Lab: MSP432P401R FreeRTOS Complete introduction

Introduction

This lab assumes you are at least familiar with the MSP432P401R and Code Composer. It will also assume that you are familiar with FREERTOS, an RTOS that can allow you create task and add additional capability to your project.

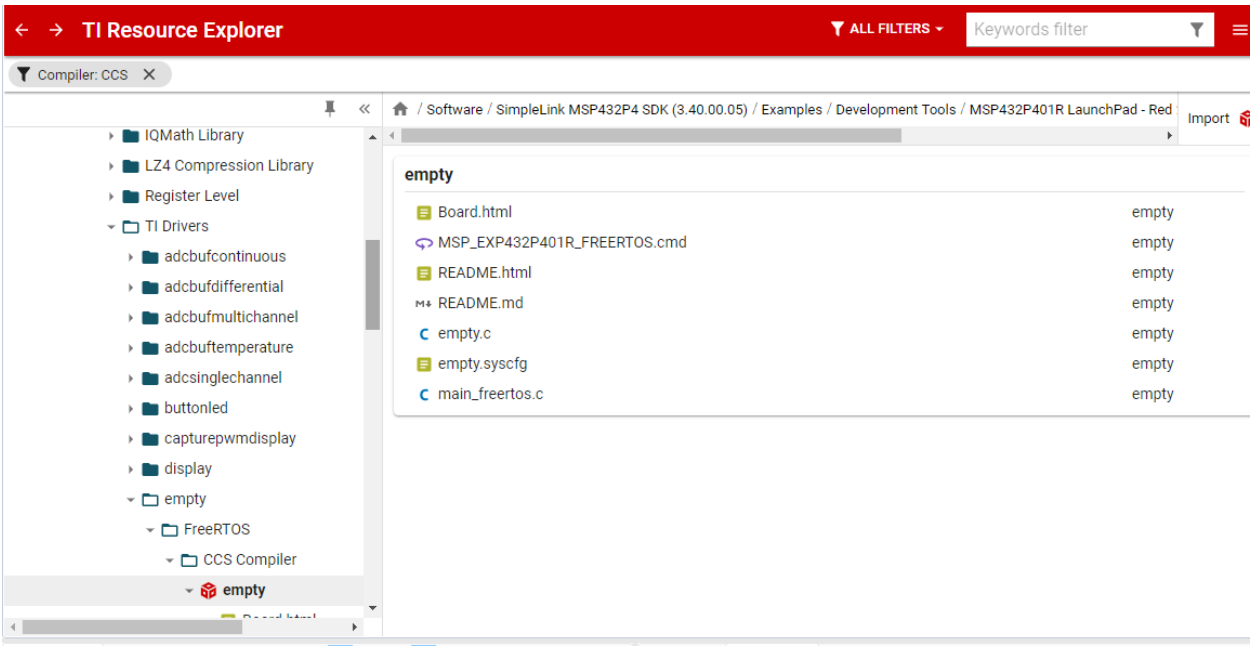
In this lab you'll learn how to add capability using the SysConfig utility to add functionality to your project. You'll also learn how to add FREERTOS tasks to your project.

Initial Setup

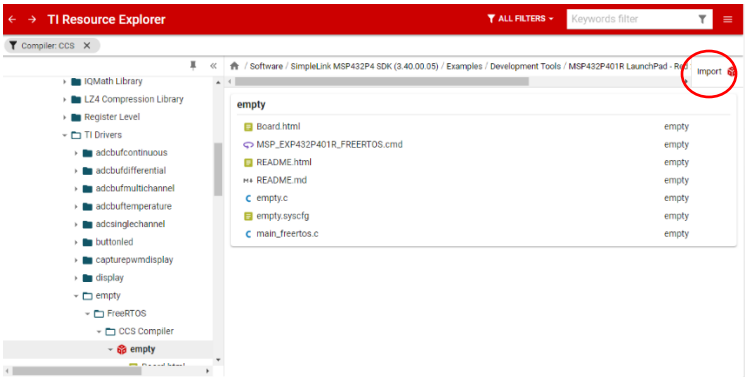
For this lab you will start with an empty project. To do this import the empty project template by selecting the empty project from this directory:

Software->SimpleLine MSP432P4 SDK -vxxx -> Examples -> Development Tools -> MSP432P401R LaunchPad – Red 2.x -> TI Drivers -> empty -> FreeRTOS 0> CCS Compiler -> empty

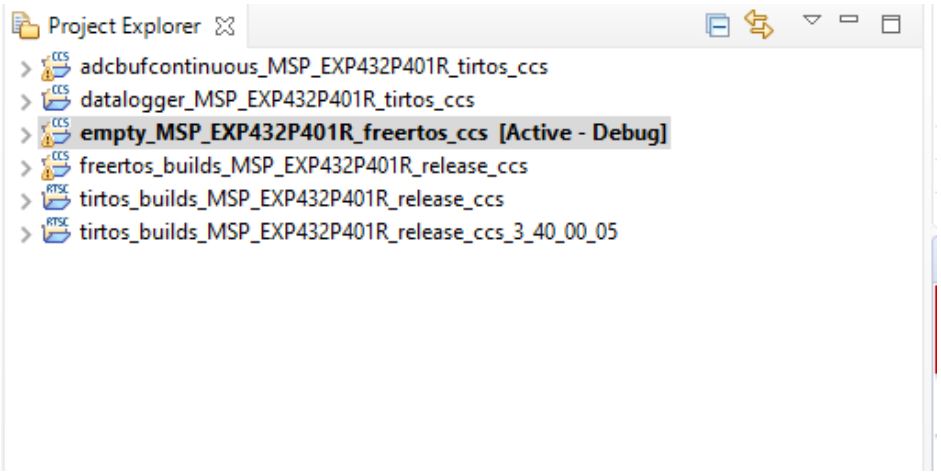
Like this:



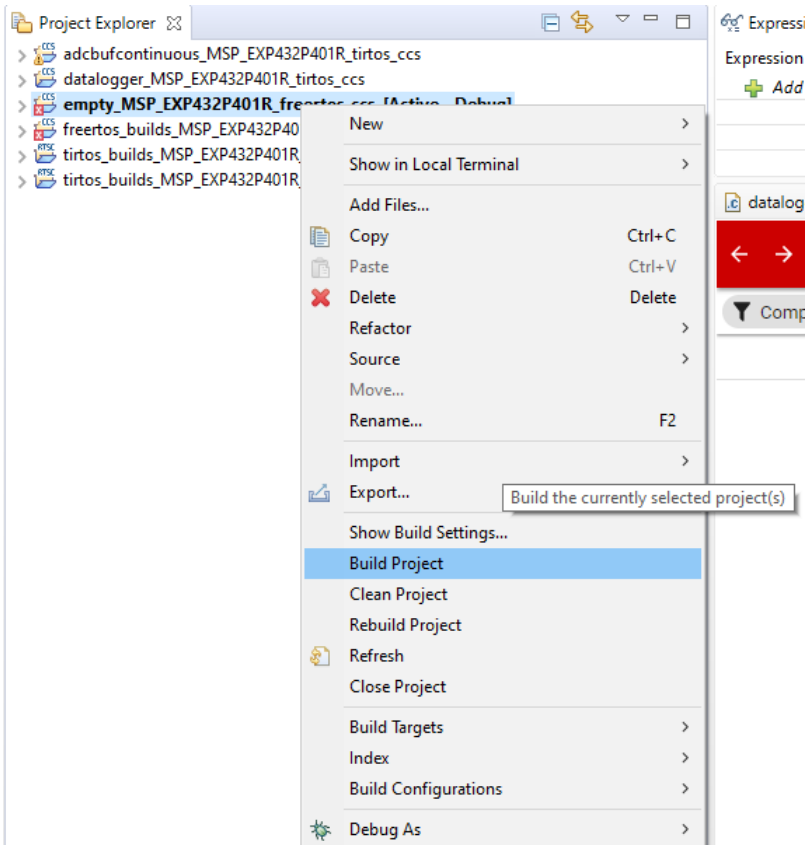
Double Click on the directory, then click on the Import to IDE button



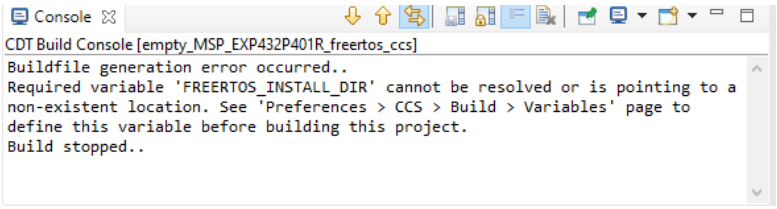
This project should then load into the project explorer window (you may get some warnings, or request to download additional projects to support this project. Select yes to these). You should now see this project in the window (along with any additional projects that are needed to support the project.):



To make sure everything has downloaded successfully, build the project to make sure that it builds successful. To do this select the project, then select the right mouse button, then the Build Project selection:

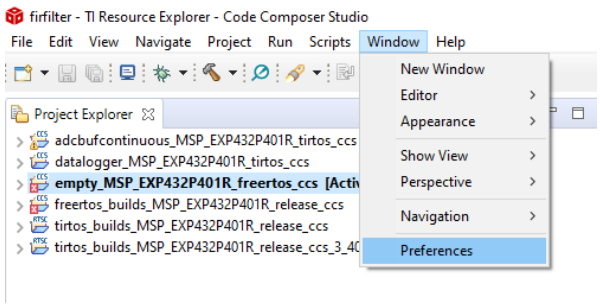


The build will not be successful, you should see this in your Console window:

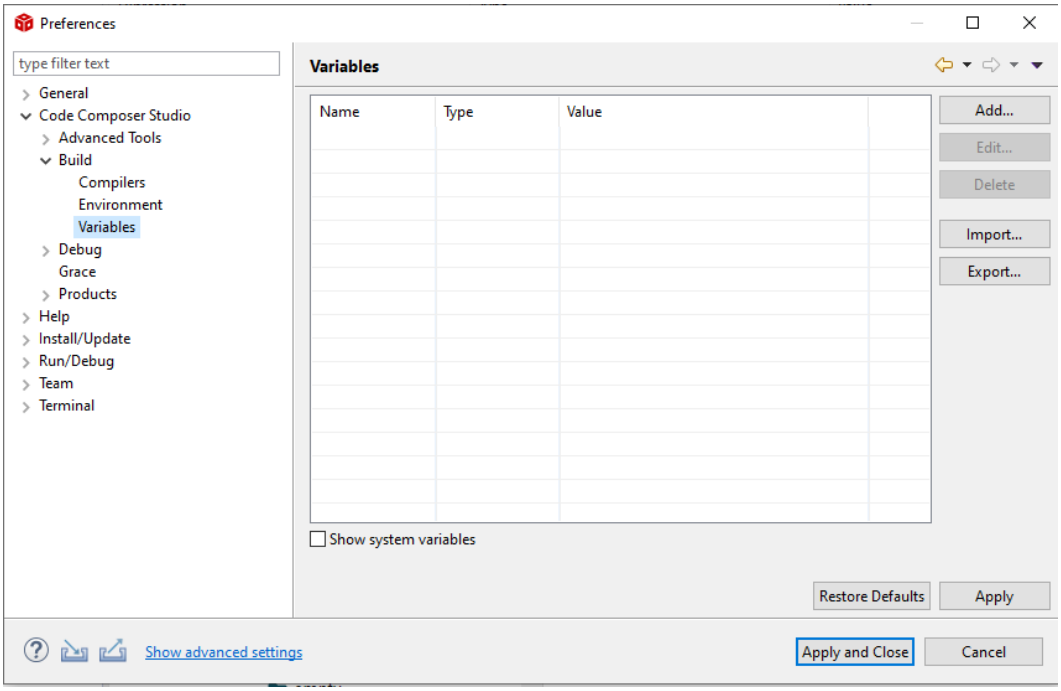


You will need to set a variable FREERTOS_INSTALL_DIR that holds the directory for your download of the FREERTOS code. You will need to download the latest FREERTOS code from freertos.org. This will download as an exe file, run the file and it will extract the code to a directory.

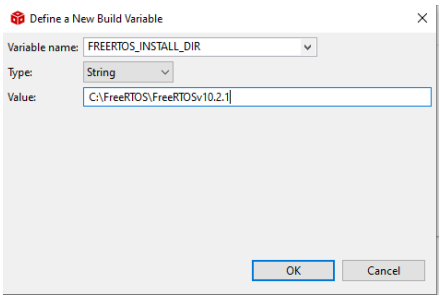
You'll need a new variable inside the Code Composer studio that points to that directory. So go to Window-> Preferences:



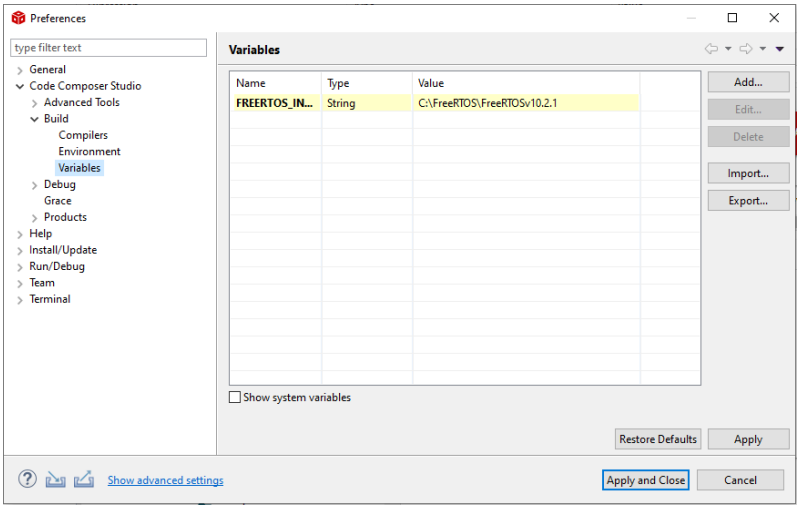
Once the Preference Window has popped up, go to the Code Composer Studio->Build->Variable selection:



Click on the Add selection and add the FREERTOS_INSTALL_DIR variable with the directory that you specified to install FREERTOS:



Hit OK, then Click Apply and Save.

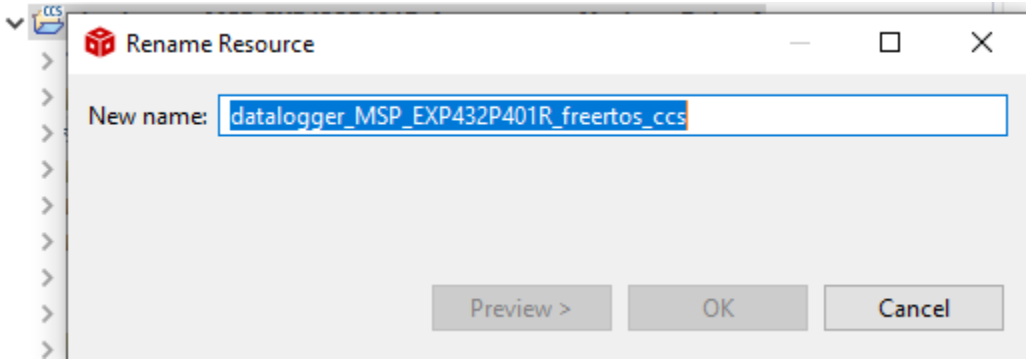


Build the Project, and the Console should now show that you have a successful build. Then download the code to the device and you should see the red LED blink. (Make sure you have hit Resume in the Run tab to run the code!)

Editing the Project

Rename the project and any file with "empty" in the name to something meaningful.
I'm using "datalogger" for this example.

The project name becomes "datalogger_MSP_EXP432P401R_freertos_ccs". Right-click on the project root in the Project explorer and select *Rename...*.



Then rename the following files by right clicking on them selecting rename:

empty.c -> datalogger.c
empty.cfg -> datalogger.cfg

Build and run the project again. That will show that you have successfully completed the steps. There's more to this example than meets the eye. You get the basic logging and instrumentation skeleton. And the controller is configured to go to low-energy mode when idle.

Let's look at the code a bit. First, let's start with the significant parts of main_rtos.c:

```
/* RTOS header files */
#include <FreeRTOS.h>
#include <task.h>

/* Driver configuration */
#include <ti/drivers/Board.h>

extern void *mainThread(void *arg0);

/* Stack size in bytes */
#define THREADSTACKSIZE   1024

/*
 *  ======== main ========
 */
int main(void)
{
    pthread_t           thread;
    pthread_attr_t      attrs;
    struct sched_param  priParam;
    int                 retc;

    /* initialize the system locks */
#ifdef __ICCARM__
    __iar_Initlocks();
#endif

    /* Call driver init functions */
    Board_init();

    /* Initialize the attributes structure with default values */
    pthread_attr_init(&attrs);

    /* Set priority, detach state, and stack size attributes */
    priParam.sched_priority = 1;
    retc = pthread_attr_setschedparam(&attrs, &priParam);
    retc |= pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    retc |= pthread_attr_setstacksize(&attrs, THREADSTACKSIZE);
    if (retc != 0) {
        /* failed to set attributes */
        while (1) {}
    }

    retc = pthread_create(&thread, &attrs, mainThread, NULL);
    if (retc != 0) {
        /* pthread_create() failed */
        while (1) {}
    }

    /* Start the FreeRTOS scheduler */
    vTaskStartScheduler();

    return (0);
```

}

If you've already played with FREERTOS you'll notice that this code provides functions for setting up the threads in FREERTOS. Specifically there is a function to set the priority, set the stack size, and set up the state of the thread. Then the pthread_create adds the task to the ready queue.

The vTaskStartScheduler() functions starts the scheduler and everything starts.

Notice specifically that there is only one task, and it will start by running the mainThread function.

This task is defined in the file datalogger.c. Let's now look at that code:

```
/* Driver Header files */
#include <ti/drivers/GPIO.h>
// #include <ti/drivers/I2C.h>
// #include <ti/drivers/SPI.h>
// #include <ti/drivers/UART.h>
// #include <ti/drivers/Watchdog.h>

/* Driver configuration */
#include "ti_drivers_config.h"

/*
 *  ======== mainThread ========
 */
void *mainThread(void *arg0)
{
    /* 1 second delay */
    uint32_t time = 1;

    /* Call driver init functions */
    GPIO_init();
    // I2C_init();
    // SPI_init();
    // UART_init();
    // Watchdog_init();

    /* Configure the LED pin */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);

    while (1) {
        sleep(time);
        GPIO_toggle(CONFIG_GPIO_LED_0);
    }
}
```

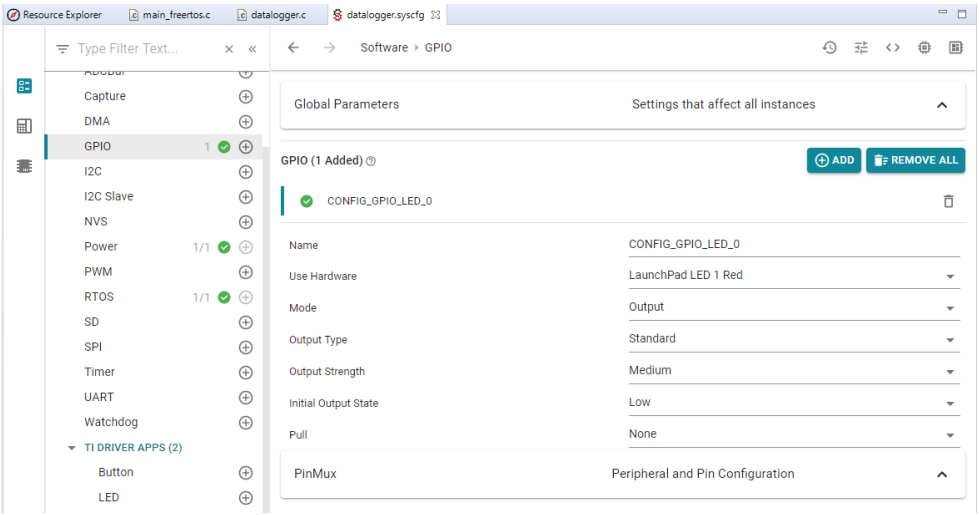
In this case the function is fairly straightforward. It sets up the GPIO (so you can use an LED), configures the LED, turns on the LED, then goes through a while forever loop flashing the LED once a second.

Using the SysConfig to add functionality to your project

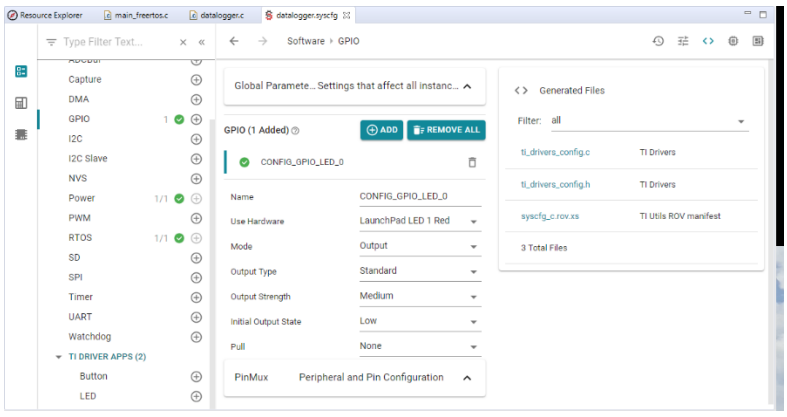
The SysConfig tool in Code Composer can make it easy to add additional capability to your project. So let's get started using it.

Now that we've confirmed the setup is working, now let's actually look at the SysConfig GUI Tool in CCS. We'll see where the `CONFIG_GPIO_LED_0` constant was defined and how the GPIO configuration was done also.

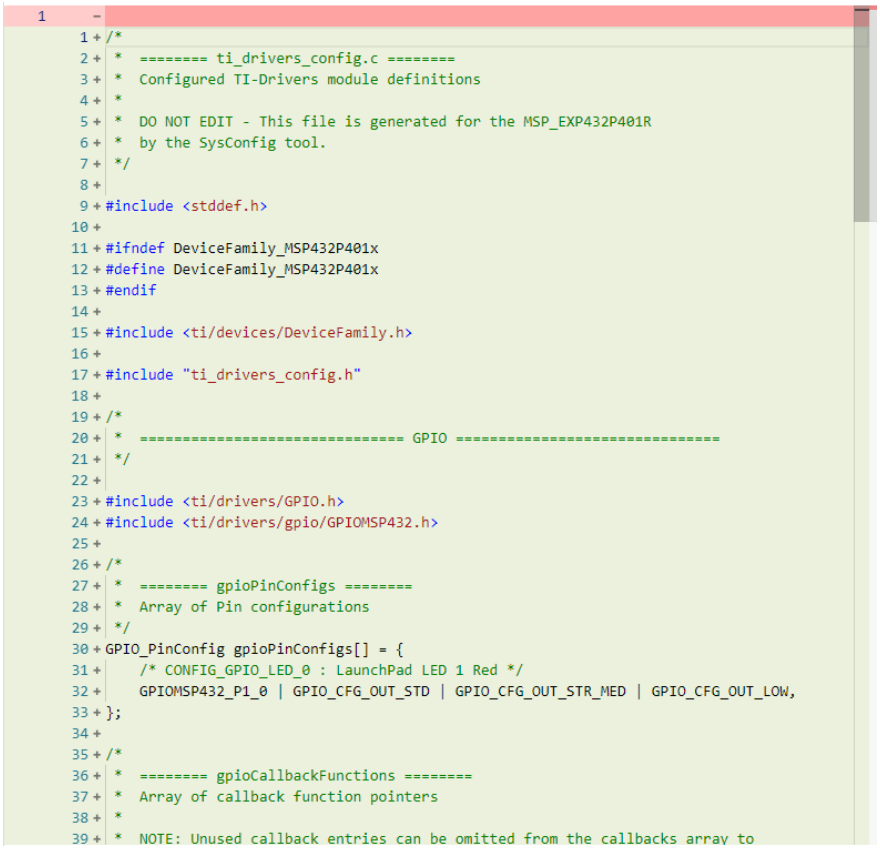
1. **Double-click the `datalogger.syscfg` file in the Project Explorer**. After a couple seconds you should see this window.



2. Let's look at a preview of the TI Driver configuration code that is going to be generated by the `datalogger.syscfg` file. **Select the `<>` icon on the top right of the SysConfig GUI.** Note: all SysConfig generated files can be previewed here. For example the `syscfg_c.rov.xs` file is generated for viewing non-TI-RTOS modules in Runtime Object View and `ti_ndk_config.c` file for MSP432E4 Networking configuration.



3. **Click the ti_drivers_config.c file.** This opens the preview of the generated `ti_drivers_config.c`. In later tasks in this lab, you'll see this file updated as we change the TI Driver configuration. As you look at the `ti_drivers_config.c`, you'll notice it is rather empty (as expected...the project is called Empty after all). Let's look at a couple key items in this file
    - *Generation Disclaimer*: At the top, you'll see the disclaimer to not modify this file since it is generated. Any changes made to this file will be wiped out when it is re-generated. We'll show how to modify these generated files later on in this lab (spoiler...add the files to the project and exclude the .syscfg file).
    - *Configuration of the LED*: The configuration for the LED that this example uses.
    - As changes are made, green shading denotes additions and red shading denotes removed items (we'll see this in action later).

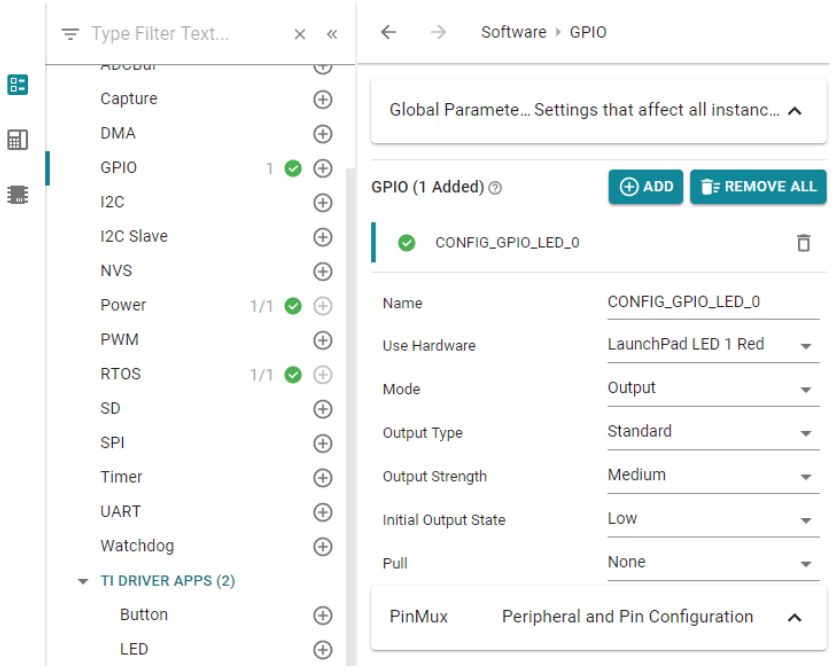


```
1 -
1 + /*
2 +  *  ======== ti_drivers_config.c ========
3 +  *  Configured TI-Drivers module definitions
4 +  *
5 +  *  DO NOT EDIT - This file is generated for the MSP_EXP432P401R
6 +  *  by the SysConfig tool.
7 +  */
8 +
9 + #include <stddef.h>
10 +
11 + #ifndef DeviceFamily_MSP432P401x
12 + #define DeviceFamily_MSP432P401x
13 + #endif
14 +
15 + #include <ti/devices/DeviceFamily.h>
16 +
17 + #include "ti_drivers_config.h"
18 +
19 + /*
20 +  *  ============================= GPIO ==============================
21 +  */
22 +
23 + #include <ti/drivers/GPIO.h>
24 + #include <ti/drivers/gpio/GPIOMSP432.h>
25 +
26 + /*
27 +  *  ======== gpioPinConfigs ========
28 +  *  Array of Pin configurations
29 +  */
30 + GPIO_PinConfig gpioPinConfigs[] = {
31 +     /* CONFIG_GPIO_LED_0 : LaunchPad LED 1 Red */
32 +     GPIOMSP432_P1_0 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_MED | GPIO_CFG_OUT_LOW,
33 + };
34 +
35 + /*
36 +  *  ======== gpioCallbackFunctions ========
37 +  *  Array of callback function pointers
38 +  *
39 +  *  NOTE: Unused callback entries can be omitted from the callbacks array to
```

Device/Board Specific

Since the low level driver configuration is device/board specific, if you are using a different device or board, your generated `ti_drivers_config.c` will have different names for the constants and data structures.

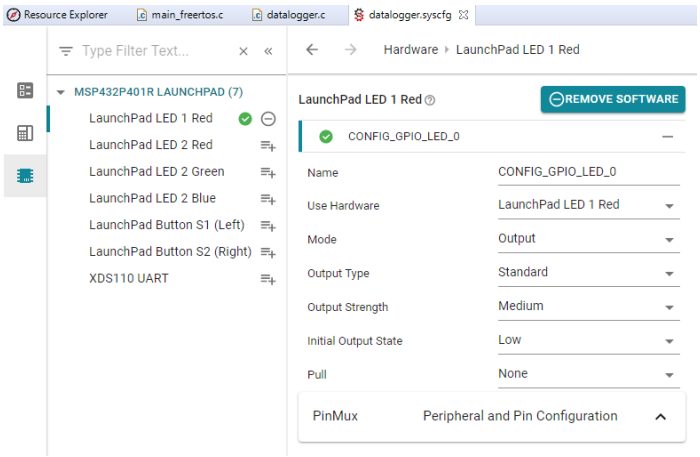
4. **Select the GPIO module in the Software view.** Here is the configuration for there the LED that the example uses.



Where did the LaunchPad LED 1 Red in the Use Hardware field in the above picture come from?

Exploring SysConfig (Hardware)

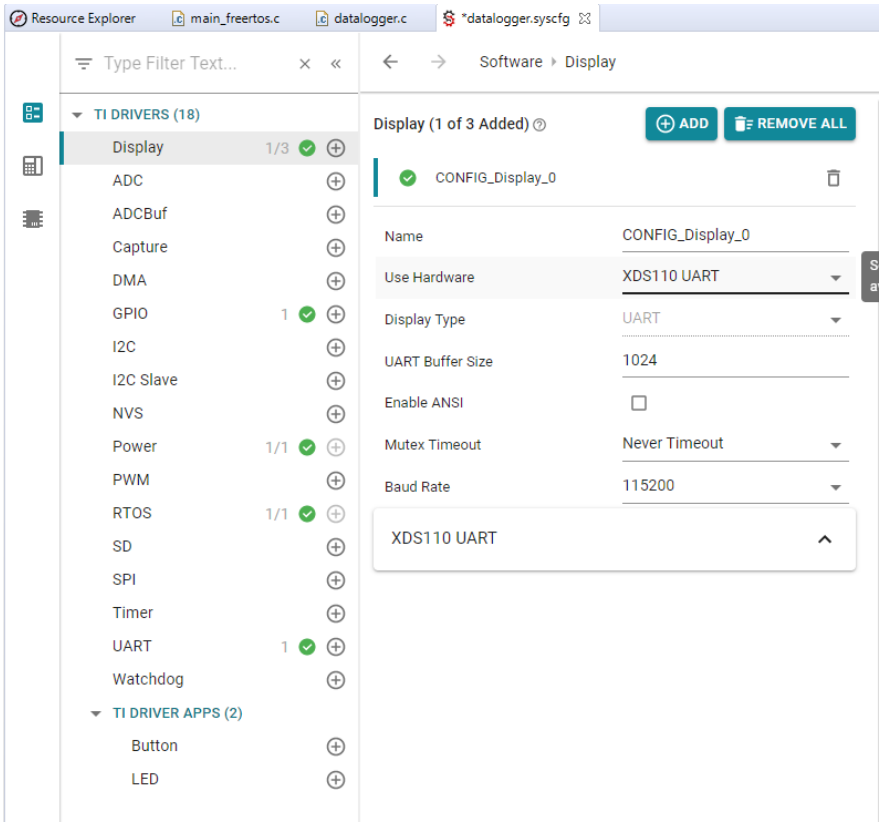
1. **Select the Hardware view.** The hardware components on this LaunchPad are defined under this section. You can see that the `LaunchPad LED 1 RED` is being currently used (green check-mark) and shows the software configuration of it.



Adding new Capability - Output via the Display Module

Let's start using SysConfig to configure new things! First let's add a `Display_printf` to see output via the UART. All SimpleLink LaunchPads have a back-channel UART via the emulation USB cable.

1. **Have the `ti_drivers_config.c` preview open in SysConfig.** We can see the `Display` source code configuration get added when you do the next step!
2. **In the SysConfig go to the Softare view and hit the**
3. In SysConfig, go back to the **Software** view, **click the + icon next to the Display module and then select `XDS110 UART` in the *Use Hardware* drop-down and save the file.** When you hit the + icon, you'll see the `ti_drivers_config.c` file change in the preview.



4. Now let's add the usage of the `Display` module into the `datalogger.c` file.
    - **Add Display.h header**

```
#include <ti/display/Display.h>
```

datalogger.c Add in with other #includes

- **Add initialization of the Display instance.** Also add a count variable that will be used in the `Display_printf`.

  Select text

  ```
  Display_init();
  Display_Handle hSerial = Display_open(Display_Type_UART, NULL);
  uint32_t count = 0;
  ```

  datalogger.c :: mainThread() in the function before the `while` loop.

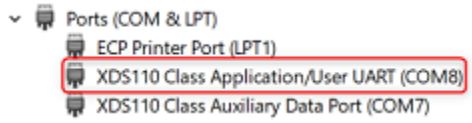
- **Add the `Display_print` call into the `while` loop.**

  Select text

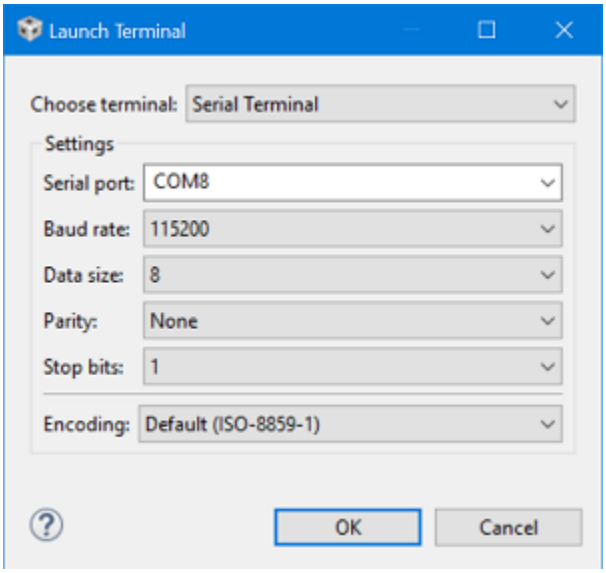
  ```
  while (1) {
     sleep(time);
     GPIO_toggle(CONFIG_GPIO_LED_0);
     Display_printf(hSerial, 1, 0, "LED Toggle %d", count++);
  }
  ```

  datalogger.c :: mainThread() the new while loop

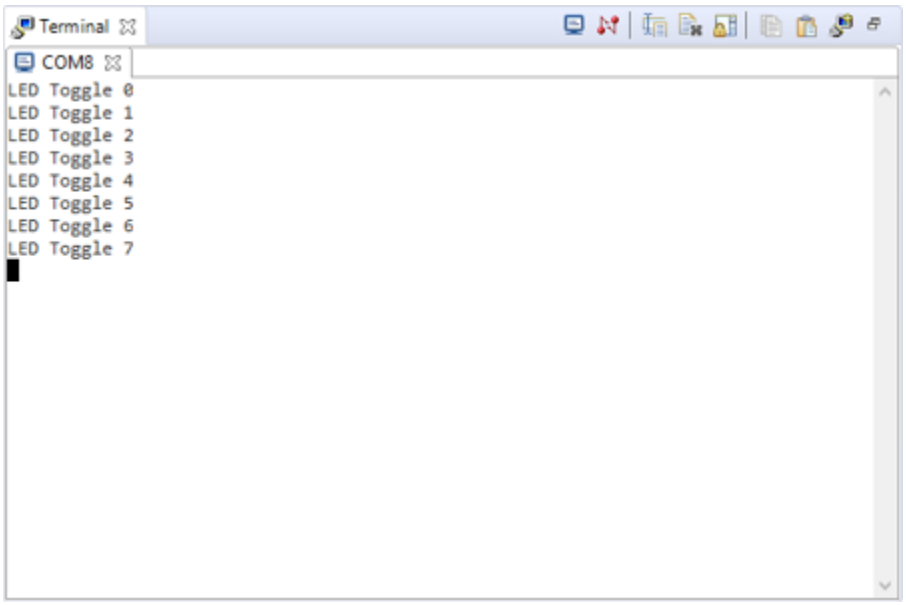
5. **Rebuild/load/run the application.**
6. **Open your favorite terminal application** (e.g. PuTTY, CCS Terminal). You'll probably need to use Device Manager (if using Windows) to determine the correct COM port.



with the following settings (note your COM port will most likely be different!).



You should see the output in the terminal session

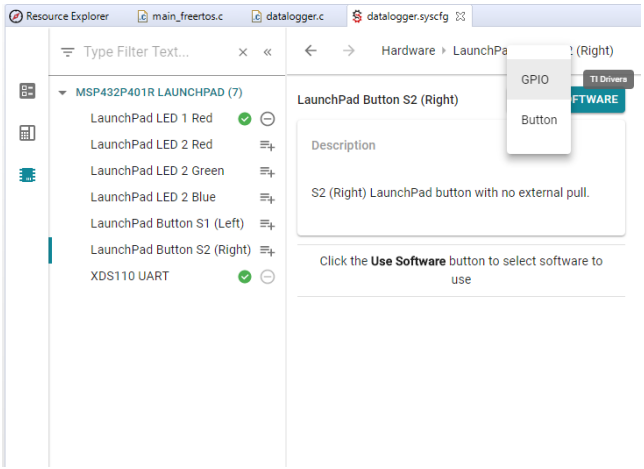


Now you see how to add capability using the SysConfig system.

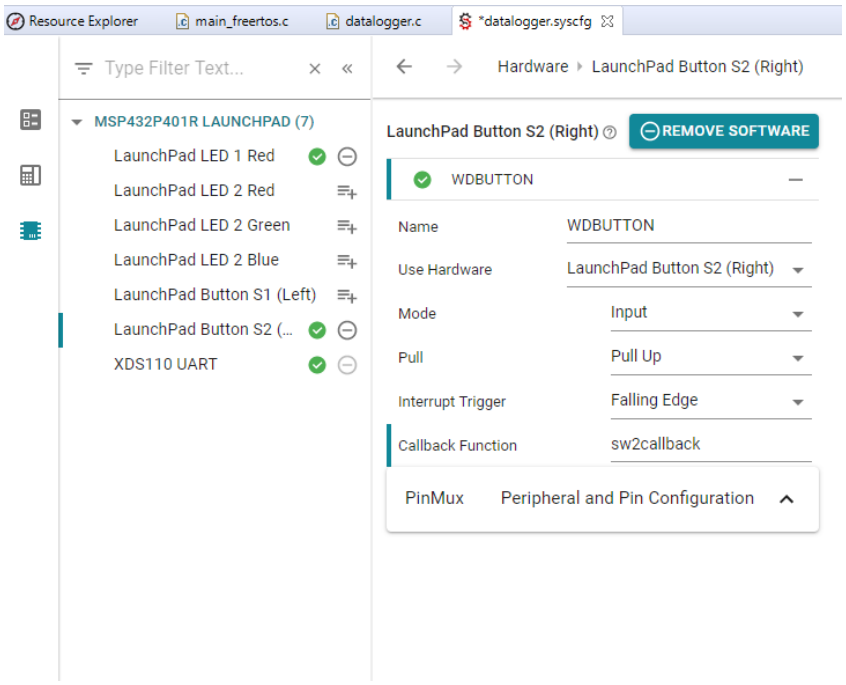
How to Add a button

Let's add a button to the application. When pressed, it will disable the print to the terminal in the `while` loop.

1. **Have the `ti_drivers_config.h` preview open in SysConfig.** We can see the name of the button get modified as we change it!
2. In SysConfig, let's add a button via the Hardware view instead of Software.
    - **Select Hardware** and **click the + icon next to LaunchPad User Button SW2 (Right). Make sure to select GPIO.**



- **Change the name of the constant to WDBUTTON.** Note: you'll see the name change in the `ti_drivers_config.h` preview window.
- **Select `Pull Up` for the *Pull* field and `Failing Edge` for the *Interrupt Trigger*.** Note: your board might have different requirements.
- Finally, **set the button callback to `sw2callback`**



3. Now let's add the usage of the button into the `datalogger.c` file.
   - **Add the enabling of the callback**. Note the name of the constant is from the SysConfig Name setting.

     ```
     GPIO_enableInt(WDBUTTON);
     ```
     datalogger.c after the `GPIO_setConfig`
   - Add a global variable to control the print

     ```
     int setPrint = 1;
     ```

     datalogger.c after the #includes
   - Now use this global to turn on and off the print by adding this to the mainThread function.

     ```
     if (setPrint)
        Display_printf(hSerial, 1, 0, "LED Toggle %d", count++);
     ```
   - **Add the callback** (note the name must match the SysConfig Callback Function setting).

     ```
     void sw2callback(uint_least8_t index)
     {
     ```

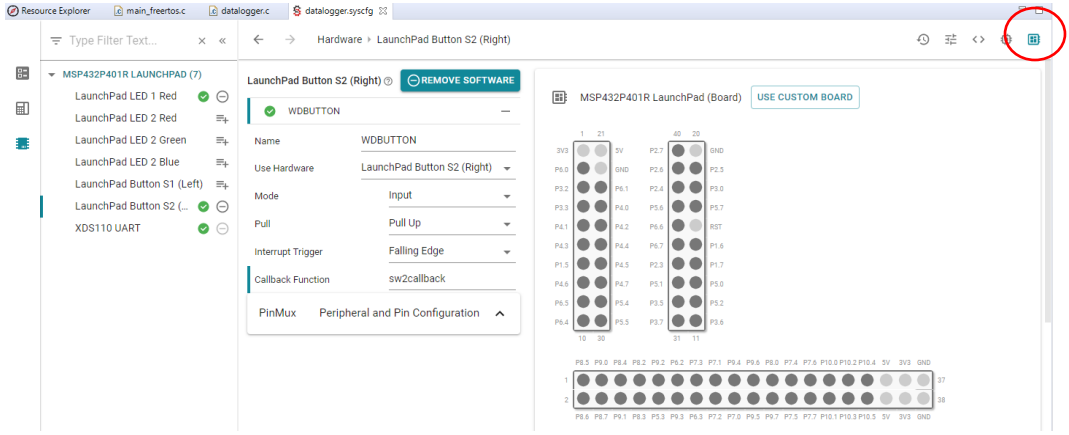
```
}
```

datalogger.c above the mainThread function

4. **Rebuild/load/run the application.**
5. **Hit BTN-1** (or whatever button you specify) refer to Board.html for details. Your output should turn on and off when you press the right switch.

Now it may not happen exactly each time. And the reason refers back to the idea of switch bounce. When you press the switch it can actually send more than one switch event to the system to be processed. There are ways to deal with this, both hardware and software.

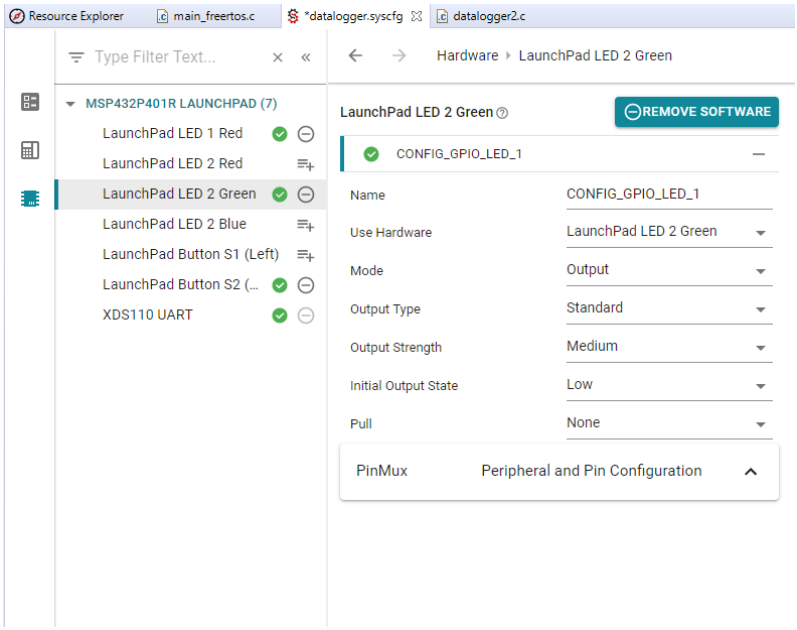
By the way, if you are unfamiliar with the board, you can select the Show Board selection and it will show you the specifics about the output to your board.



Adding another Task to your program

Now that you have a basic system up and working, let's add a second process to your program. To do this let's copy the datalogger.c file, it will ask for a new name, so call it datalogger2.c.

Now let's use the SysConfig to add another LED to our project. In this case use it to create a Green LED with the name CONFIG_GPIO_LED_1:



And we can access this from our new task by modifying the datalogger2.c code. Make it simple, like this:

```
/* For usleep() */
#include <unistd.h>
#include <stdint.h>
#include <stddef.h>

/* Driver Header files */
#include <ti/drivers/GPIO.h>
#include <ti/display/Display.h>
// #include <ti/drivers/I2C.h>
// #include <ti/drivers/SPI.h>
// #include <ti/drivers/UART.h>
// #include <ti/drivers/Watchdog.h>

/* Driver configuration */
#include "ti_drivers_config.h"

/*
 *  ======== mainThread ========
 */
void *mainThread2(void *arg0)
{

    /* Call driver init functions */
    GPIO_init();
    Display_init();
    Display_Handle hSerial = Display_open(Display_Type_UART, NULL);
    uint32_t count = 0;

    // I2C_init();
    // SPI_init();
```

```
    // UART_init();
    // Watchdog_init();

    /* Configure the LED pin */
    GPIO_setConfig(CONFIG_GPIO_LED_1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);


    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_1, CONFIG_GPIO_LED_ON);

    while (1) {
        vTaskDelay(1000);
        GPIO_toggle(CONFIG_GPIO_LED_1);
        Display_printf(hSerial, 1, 0, "LED Toggle %d", count++);
    }

}
```

This should toggle the Green LED roughly every second. The vTaskDelay function is significant, it forces the task to be put on the blocked queue for the designated time.

You'll also need to modify the code in datalogger.c:

```
/* For usleep() */
#include <unistd.h>
#include <stdint.h>
#include <stddef.h>

/* Driver Header files */
#include <ti/drivers/GPIO.h>
#include <ti/display/Display.h>
// #include <ti/drivers/I2C.h>
// #include <ti/drivers/SPI.h>
// #include <ti/drivers/UART.h>
// #include <ti/drivers/Watchdog.h>

/* Driver configuration */
#include "ti_drivers_config.h"

int setPrint = 1;

void sw2callback(uint_least8_t index)
{
    if (setPrint)
        setPrint = 0;
    else
        setPrint = 1;
 }

/*
 *  ======== mainThread ========
 */
void *mainThread(void *arg0)
{
```

```
    /* Call driver init functions */
    GPIO_init();
    Display_init();


    // I2C_init();
    // SPI_init();
    // UART_init();
    // Watchdog_init();

    /* Configure the LED pin */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

    GPIO_enableInt(WDBUTTON);

    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);

    while (1) {
        vTaskDelay(400);
        GPIO_toggle(CONFIG_GPIO_LED_0);

    }
}
```

You'll note that you removed the print code, you won't want that now in a system where we are trying to toggle LEDs at a fairly high rate.

Now while you have created a second function, it is not yet a task. You'll need to modify the main_freertos.c to add the task. Your code should look like this:

```
/* Driver configuration */
#include <ti/drivers/Board.h>

extern void *mainThread(void *arg0);
extern void *mainThread2(void *arg0);

/* Stack size in bytes */
#define THREADSTACKSIZE   1024

/*
 *  ======== main ========
 */
int main(void)
{
    pthread_t           thread;
    pthread_attr_t      attrs;
    struct sched_param  priParam;
    int                 retc;

    pthread_t           thread1;
    pthread_attr_t      attrs1;
    struct sched_param  priParam1;
```

```
    /* initialize the system locks */
#ifdef __ICCARM__
    __iar_Initlocks();
#endif

    /* Call driver init functions */
    Board_init();

    /* Initialize the attributes structure with default values */
    pthread_attr_init(&attrs);

    /* Set priority, detach state, and stack size attributes */
    priParam.sched_priority = 1;
    retc = pthread_attr_setschedparam(&attrs, &priParam);
    retc |= pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    retc |= pthread_attr_setstacksize(&attrs, THREADSTACKSIZE);
    if (retc != 0) {
        /* failed to set attributes */
        while (1) {}
    }

    retc = pthread_create(&thread, &attrs, mainThread, NULL);
    if (retc != 0) {
        /* pthread_create() failed */
        while (1) {}
    }

    /* Initialize the attributes structure with default values */
    pthread_attr_init(&attrs1);

    /* Set priority, detach state, and stack size attributes */
    priParam1.sched_priority = 1;
    retc = pthread_attr_setschedparam(&attrs1, &priParam1);
    retc |= pthread_attr_setdetachstate(&attrs1, PTHREAD_CREATE_DETACHED);
    retc |= pthread_attr_setstacksize(&attrs1, THREADSTACKSIZE);
    if (retc != 0) {
        /* failed to set attributes */
        while (1) {}
    }

    retc = pthread_create(&thread1, &attrs1, mainThread2, NULL);
    if (retc != 0) {
        /* pthread_create() failed */
        while (1) {}
    }
    /* Start the FreeRTOS scheduler */
    vTaskStartScheduler();

    return (0);
}
```

Note that this is not the entire file, just the area that needs to be changed. You are now creating two processes, each with its own LED. Note that they are set to the same priority.

Now the final step is to toggle the first LED only when the switch is pressed. To do this we need to introduce a synchronization semaphore. So you will now add this code to datalogger.c:

```
/* For usleep() */
#include <unistd.h>
#include <stdint.h>
#include <stddef.h>
#include <FreeRTOS.h>
#include <semphr.h>
/* Driver Header files */
#include <ti/drivers/GPIO.h>
#include <ti/display/Display.h>
// #include <ti/drivers/I2C.h>
// #include <ti/drivers/SPI.h>
// #include <ti/drivers/UART.h>
// #include <ti/drivers/Watchdog.h>

/* Driver configuration */
#include "ti_drivers_config.h"
SemaphoreHandle_t xMutex;


void sw2callback(uint_least8_t index)
{
    xSemaphoreGiveFromISR( xMutex, NULL);
 }

/*
 *  ======== mainThread ========
 */
void *mainThread(void *arg0)
{

    /* Call driver init functions */
    GPIO_init();
    xMutex = xSemaphoreCreateBinary();

    // I2C_init();
    // SPI_init();
    // UART_init();
    // Watchdog_init();

    /* Configure the LED pin */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

    GPIO_enableInt(WDBUTTON);

    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);

    while (1) {
        xSemaphoreTake( xMutex, portMAX_DELAY );
        vTaskDelay(400);
        GPIO_toggle(CONFIG_GPIO_LED_0);
```

```
    }
}
```

At the top of the file you'll need to include the semaphore capabilities, then you create a Binary semaphore (this is the kind you'll use for synchronization.) Then you'll take the Semaphore each time through the while loop. When the semaphore isn't available you'll be placed on the blocked queue until the semaphore is available. This will happen with the ISR gives up the semaphore.

When you compile and run this code you should see the green LED flashing at a 1 second rate, and the RED led will change state when you press the switch on the right hand side of the board.

Lab Submission:

Create the code and then capture a video of your project working correctly. Submit this on i-learn.