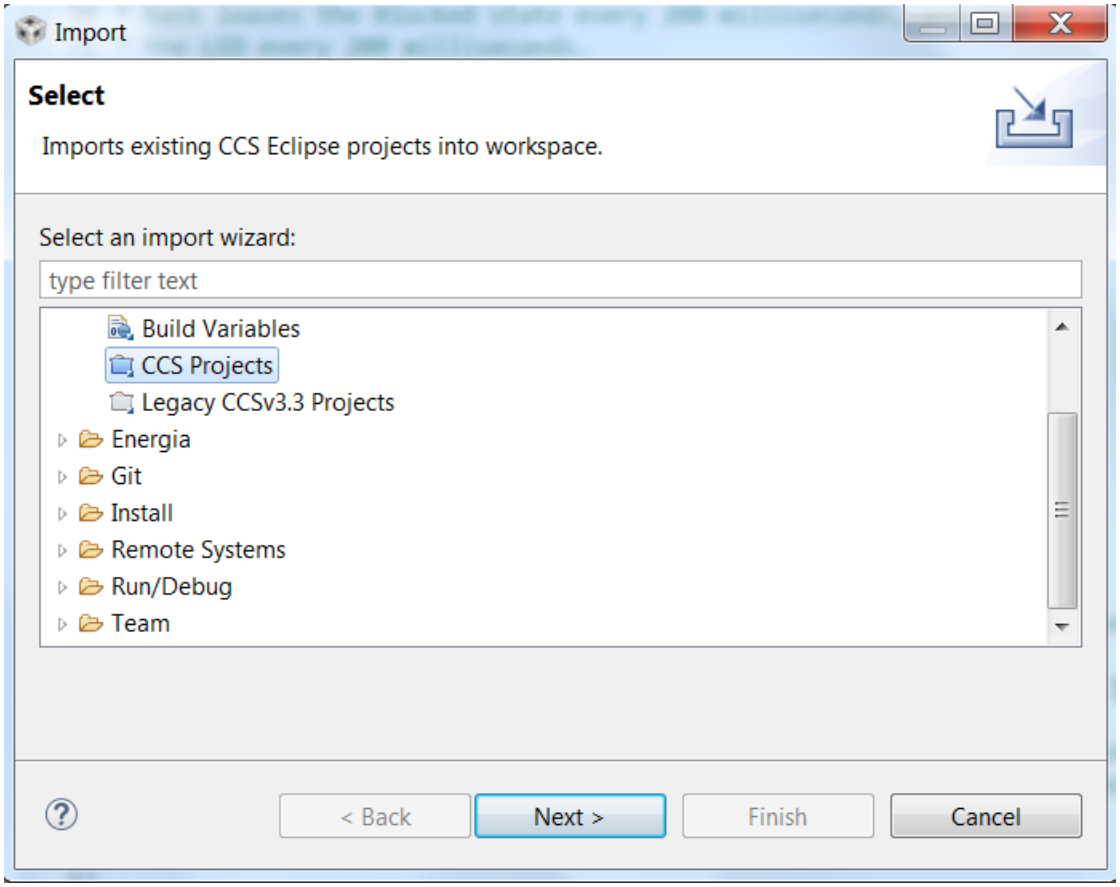


Installing FreeRTOS

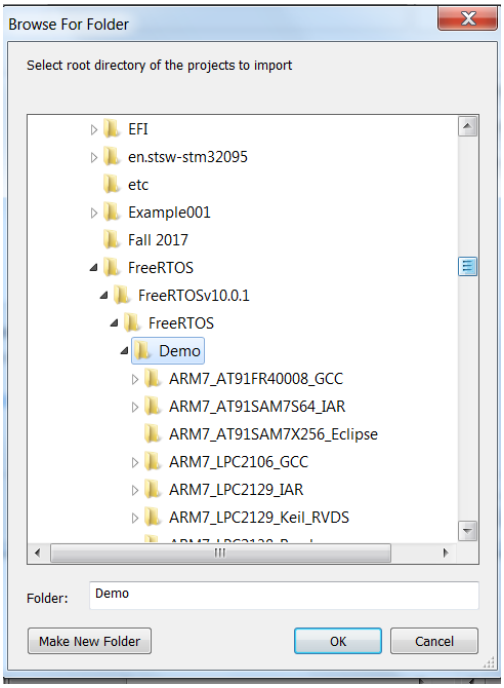
FreeRTOS comes as a free download. The instructions for how to use FreeRTOS with your MSP432 are included here: https://www.freertos.org/TI_MSP432_Free_RTOS_Demo.html#GCC.

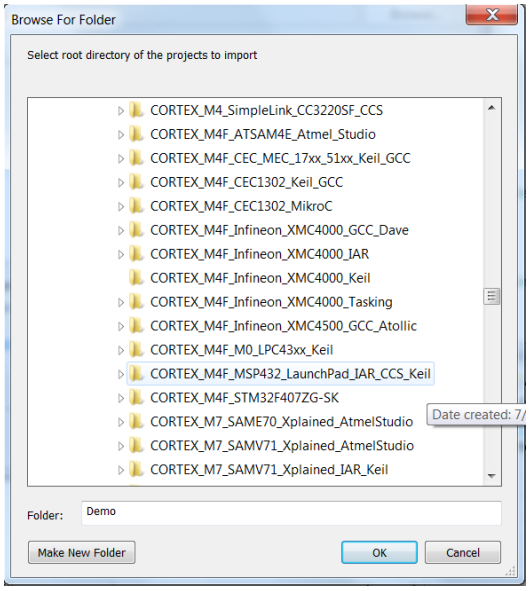
You'll download the file from https://www.freertos.org/a00104.html . Download the exe zip file onto your pc. Now run that file and unzip the files to a known location on your PC. Now open your CodeComposer development environment.

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

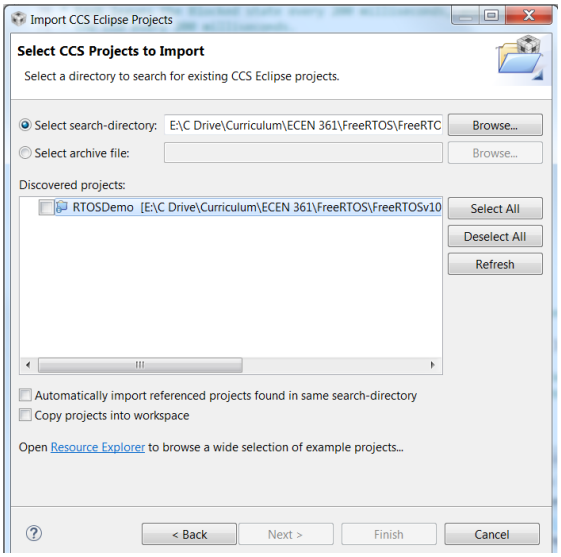


Hit Next. Now select the Browse button by the Select search-directory. Go to FreeRTOS/FreeRTOSv10.0.0.1/FreeRTOS/DEMO/CORTEX_M4F_MSP432_LaunchPad_IAR_CCS_KEIL, like this

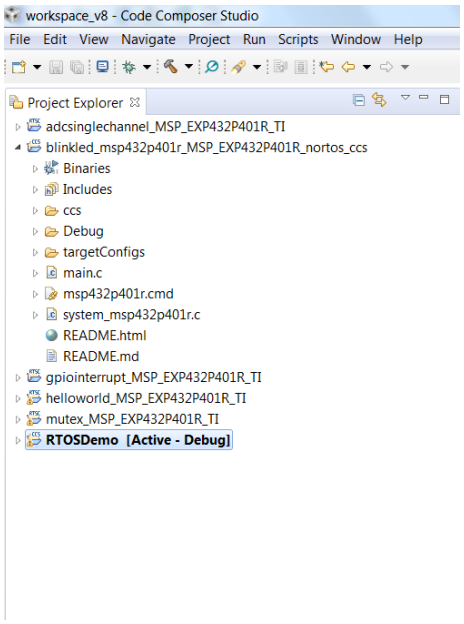




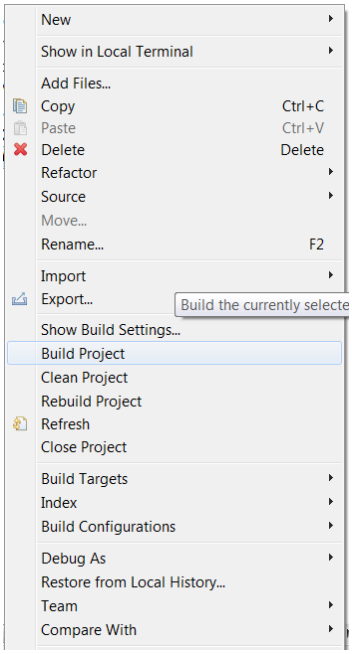
Then click OK.
Now you should see this selection:



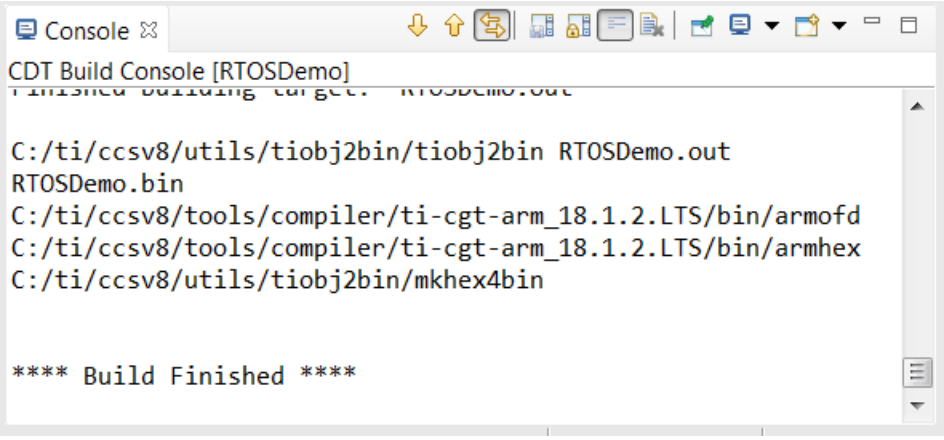
Make sure the RTOSDemo is selected, click both the two boxes at the bottom, then click Finish, and the Project should appear in your projects folder.



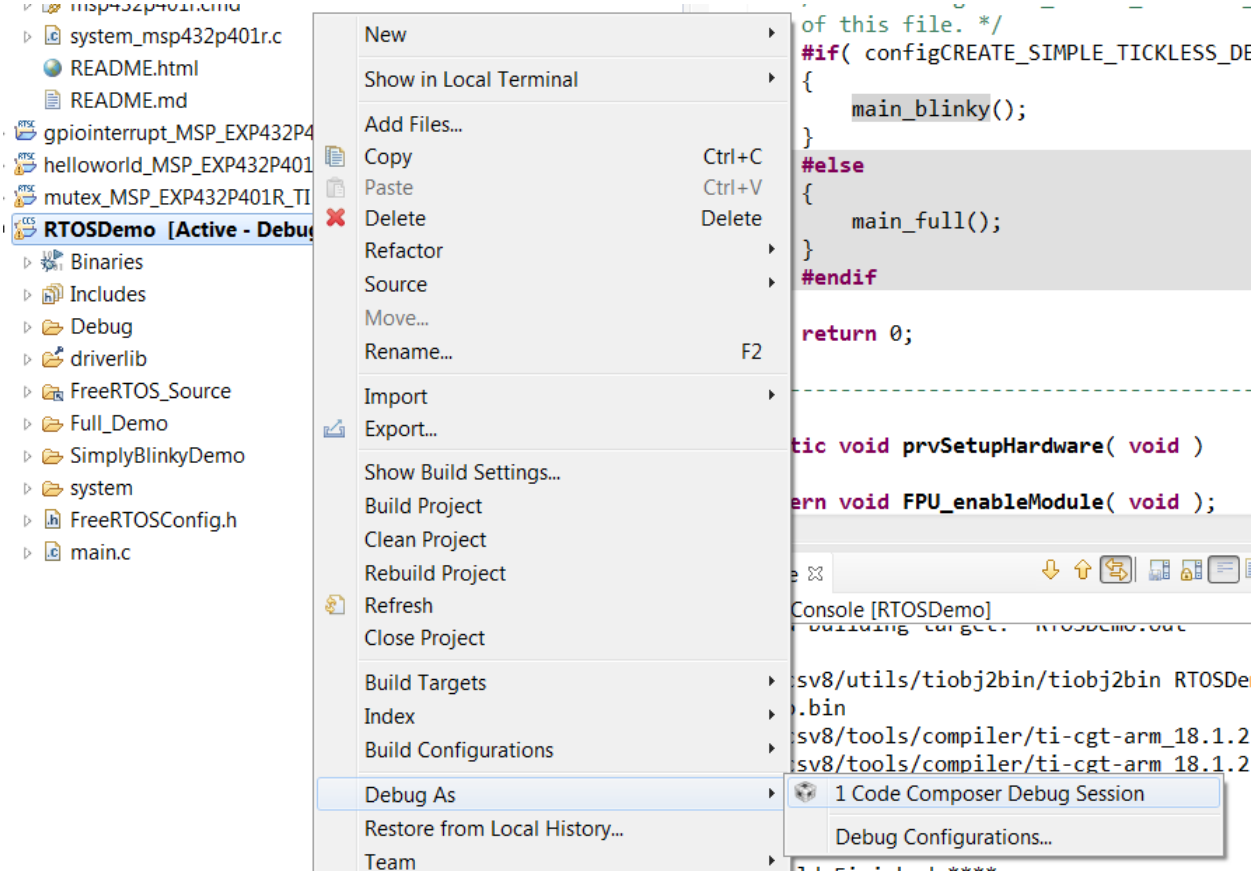
To make sure it installed correctly, go head and Build the project by right clicking on the project, then selecting Build Project:



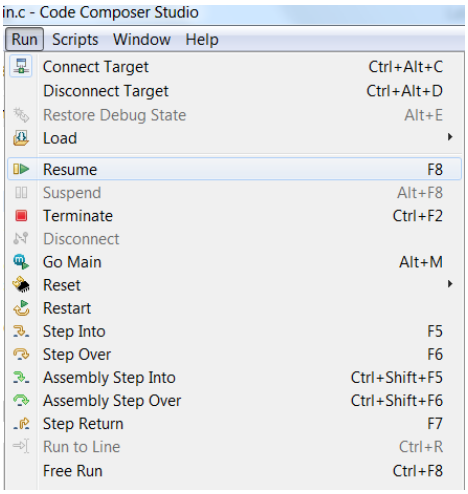
In the Build Console window you should see that the code has built.



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

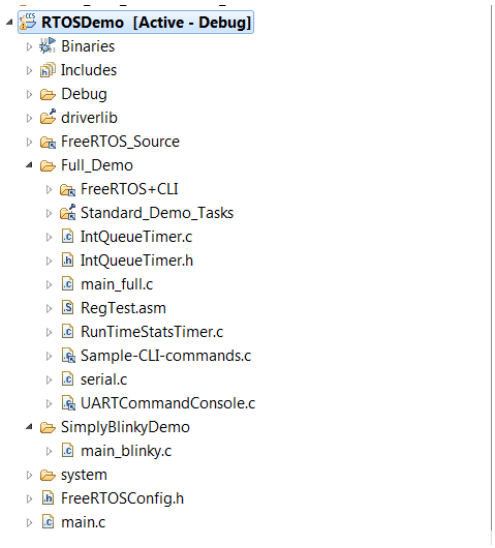


If everything worked as it should, your code will now be downloaded to your processor. To enable it to run, click Run -> Resume.



A Red LED should be blinking on your MSP432 board.

Now let's take a brief look at the code. If you look at the directory you'll see several different .c files with main in the title:



The main.c at the RTOSDemo directory level is file that allows the user to select between a simple demo, the code in SimplyBlinkyDemo directory, or the more full feature demo, the code in the Full_Demo directory. This selection is made by a variable definition in the FreeRTOSConfig.h, the

#define configCREATE_SIMPLE_TICKLESS_DEMO 1

If this #define is set to 1, the code will run the simple blinking demo. If is set to 0, then it will run the full featured demo. Here is that code in main.c:

```
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;
}
```

We'll not go into detail on these right now, let's start with a simple implementation of FreeRTOS.

Your first FreeRTOS application

Let's build your first FreeRTOS application, we'll put it in the main.c file. You'll take everything out of the main.c file, and replace it with the code you see here (it is included in a .txt file on i-learn for you to copy.) Now let's go through the code so you can understand how FreeRTOS works.

The first set of statements are the includes:

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

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

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

These two prototypes are the two functions you'll be using for your project.

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

Now a prototype for a function that is needed by the MSP432 configuration

```
/*
 * The full demo configures the clocks for maximum frequency, wheras this blinky
 * demo uses a slower clock as it also uses low power features.
 */
static void prvConfigureClocks( void );
```

And now main

```
void main( void )
{
        /* See http://www.FreeRTOS.org/TI_MSP432_Free_RTOS_Demo.html for
        instructions and notes regarding the difference in power saving that can be
        achieved between using the generic tickless RTOS implementation (as used by
        the blinky demo) and a tickless RTOS implementation that is tailored
        specifically to the MSP432. */

        /* The full demo configures the clocks for maximum frequency, wheras this
        blinky demo uses a slower clock as it also uses low power features. */
        prvConfigureClocks();

  /* Create the first task at priority 1... */
  xTaskCreate( vTaskFunction2, "Task 1", 200, NULL, 1, NULL );
```

The xTaskCreate function adds a task to the task queue. The arguments are these:

vTaskFunction – The function to be run when starting the task for the first time.

"Task 1" – The name of the task.

200 – The stack size – How much memory do you want to set aside for the stack. This will include storing the calling tree (what functions have been called to get to this point), as well as any local variables.

NULL – This is the data that will be passed to the code the first time the task is called. In this case you aren't passing any data. Notice that it is cast as a void pointer, which means it is changed from a character pointer to a pointer of no type. This configuration allows you to pass as little or as much data as you need, just know it starts at this address.

1 – The priority of the task. The higher the number the higher the priority.

NULL – This last parameters allows you to pass a handle to the task, but this is normally not used, so this is almost always NULL.

And now you set up a second task.

```
  /* ... and the second task at priority 2.  The priority is the second to
  last parameter. */
  xTaskCreate( vTaskFunction2, "Task 2", 200, NULL, 2, NULL );
```

Once the tasks are set up you start the scheduler. Since this is a pre-emptive scheduler this function will never return, so whatever comes below should never be executed.

```
  /* Start the scheduler so our tasks start executing. */
  vTaskStartScheduler();

  for( ;; );   /* The code should never get to here
  return 0;
}
```

Here is the code for the first Task.

```
void vTaskFunction1( void *pvParameters )
{
```

This is the name of one of your tasks. Now you have a forever loop. This is very characteristic of this type of task. It will run forever. Remember, though, that the task will be interrupted (pre-empted) while in this loop, so can be moved either to or from the ready queue. Watch the animation video for an explanation of how this happens.

```
  /* As per most tasks, this task is implemented in an infinite loop. */
  for( ;; )
  {

      GPIO_toggleOutputOnPin(
                              GPIO_PORT_P4,
                              GPIO_PIN0
                              );
```

Now you are going to call vTaskDelay() with a value. This function in the FreeRTOS world moves this task to the blocked queue for a specific length of time. Thus any other tasks with lower priority that can be scheduled will be run while this is on the blocked queue. Once the delay time has passed the task is moved back to the ready queue, and then scheduled if it is the highest priority task.

```
    /* 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 specified in 'ticks'. */
    vTaskDelay( 250 / portTICK_PERIOD_MS );
  }
}
```

Here is the code for the second task.

```
void vTaskFunction2( void *pvParameters )
{
```

This is the name of the second of your tasks. Now you have a forever loop. This is very characteristic of this type of task. It will run forever. Remember, though, that the task will be interrupted (pre-empted) while in this loop, so can be moved either to or from the ready queue. Watch the animation video for an explanation of how this happens.

```
  /* As per most tasks, this task is implemented in an infinite loop. */
  for( ;; )
  {

      GPIO_toggleOutputOnPin(
                              GPIO_PORT_P4,
                              GPIO_PIN1
                              );
```

Now you are going to call vTaskDelay() with a value. This function in the FreeRTOS world moves this task to the blocked queue for a specific length of time. Thus any other tasks with lower priority that can be scheduled will be run while this is on the blocked queue. Once the delay time has passed the task is moved back to the ready queue, and then scheduled if it is the highest priority task.

```
    /* 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 specified in 'ticks'. */
    vTaskDelay( 500 / portTICK_PERIOD_MS );
  }
}
```

Here is the function to set up the clocks on the MSP430 and the output pins.

```
static void prvConfigureClocks( void )
{
  extern void FPU_enableModule( void );

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

  /* Ensure the FPU is enabled. */
  FPU_enableModule();
  /* Set Flash wait state for high clock frequency.  Refer to datasheet for
          more details. */
  FlashCtl_setWaitState( FLASH_BANK0, 2 );
  FlashCtl_setWaitState( FLASH_BANK1, 2 );

     From the datashee:  For AM_LDO_VCORE1 and AM_DCDC_VCORE1 modes, the maximum
    CPU operating frequency is 48 MHz and maximum input clock frequency for
    peripherals is 24 MHz. */
    PCM_setCoreVoltageLevel( PCM_VCORE1 );
    CS_setDCOCenteredFrequency( CS_DCO_FREQUENCY_48 );
    CS_initClockSignal( CS_HSMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1 );
    CS_initClockSignal( CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1 );
    CS_initClockSignal( CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1 );
    CS_initClockSignal( CS_ACLK, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1 );
```

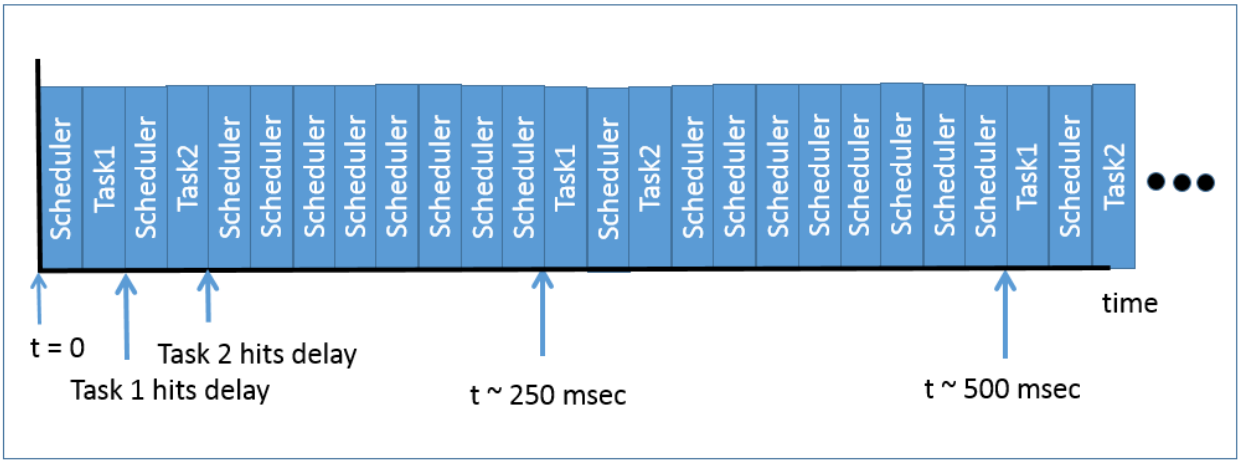
```
        /* Selecting P1.0 as output (LED) and P2.0 and output (LED) */

   MAP_GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    MAP_GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    MAP_GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN0 );
    MAP_GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN1 );


}
//---------------------------------------------------------------------------
```

And some additional code that you need to support this implementation of FreeRTOS on the MSP432.

```
/*-----------------------------------------------------------*/

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

So download and run this code on your MSP432. Watch the LEDs. What you are seeing is Task 1 running and toggling the LED every 250 msec, and Task 2 running and toggling the LED every 500 msec. When task one has toggled it goes to the blocked queue and task 2 runs, then when Task 2 gets scheduled and run, then Task 2 gets blocked and so forth. Here is the timing diagram for this code:



Here is your assignment:

1. Using FreeRTOS to recreate the system you created in Lab #4. The system will have three tasks. Each task will turn on and off an LED when it is run. Use the same timing as in Lab #4. Make sure you use vTaskDelay to delay the tasks. The third LED should be connected to Port4, Pin2.

   You can use a single function and pass a void pointer to a delay variable to establish the delays for the two timed LED values. The idle process could just always run. You will want to consider the priority parameter. Should they all be the same priority? Should they have different priorities?

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

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