

CC1350 TI-RTOS/RF LABS – LAB 2

Tasks to complete:

Complete Lab 2 of the CC1350 LAB document. For each task provide snapshot of the modifications in the cfg file XGCONF, Stack Usage for each case, log variables using ROV, execution graphs using RTOS Analyzer, snapshot of GUI and UART, and videos of the demo (both board and GUI).

Follow the submission guideline to be awarded points for this Lab.

Submit the following for all Labs:

1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder CC1350-LABS, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.
3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (with all snapshots requested), b) source code file(s) with all include files, c) text file with youtube video links (see template).

GUI Composer Project Zero | JTAG/XDS

Introduction

This workshop is a continuation of the [TI Drivers Project Zero](#) exercise. We will expand on that example by creating a simple PC-side graphical user interface (GUI) that complements our example. If you remember, we created a simple application that continually reads an ADC channel & toggles an LED on or off depending on whether or not our ADC reading exceeds a threshold.

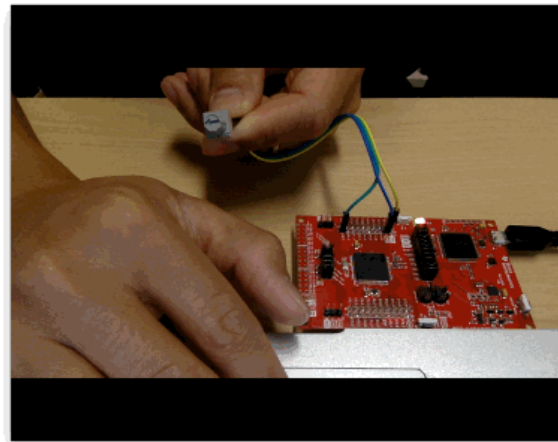
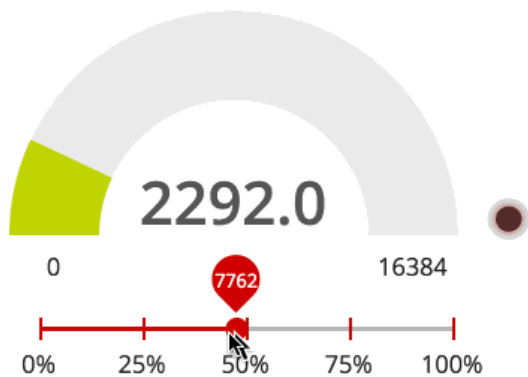
To do this, we will use a nifty tool called GUI Composer. GUI Composer is a browser-based utility for creating PC-side graphical interfaces for interacting with your hardware. GUI Composer supports several interfaces today, including Serial/UART, JTAG/XDS, or MQTT (for IoT applications).

For this demo, we will use the JTAG/XDS interface. We will visualize our latest ADC readings in an analog gauge & will add a horizontal slider for setting dynamically setting the threshold that triggers the LED to turn on/off.

Here's what we'll learn:

- Get introduced to TI's GUI Composer
- Create a simple analog gauge for visualizing ADC values
- Create a horizontal slider so threshold can be dynamically set
- Add an "LED" indicator to the GUI that mirrors the physical LED on our LaunchPad kit

Preview of the GUI we're making:



What is GUI Composer

GUI Composer is a browser-based "What you see is what you get" (WYSIWYG) tool for developing PC-side HTML-based graphical user interfaces (GUIs) that can complement your embedded project/application. With this tool, users will be able to drag & drop various GUI elements into a sandbox to build their interface, including gauges, dials, sliders, line charts & more.

Features:

- Allows you to build HTML-based GUIs graphically (simple drag & drop interface)
- Supports communication with the target device via USB (serial I/O), XDS Debug Port or Internet (MQTT / IoT)
- Wide variety of components to choose from, including graphs, gauges, dials, buttons, menus, meters, and many more
- Components configured via easy-to-use properties
- Full JavaScript editor provided for advanced users.


What interfaces does GUI Composer support?

GUI Composer supports various data transport interfaces. The following are supported:

- Serial/UART
- **JTAG/XDS Debug interface (This is the interface we will use for this training)**
- MQTT (Internet of Things)

Prerequisites

Recommended material

- [TI Drivers Project Zero](#) 
- In this exercise, we will modify the output of that previous lab to leverage GUI Composer.

Software for desktop development

- This tutorial can be done 100% with a web browser in the cloud. We will use CCS Cloud to make small modifications to an existing example & will use GUI Composer to build a simple graphical interface.
- However, the exercises can also be completed using desktop/offline tools as well. If you want to run the exercises offline, you will need to download & install the following:
 - CCS 7.0+
 - SimpleLink SDK for your given LaunchPad

Hardware requirements

- A SimpleLink MCU LaunchPad Development Kit

Recommended Reading

- GUI Composer User's Guide: <https://dev.ti.com/gc/designer/help/UsersGuide/index.html>

Task 1 - Setting up code example for GUI Composer

1. Using a JTAG/XDS Debug interface to exchange data with your GUI

When using the JTAG/XDS Debug interface, GUI Composer can use your firmware's symbolic information to bind widgets. This essentially allows you to bind your application's global variables to GUI Composer widgets. In this training, we'll learn how to upload your project's .out file into GUI Composer, where it will parse the symbols within your application. We will then be able to bind our application's variables to various GUI elements. This mode requires the device to support non-intrusive memory access. There is no overhead on application processing and there is no code that needs to be added to your application. This option is available on all SimpleLink MCUs.

GUI Composer uses bindings to the ti-program-model component to allow widgets to be automatically updated to reflect the value of target-side global variables when those variables change, and to update the target-side global variable when the widget is configured by the user. For example, you can bind ADC readings to an analog gauge widget, or bind a variable within your application that is altered by a GUI Composer slider.

2. Let's turn the variables we want to visualize & modify into global variables

In this exercise, we want to build a simple GUI that does the following things:

- Visualize our latest ADC readings using an analog gauge
- Get an "LED" notification when we are above/below a threshold
- Feature a slider that allows us to modify the threshold for the LED alert

This means we need a global variable for our ADC reading & a global variable for our threshold. We will also need a global variable that tells us if we're above or below the threshold. To do this, we need to make a few modifications to the example project we created in the ***TI Drivers Project Zero*** lab.

Make adcValue a global variable

First, let's make adcValue a global variable. We need to get rid of the local declaration we have in our while(1) loop, then add a global declaration to the top of our code. We will eventually bind this global variable to an analog gauge in the GUI to visualize our latest readings.

Create a global variable for threshold

Second, we need to make our threshold to be determined by a variable. Currently, our threshold is hard-coded at 100. Let's define a new global variable called "threshold" and replace our hardcoded limit of 100. We will eventually bind this global variable with a slider to enable the threshold to be altered via the GUI.

Create a global variable for alert

Lastly, we need to create a new global variable for our alert. This will be '1' if our ADC reading is above the threshold, or '0' if it is below. We can use this global variable to feed the LED indicator in the GUI we are about to build.

Let's also make our variables update more frequently

To do this, we will use the `usleep()` API instead of the `sleep()` API. This allows us to pass in the number of microseconds we want to sleep for.

We should end up with the following

Select text

```
/*
 * ===== empty.c =====
 */

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

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

/* Board Header file */
#include "Board.h"

/* global variables FOR GUI COMPOSER */
uint16_t adcValue = 0;
uint16_t threshold = 100;
uint16_t trigger = 0;

/*
 * ===== mainThread =====
 */

void *mainThread(void *arg0)
{
    /* ~10 loops/second */
    uint32_t time = 100000; // update ~10/second

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

    /* Open ADC Driver */
    ADC_Handle adc;
    ADC_Params params;
    ADC_Params_init(&params);
    adc = ADC_open(Board_ADC0, &params);
    if (adc == NULL) {
        // Error initializing ADC channel 0
    }
}
```

```

        while (1);
    }

    /* Open Display Driver */
    Display_Handle    displayHandle;
    Display_Params    displayParams;
    Display_Params_init(&displayParams);
    displayHandle = Display_open(Display_Type_UART, NULL);

    while (1) {
        int_fast16_t res;
        res = ADC_convert(adc, &adcValue);
        if (res == ADC_STATUS_SUCCESS) {
            Display_printf(displayHandle, 1, 0, "ADC Reading %d", adcValue);

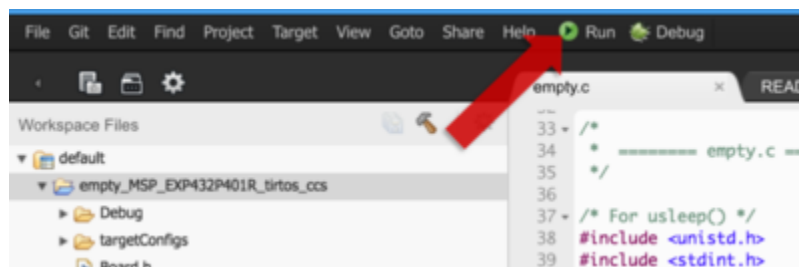
            if(adcValue >= threshold){
                GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
                trigger = 1;
            } else{
                GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
                trigger = 0;
            }
        }

        usleep(time);
    }
}

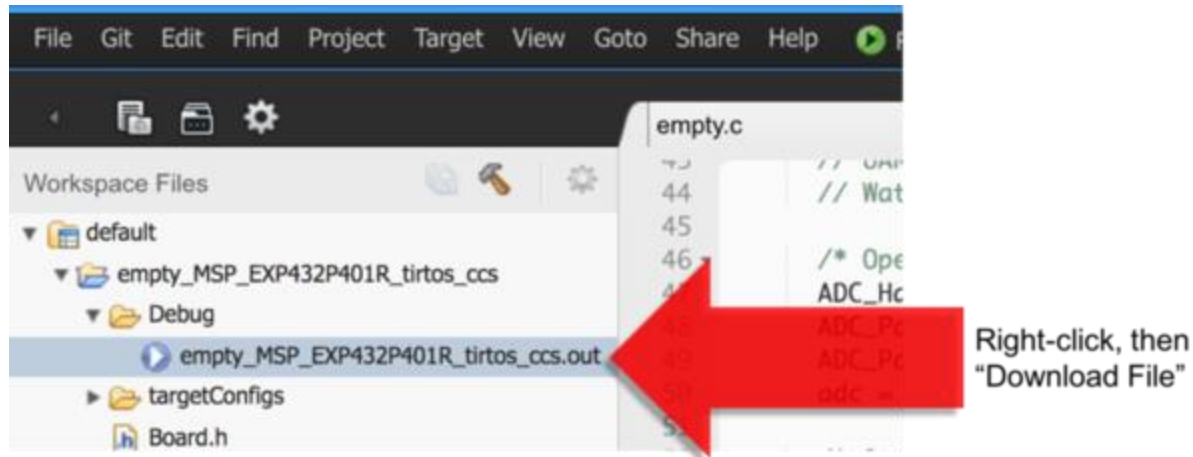
```

3 Let's compile our code to generate a .out file

Once we've made the small changes to our project, we can go ahead and compile our project. This will generate a .out file, which we can ultimately pass on to GUI Composer. With the .out file, GUI Composer will be able to use the symbolic information to extract the global variables within your application. Clicking the "run" button in CCS Cloud will compile our code to generate a .out file & will also flash our LaunchPad.



Once the project is built, we can find the .out file by going to the Project Explorer/Workspace Files window within your IDE. In CCS Cloud, you will find it in the Debug folder within your project. Right-click the file to download it locally.



Task 2 - Creating our first GUI!

1. Launch GUI Composer & Create new GUI project!

Now that we have our .out file, we can launch GUI Composer by navigating to <http://dev.ti.com/gc>

Select "CREATE A NEW PROJECT" to launch the "New Project Wizard."

Let's use the following parameters:

- Project Template: Application
- Project Name: GUI Composer Demo
- Application Name: GUI Composer Demo
- Target Communications: XDS Debug Port / Target Monitor
- Enable TI-Branding: [Check]
- Then, press NEXT >>

2. The XDS / TARGET MONITOR CONFIGURATION window

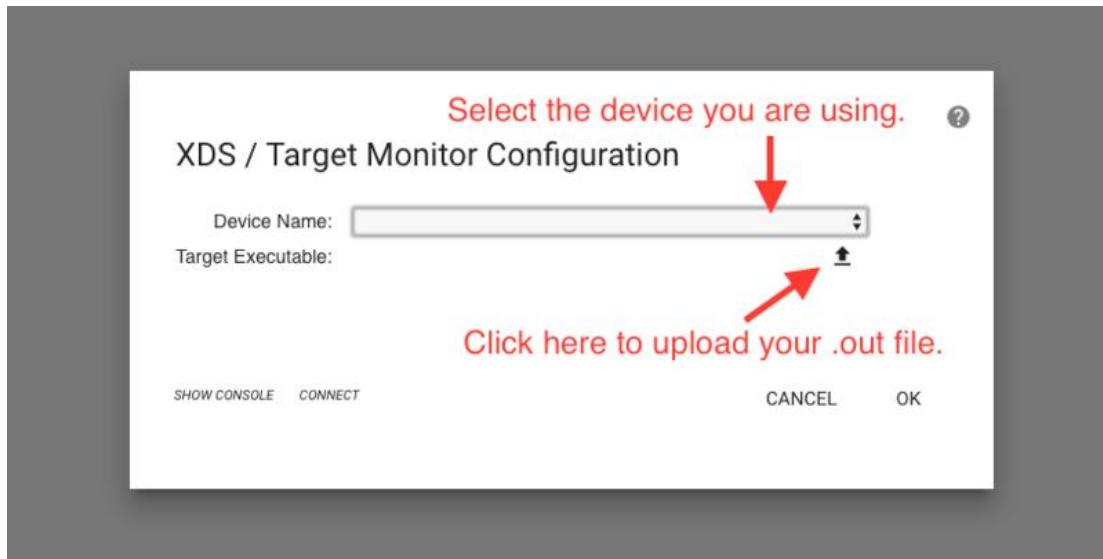
We need to tell GUI Composer which device we are using as well as upload the .out file of our firmware. This will allow GUI Composer to un-intrusively read/modify global variables without any additional code required in our firmware.

Select the device you are using

In the development of this tutorial, we used the MSP-EXP432P401R LaunchPad, so we will select MSP432P401R. Be sure to select the device you are developing with.

Upload your .out file

Click the upload button & navigate to the .out file that we downloaded from CCS Cloud.



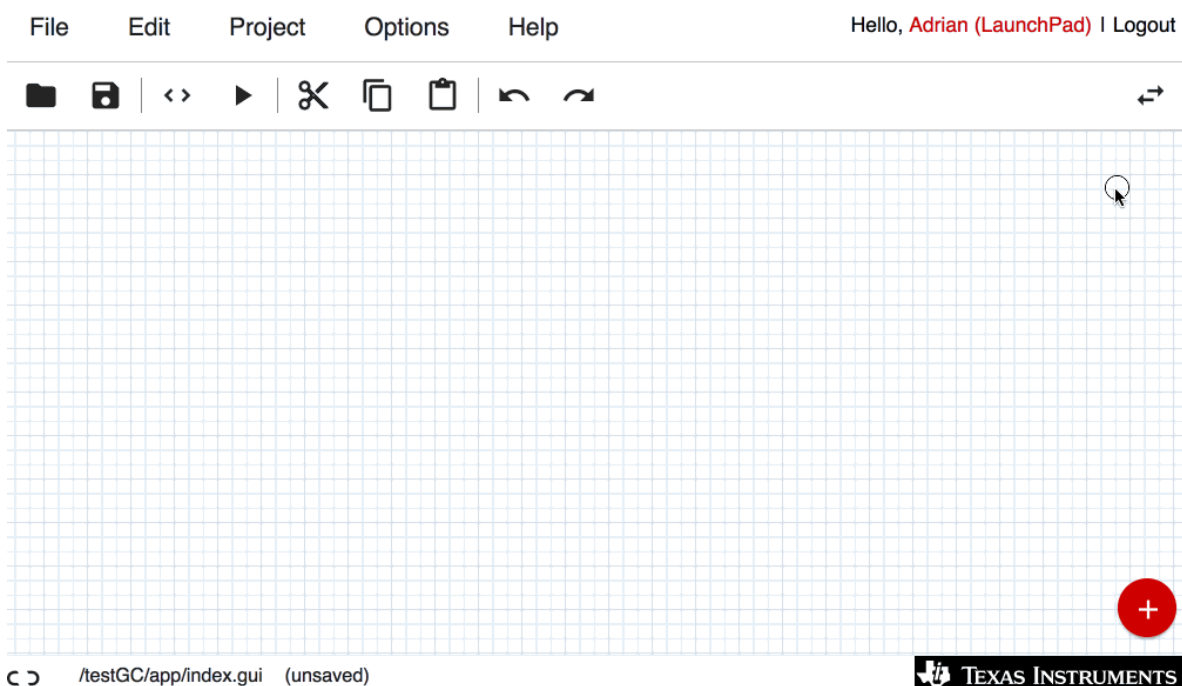
Press OK

3. Adding elements to your GUI

At this point, we should have a blank window to start developing our GUI. We can search the Palette of available GUI elements. For this demo, we need 3 different elements:

- Dials & Gauges > Analog Meter (for our ADC readings)
- Common Widgets > Horizontal Slider (for modifying our threshold)
- Status Indicators > LED (to indicate if ADC reading is above/below threshold)

Search for these items in the GUI Palette & drag them into the GUI editor window. You should end up with the following:



4. Editing GUI elements with Properties & Styles panel

Each GUI element can be modified & edited using the Properties & Styles panel. Click on the GUI element you want to modify, then simply edit the parameters in the side configuration pane.

For this tutorial, we need to make a few small adjustments

Analog meter

- Change the max-value to the maximum digital number your device's ADC resolution can support. The MSP432P401R device we are using has a 14-bit Analog to Digital Converter, which can give us 2^{14} unique readings. So we'll put 16,383 as our max-value.

Horizontal slider

- We want to change the labels here to reflect percentage, so let's change the "labels" parameter to: 0%, 25%, 50%, 75%, 100%
- We also need to change the max-value of the slider to match that of the analog meter. In the case of a 14-bit ADC, we will change it to 16,383.

LED

- If we wanted to, we can change the LED color, but we'll go ahead and leave ours red.

5. Binding global variables with GUI widgets

Now that our GUI elements are configured, we can go ahead & bind the global variables to them. To do this, we will again modify the PROPERTIES pane for each widget.

Binding Analog meter

Click the analog meter widget to make it "active." Click the "bind" icon next to the "value" field. This will introduce a drop down menu that is pre-set for "target_device." Leave that as-is. In the empty text field to the right, we will type the name of the global variable that we want to bind. In this case, we want to bind the global variable `adcValue`



Binding Horizontal slider

Click the horizontal slider widget to make it "active." Click the "bind" icon next to the "value" field. Again, leave "target_device" drop down as-is & type in the global variable we want to bind to the horizontal slider. In this case, we want to bind the global variable `threshold`



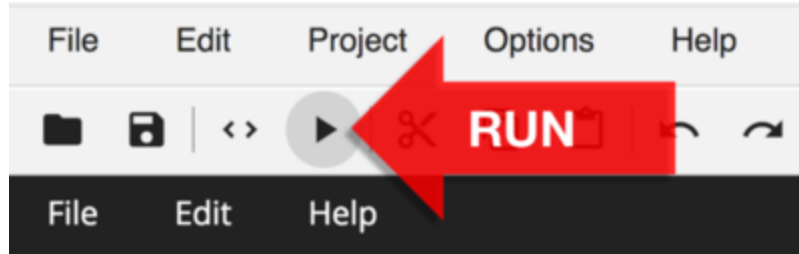
Binding LED indicator

Click the LED indicator widget to make it "active." Click the "bind" icon next to the "on" field. Again, leave "target_device" drop down as-is & type in the global variable we want to bind to the LED indicator. In this case, we want to bind the global variable `trigger`



6. Let's run our GUI!

And that's it! Now that we've successfully built our GUI, we can go ahead and run our GUI by clicking the "play" button.



This will open up your GUI in a new tab. At this point, your GUI will start to connect to your LaunchPad, which is running the same firmware that we uploaded into GUI Composer.

At this time, you should see the analog meter updating appropriately. You can also slide the horizontal slider to change the target threshold. Lastly, the LED indicator in the GUI should match the status of the LED found on your LaunchPad.

7. Exporting your GUI/app

Once you're happy with your GUI, you GUI Composer can export your GUI as a standalone application. You can export your GUI by clicking on **File > Export > as stand-alone app**

This will generate a zip file of your GUI, which can now run standalone.

Lab 2: RTOS Example

Software for desktop development

This tutorial can be done 100% with a web browser in the cloud. However, the exercises can also be completed using desktop/offline tools as well. If you want to run the exercises offline, you will need to download & install the following:

- CCS 7.1+
- SimpleLink SDK for your given LaunchPad

Hardware requirements

- A SimpleLink LaunchPad Development Kit

Recommended Reading

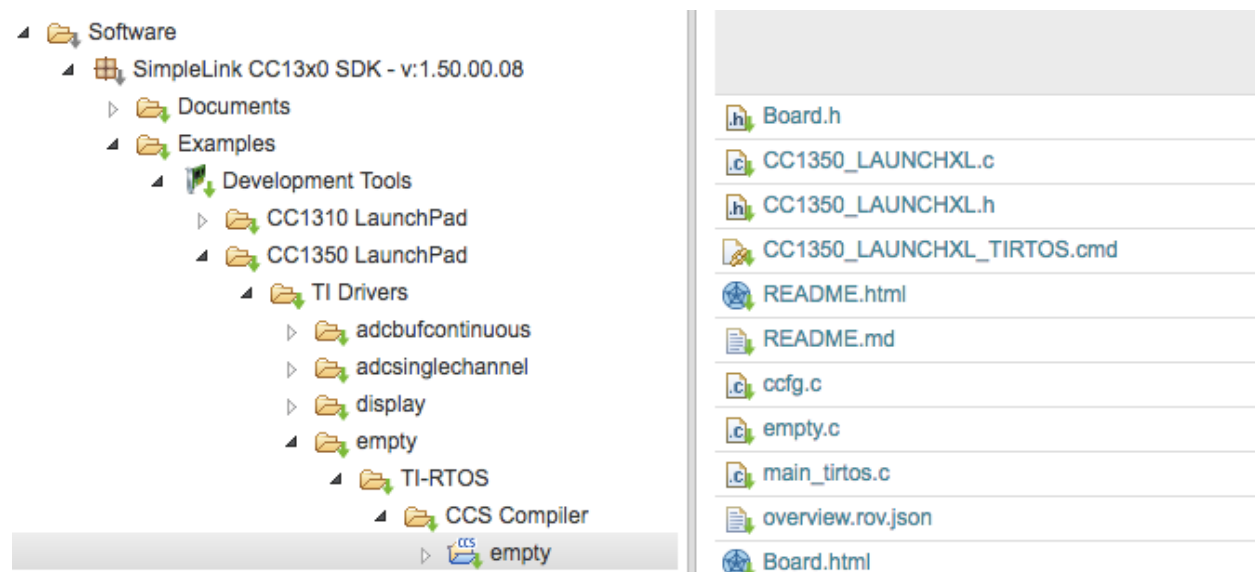
- SimpleLink SDK Quick Start Guide
- SimpleLink SDK User Guide

Task 1 - Finding the Blink LED code example

Let's look at the "empty" example.

The "empty" project is a template file that builds a framework for a new SimpleLink SDK-based project. In short, it creates a single-threaded application using POSIX APIs that toggles an LED pin high & low using TI Driver APIs. Note that there are several versions of each TI Driver example. An example is available based on TI-RTOS (and FreeRTOS for most SimpleLink devices). Additionally, there are variants that use either the CCS or GCC compiler. Select any version you like. In the screenshot below, we are going to use the TI-RTOS variant that uses the CCS compiler.

Software > [SimpleLink SDK] > Examples > Development Tools > [LaunchPad] > TI Drivers > Examples > empty



Taking a closer look at the source files

Let's take a look at the empty.c source file - simply double-click the file to open it up in CCS Cloud. In general, this "empty" project provides you with a framework that you can use as a starting point for your own project. It shows you where to include the header files for the TI Drivers you want to use (only the GPIO driver is used in this example). It also provides a framework for your main loop, which is running inside of a single thread called "mainThread."

Inside of mainThread, we see that we have to initialize the GPIO driver before we can use it by calling the `GPIO_init()` function. Once the driver has been initialized, we can use the driver to set the GPIO high using the `GPIO_write()` API. We also notice that we are using the `Board_PIN_LED0` designator, which was declared in our Board.h file (and is documented in the Board.html file).

Further down mainThread, we see our `while(1)` loop, which is our main loop for this simple example. Within this loop, we use another GPIO Driver API to toggle the pin. We do this once every second, which is determined by the `sleep()` function, which takes a parameter for number of seconds. The `sleep()` function is provided by the unistd.h header that is included in the "empty" project by default.

Another file to take a look at is "main_tirtos.c" or "main_freertos.c" - each example includes one of these files depending on the underlying kernel being used. However, since we are using POSIX in our TI Driver examples, these files are largely identical. Within this file, we configure the kernel, create & configure our thread(s) and set their priorities. For instance, we see that we use the POSIX API `pthread_create()` to create our mainThread, which is what we saw in our main empty.c source file.

Build/compile your source code

Let's load our LaunchPad!

Task2: Let's modify the example so LED is ON only if an ADC reading exceeds a threshold

For this section, we will use the ADC TI Driver to take an ADC sample periodically. If the ADC reading is greater than a threshold, we will turn the LED on, else we leave it off. To learn how to do this, we can use the TI Driver code examples while looking at the TI Driver API guides.

Let's look at the TI Drivers at [\[SimpleLink SDK\] > Documentation > tidrivers > tidriversAPI.html](#)

Click on ADC.h to see the API guide for the ADC TI Driver.

Software

SimpleLink CC13x0 SDK - v:1.50.00.08

Documents

Release Notes

Documentation Overview

TI 15.4 Stack Quick Start Guide

TI 15.4 Stack SDK User's Guide

TI 15.4 Stack Migration Guide

BLE User's Guide

BLE Stack Quick Start Guide

BLE APIs

Proprietary RF Getting Start Guide

Proprietary RF User's Guide

Proprietary RF Examples User's Guide

Drivers

TI Drivers Power Management

TI Drivers Runtime APIs

Kernel

Instrumentation

Shown below is a matrix of available drivers.

NOTE: Please view the **Device Specific** driver implementations as they may contain or exclude certain features that are defined in the top level interfaces.

Driver Interfaces	CC13xx/CC26xx Implementations
ADC.h	ADCCC26XX.h
ADCBuf.h	ADCBufCC26XX.h
Crypto	CryptoCC26XX.h
GPIO.h	GPIOCC26XX.h
GPTimer	GPTimerCC26XX.h
I2C.h	I2CCC26XX.h

Using TI Drivers: Init, Open, Use, Close.

The TI Driver API Guides show how to initialize, open, use & close each of the TI Drivers. In this case, we want to learn about how to use the ADC driver.

1. First off, we see that we need to include the ADC driver header file.
2. Secondly, we see that we have to create a handle for our ADC driver. Once a handle has been created, we have to initialize, then open the driver for use.
3. Once initialized & opened, the driver can be used. In this simple example, we use the `ADC_convert()` API.

For our simple example, that's all we need. However, for full details on the ADC driver, you can scroll down the API guide to learn more about the various functions that are exposed by the driver.

Adding a simple ADC conversion to our "empty" project

1. Let's copy & paste the include header code to include the ADC driver into the top of our empty.c example:

```
#include <ti/drivers/ADC.h>
```

2. Declare ADC conversion result variables
- ```
/* ADC conversion result variables */
uint16_t adcValue0;
uint32_t adcValue0MicroVolt;
```

3. Now that we have the driver added to our project, let's be sure to call `ADC_init()` to initialize the ADC driver. The "empty" project recommends where to place these driver initializations.

```
ADC_init();
```

4. Next, let's create an ADC handle (we can name it whatever we want, but let's stick to "adc"). Let's also initialize & open the driver so we can use it. We can add this code inside of mainThread, right before the while(1) loop.  
We have to make a small change to the code snippet provided by the API guide. We need to change the ADC pin passed into the ADC\_open() API to ensure we are using an available ADC channel. We can again refer to the Board.html resource, or take a look directly at the Board.h file included in the empty project example. Referring to the Board.h file, we see that "Board\_ADC0" is available.

```
ADC_Handle adc;
ADC_Params params;
int_fast16_t res;

ADC_Params_init(¶ms);
adc = ADC_open(Board_ADC0, ¶ms);

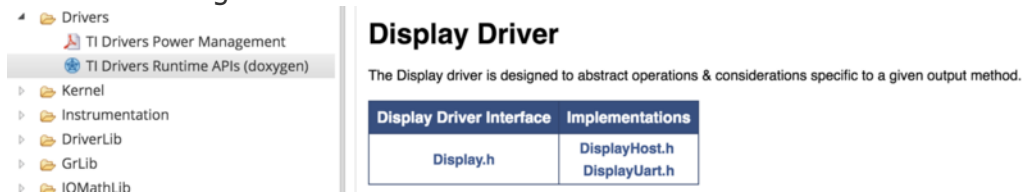
if (adc == NULL) {
 Display_printf(display, 0, 0, "Error initializing ADC channel 0\n");
 while (1);
}
```

5. Finally, let's add the ADC conversion code inside of the while(1) loop to periodically sample the ADC every second. Let's also use a simple if-statement to set the LED high when the ADC result is  $\geq$  an arbitrary threshold of 100, but off when  $<$  threshold. We'll use the GPIO\_write() API we learned about earlier in the simple blink "empty" example. Let's also remove the GPIO\_toggle() from the original empty example.

```
while (1) {
 int_fast16_t res;
 res = ADC_convert(adc, &adcValue);
 if (res == ADC_STATUS_SUCCESS) {
 Display_printf(displayHandle, 1, 0, "ADC Reading %d", adcValue);
 if(adcValue >= threshold){
 GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
 trigger = 1;
 } else{
 GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
 trigger = 0;
 }
 }
 usleep(time);
}
```

## Task 3: Adding a serial UART transmission to report ADC readings

A Display Driver is available, which offers a consistent API set for displaying data across multiple mediums (Serial UART, LCD, etc.). In this case, we will use the Display API to send our ADC readings back to a terminal window.



We can learn more about the Display Driver within the TI Driver API Guide. Click on Display.h to learn about how to use the driver to send data over a serial UART.

Add the required header file to the top of our empty.c source file:

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

Display Driver header.

Just like the ADC driver, we see that we need to include the appropriate header file. We also see that we need to initialize & open the driver before we can use it.

Next, we need to create a handle for our Display driver, and need to initialize & open the driver. Note that we need to use the "Display\_Type\_UART" parameter when we open the Display driver to use it for serial communication.

```
Display_Handle displayHandle;
Display_Params displayParams;

Display_Params_init(&displayParams);
handle = Display_open(Display_Type_UART, NULL);
```

Ultimately, we can use the driver & close it when we no longer need it.

We can now use the Display\_printf() API to send data over serial UART. Let's send our ADC readings after every ADC conversion.

```
Display_printf(displayHandle, 1, 0, "ADC Reading %d", adcValue);
```



## Task 4 - Adding GPIO interrupts to our base example.

The SimpleLink™ SDK includes a simple code example for learning how to setup a simple GPIO interrupt. You can find it using TI Resource Explorer by navigating to:

'[Simplelink SDK] > Examples > Development Tools > [Your Hardware kit] > TI Drivers > gpinterrupt'

Reading the associated README.html provided with the code example, we see that this example simply toggles an LED when a button is pressed.

In general, there are a few things to do when creating a GPIO interrupt within your application.

### Need to include GPIO driver header file & initialize it

We first need to ensure that the GPIO driver header file is included. This should already be the case in our example code because we are already using the GPIO driver to control the LEDs.

```
#include <ti/drivers/GPIO.h>
```

We also need to initialize the driver. But again, this has already been done since we're using the GPIO driver for the LEDs.

```
GPIO_init();
```

### Callback functions for your interrupt

What do you want to accomplish when your interrupt is triggered? In general, you want your interrupts to be short and sweet. In the case for the `gpio interrupt` example, they simply toggle an LED & increment a global variable `count`. You'll also notice that they have 2 callback functions, one for each button (if your hardware has 2 buttons).

For this exercise, we want to raise/lower our alert `threshold` in our callback function. We'll need 2 callback functions, one for the left-button on our LaunchPad, and another for the right-button on our LaunchPad.

The callback function for the left-button should decrement the global variable `threshold`. It should also include some basic logic to ensure that `threshold` never becomes a below-zero value.

The callback function for the right-button should increment the global variable `threshold`. It should also include some basic logic to ensure that `threshold` never becomes larger than the maximum reading your device's ADC resolution supports.

With this in mind, let's add in 2 callback functions to the top of our project:

```
/*
 * ===== gpioButtonFxn0 =====
 * Callback function for the GPIO interrupt on Board_GPIO_BUTTON0.
 */
void gpioButtonFxn0(uint_least8_t index)
{
 /* Clear the GPIO interrupt and decrement threshold */
 if(threshold < 250){ // Ensure threshold doesn't go below zero
 threshold = 0;
 } else {
 threshold -= 250; // decrement by 250
 }
}
```

```

}

/*
 * ===== gpioButtonFxn1 =====
 * Callback function for the GPIO interrupt on Board_GPIO_BUTTON1.
 * This may not be used for all boards.
 */
void gpioButtonFxn1(uint_least8_t index)
{
 /* Clear the GPIO interrupt and increment threshold */
 if(threshold > 16133){ // Ensure threshold doesn't go above max ADC range
 threshold = 16383;
 } else {
 threshold += 250; // increment by 250
 }
}
}

```

## Install your interrupt call back function

Once our callback functions are created, we need to install it using the `GPIO_setCallback()` function. We pass in 2 parameters, the IO that will trigger the interrupt & the callback function that we want to execute when the interrupt occurs. We need to do this for each callback function.

```

/* install Button callback */
GPIO_setCallback(Board_GPIO_BUTTON0, gpioButtonFxn0);
GPIO_setCallback(Board_GPIO_BUTTON1, gpioButtonFxn1);

```

## Enable your interrupt

Now that our callback functions have been created & installed, we simply need to enable interrupts. We need to enable both interrupts using the `GPIO_enableInt()` API.

```

/* Enable interrupts */
GPIO_enableInt(Board_GPIO_BUTTON0);
GPIO_enableInt(Board_GPIO_BUTTON1);

```

## Final Code

```
/*
 * ===== empty.c =====
 */

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

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

/* Board Header file */
#include "Board.h"

/* GLOBAL VARIABLES FOR GUI COMPOSER */
uint16_t adcValue = 0;
uint16_t threshold = 100;
uint16_t trigger = 0;

/*
 * ===== gpioButtonFxn0 =====
 * Callback function for the GPIO interrupt on Board_GPIO_BUTTON0.
 */
void gpioButtonFxn0(uint_least8_t index)
{
 /* Clear the GPIO interrupt and decrement threshold */
 if(threshold < 250){ // Ensure threshold doesn't go below zero
 threshold = 0;
 } else {
 threshold -= 250; // decrement by 250
 }
}

/*
 * ===== gpioButtonFxn1 =====
 * Callback function for the GPIO interrupt on Board_GPIO_BUTTON1.
 * This may not be used for all boards.
 */
```

```

*/
void gpioButtonFxn1(uint_least8_t index)
{
 /* Clear the GPIO interrupt and increment threshold */
 if(threshold > 16133){ // Ensure threshold doesn't go above max ADC range
 threshold = 16383;
 } else {
 threshold += 250; // increment by 250
 }
}

/*
 * ===== mainThread =====
 */

void *mainThread(void *arg0)
{
 /* ~10 loops/second */
 uint32_t time = 100000;

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

 /* Open Display Driver */
 Display_Handle displayHandle;
 Display_Params displayParams;
 Display_Params_init(&displayParams);
 displayHandle = Display_open(Display_Type_UART, NULL);

 /* Open ADC Driver */
 ADC_Handle adc;
 ADC_Params params;
 ADC_Params_init(¶ms);
 adc = ADC_open(Board_ADC0, ¶ms);
 if (adc == NULL) {
 // Error initializing ADC channel 0
 while (1);
 }

 /* install Button callback */

```

```

GPIO_setCallback(Board_GPIO_BUTTON0, gpioButtonFxn0);
GPIO_setCallback(Board_GPIO_BUTTON1, gpioButtonFxn1);

/* Enable interrupts */
GPIO_enableInt(Board_GPIO_BUTTON0);
GPIO_enableInt(Board_GPIO_BUTTON1);

while (1) {
 int_fast16_t res;
 res = ADC_convert(adc, &adcValue);
 if (res == ADC_STATUS_SUCCESS) {
 Display_printf(displayHandle, 1, 0, "ADC Reading %d", adcValue);

 if(adcValue >= threshold){
 GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
 trigger = 1;
 } else{
 GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
 trigger = 0;
 }
 }

 usleep(time);
}
}

```

## Program our LaunchPad

Now that we've added a few simple GPIO interrupts, let's go ahead and program our LaunchPad. Let's also run our GUI Composer interface that we created in the "GUI Composer : Project Zero" exercise. We should now be able to modify the threshold value using either the graphical slider in GUI Composer, or by clicking the left/right buttons on your LaunchPad kit.

Programming your LaunchPad will also generate a new .out file that we can upload into GUI Composer for visualizing our global variables.