

CC1350 TI-RTOS/RF LABS – LAB 4

Tasks to complete:

Complete Lab 4 of the CC1350 LAB document. You'll do this lab in pairs for Bonus Task 2. For each task provide snapshot of all GUI (settings) 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).

Introduction

This is the entry level guide on how to use the Sensor Controller with the GUI tool, Sensor Controller Studio (SCS). You will generate, download and debug code on the Sensor Controller processor (16-bit custom low power RISC processor). The Sensor Controller can access the peripherals which reside in the Sensor Controller Domain (AUX Domain).



Compatible Connected MCU LaunchPad kits

This workshop can be completed with any one of the SimpleLink™ Wireless MCU with Sensor Controller devices described in the table below. Install the required Associated SimpleLink Software Development Kit matching your device. More details on LaunchPads please visit the LaunchPad overview page (<http://www.ti.com/tools-software/launchpads/launchpads.html>).

Abbreviations / terminology

Abbreviation / terminology	Definition
CCS	Code Composer Studio
SCS	Sensor Controller Studio
RTC	Real-Time Clock
RTOS	Real-Time Operating System
TI-RTOS	RTOS for TI microcontrollers
SDK	Software Development Kit
HW	Hardware

Prerequisites

Software

In order to start with this exercise you will need to download the correct Software Development Kit (SDK) for your LaunchPad. This training covers the CC13x0, CC13x2, CC2640R2 and the CC26x2 device families.

Device	SDK downloads
CC13x0	SimpleLink CC13x0 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK)
CC13x2	SimpleLink CC13x2 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC13X2-SDK)
CC2640R2	SimpleLink CC2640R2 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC2640R2-SDK)
CC26x2	SimpleLink CC26x2 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC26X2-SDK)

The following software applies for all device families:

- Code Composer Studio (<http://www.ti.com/tool/CCSTUDIO>) version 8.1.0 or higher
Make sure that CCS is using the latest updates: *Help → Check for Updates*
- Sensor Controller Studio (<http://www.ti.com/tool/SENSOR-CONTROLLER-STUDIO>) version 2.2.0 or higher

Hardware

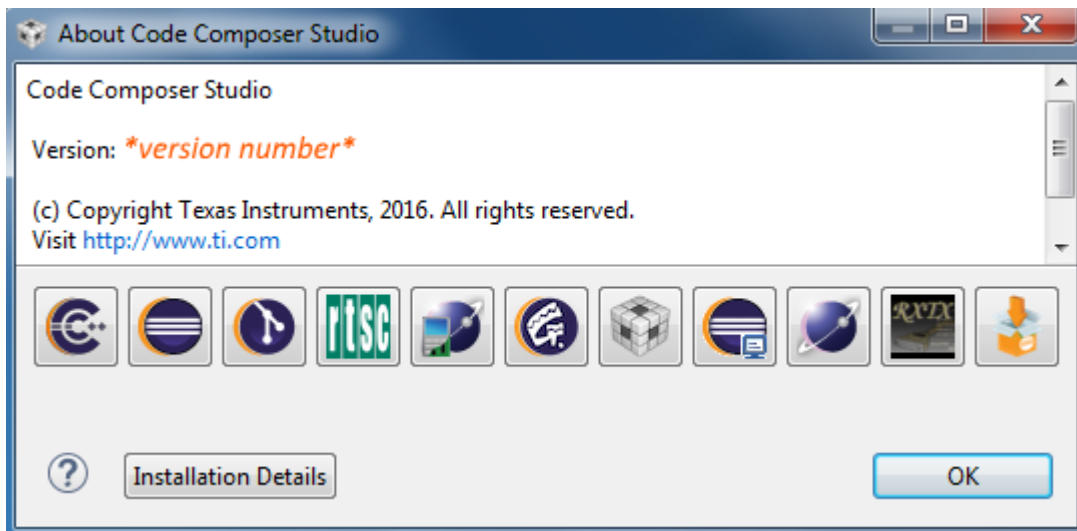
One LaunchPad connected with a USB micro cable:

- LAUNCHXL-CC1310 (<http://www.ti.com/tool/LAUNCHXL-CC1310>),
- LAUNCHXL-CC1312R1 (<http://www.ti.com/tool/LAUNCHXL-CC1312R1>),
- LAUNCHXL-CC1350 (<http://www.ti.com/tool/LAUNCHXL-CC1350>),
- LAUNCHXL-CC1352R1 (<http://www.ti.com/tool/LAUNCHXL-CC1352R1>),
- LAUNCHXL-CC1352P (<http://www.ti.com/tool/LAUNCHXL-CC1352P>),
- LAUNCHXL-CC26x2R1 (<http://www.ti.com/tool/LAUNCHXL-CC26X2R1>), or
- LAUNCHXL-CC2640R2 (<http://www.ti.com/tool/LAUNCHXL-CC2640R2>).

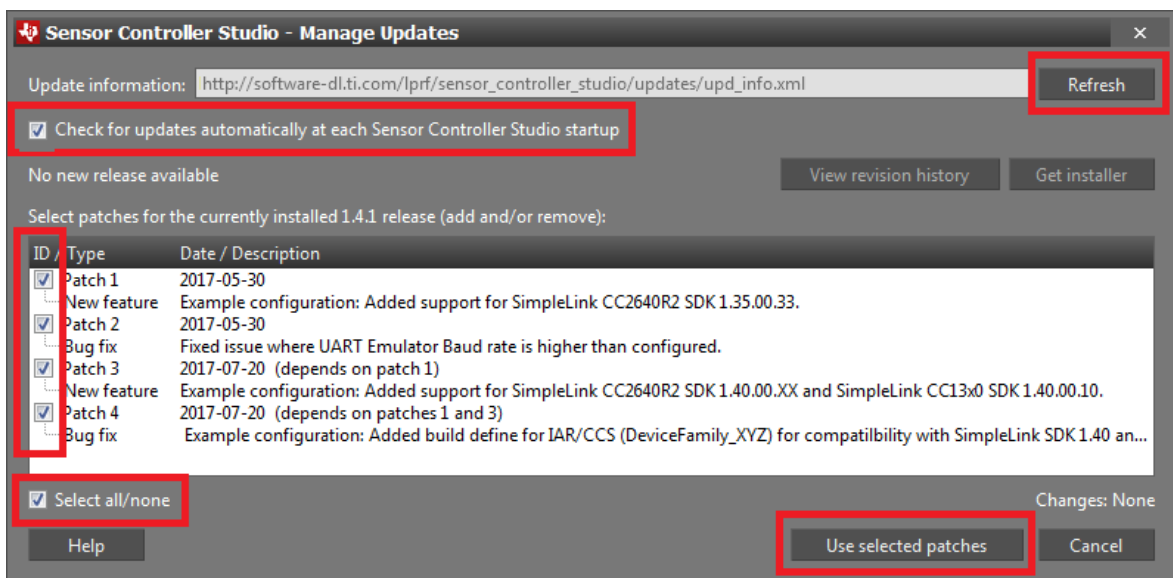
Getting Started

Set up Desktop Environment

1. Make sure CCS is installed. You can find version info in the menu: *Help → About Code Composer Studio*

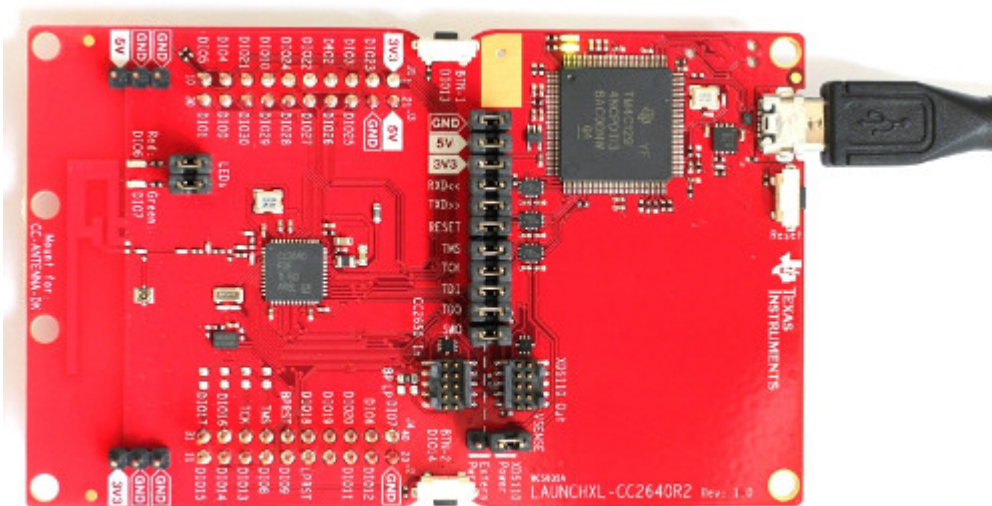


2. Make sure the correct SDK is installed. If it is installed to the default directory it should be located here: `C:\ti\simplelink_<device>_sdk_2_20_00_xx`
3. Run the `setup_sensor_controller_studio_xxx.exe` downloaded from the link above and install.
4. Open SCS and click `Updates → Check for Updates`. If any new patches are available, click `Updates → Manage Updates...` and apply all new patches. Note that the picture below is only used as an example. You may have a newer version, and there may be no patches available.

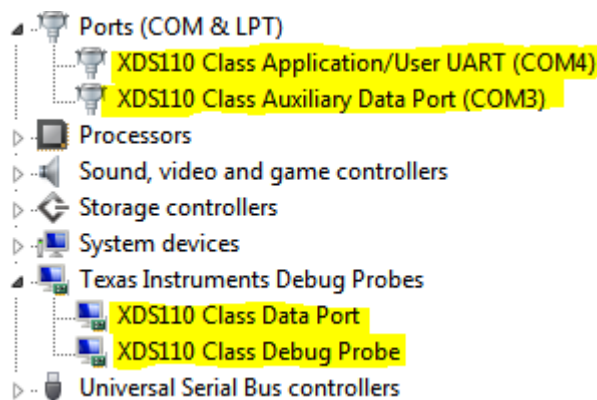


Connect LaunchPad

Simply connect the micro-USB cable from your computer to the LaunchPad as shown in the figure below:



When the LaunchPad is connected, the Windows Device Manager (*Start* → *Run* → *mmc devmgmt.msc* → *Ok*) should show you the following devices connected:



Task 1 – Set up Project in SCS

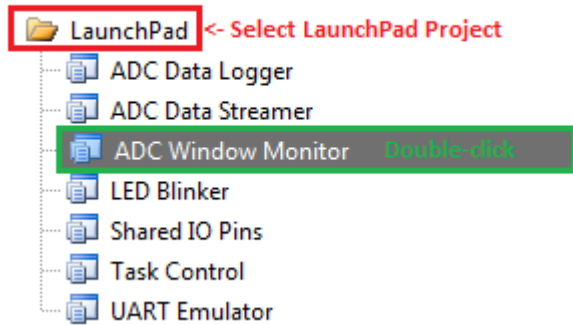
You will now configure the ADC Window Monitor example code in SCS and import the pre-made project into CCS. These are the steps that will be covered in more details in the sub-chapters below:

- Configure the SCS example project for your device.
- Generate the Sensor Controller driver from the SCS project.
- Import the premade CCS project specification which will import these files:
 - Main source example file that show how to integrate a Sensor Controller driver. This file is pre-made and is not part of the generated Sensor Controller driver.
 - Sensor Controller driver.
 - Sensor Controller driver framework.

Configure the Sensor Controller Example Project

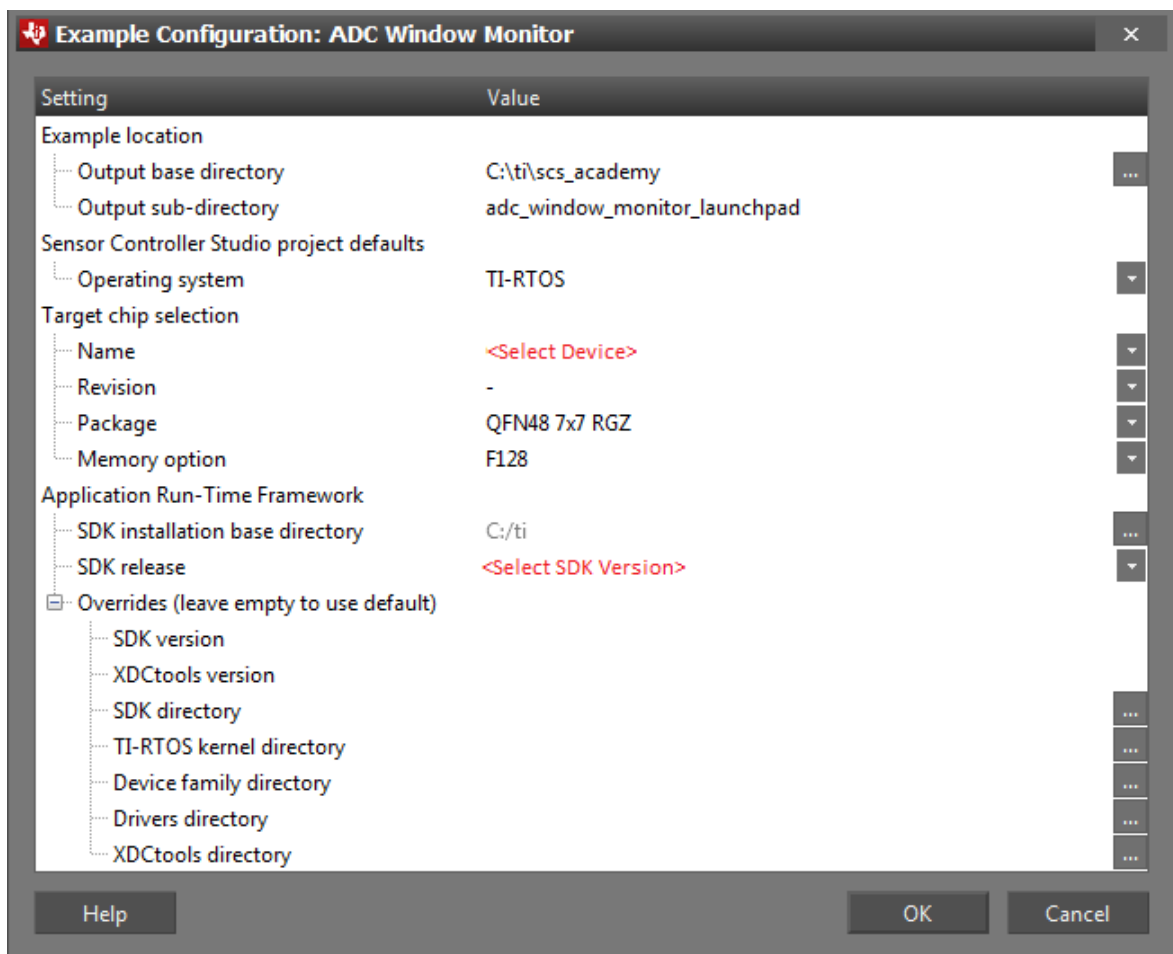
- Open SCS and double-click on the "ADC Window Monitor" LaunchPad example project as shown in the figure below.

Examples



Now an Example Configuration Window will appear (refer to figure below). Do as follows:

- Select Target chip device name.
- Select correct SDK release.
- Change Output base directory to C:\ti\scs_academy .
- Your settings should match the screen shot below.



When the project is opened in SCS, the main source project and IDE project files will automatically be populated into the output base directory. The default path here is your documents folder C:\Users\<your user>\Documents\Texas Instruments\Sensor Controller Studio\examples where <your user> is your windows user name (%USERPROFILE%).

Generate the Sensor Controller Driver

- Go to the Code Generator Pane in SCS (CTRL+G) and press **Generate driver source code**.
- Verify that the driver is generated successfully in the event log. Then press View output directory (button is located at the bottom right corner).

What does Sensor Controller Studio Generate?(multiple correct answers)

Example code to run on the main application processor

Example project for CCS/IAR

Driver Framework (scif_framework)

Sensor Controller Processor Assembler Code (sce.lst generated as an array in pAuxRamImage in scif.c)

Operating System Abstraction Layer (scif_osal_tirtos)

Sensor Controller driver (scif)

SCS generates the SCIF driver and framework. Example projects included with SCS are based on pre-made code which resides in its own `example` directory in the installation directory. Each example project includes hard-coded examples of simple main application source files (`main.c` / `main_tirtos.c`) along with project files for both CCS and IAR.

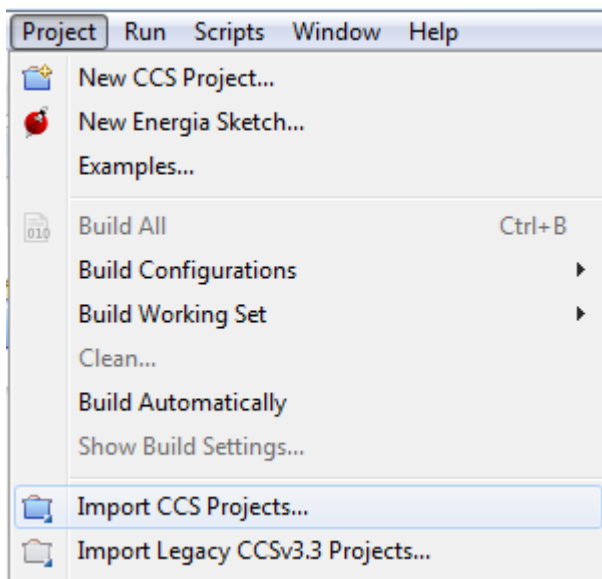
Never open Sensor Controller Studio example projects directly from the installation directory. You should instead always open example projects from within Sensor Controller Studio. Doing modifications to an example project which were opened from the installation directory would be permanent. Sensor Controller Studio uses these files to restore examples in your base output directory, allowing you to always restore the original example project.

To integrate a driver into your custom application you need to create your own main application source file (running on the main application processor). You can use the SCS examples as a reference and starting point. The Sensor Controller firmware image is found in the driver file `scif.c`. A more human- readable format of the Sensor Controller assembler source code is available in the assembler listing file `sce.lst`.

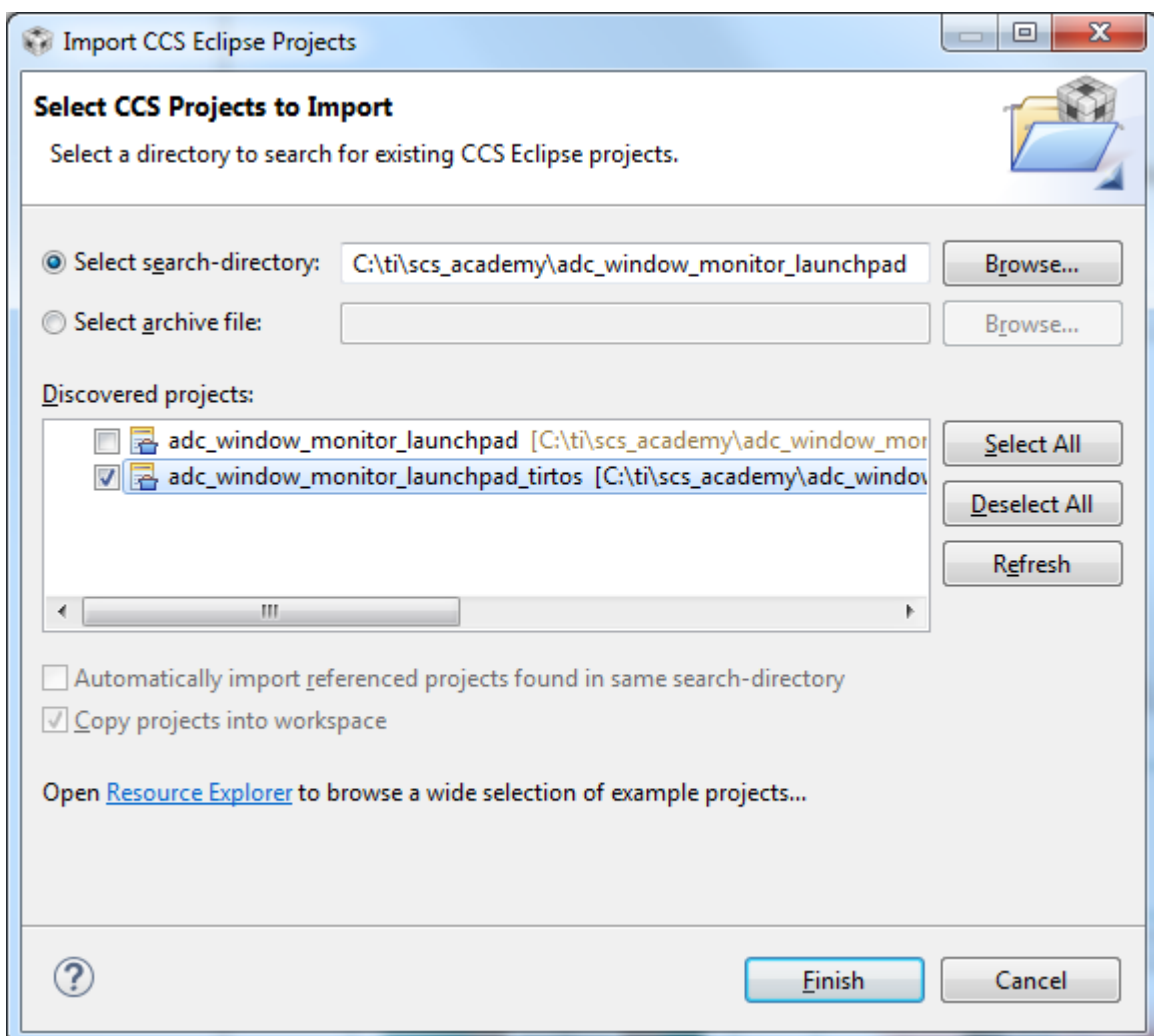
Import to Code Composer Studio

Do the following:

- Open CCS and press *Project* → *Import CCS Project*



- Select `C:\ti\scs_academy\adc_window_monitor_launchpad` as the search- directory. This is the SCS project output base directory, which was set in the example configuration.
- Select `adc_window_monitor_launchpad_tirtos` project and press **Finish**.

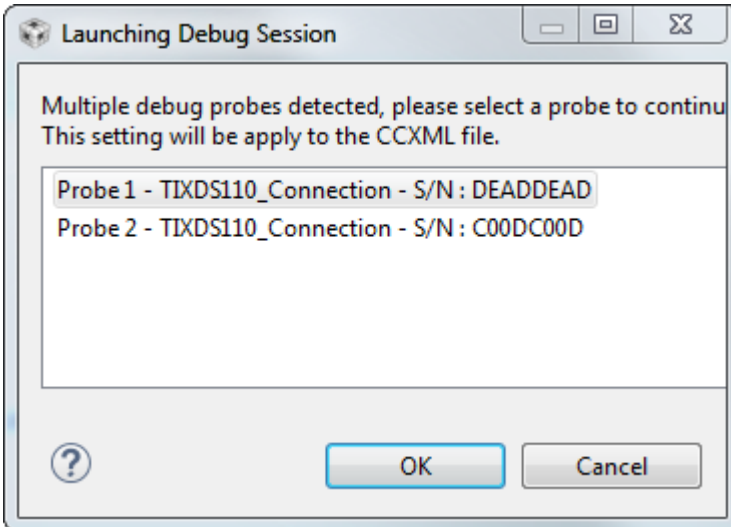


- Compile the project. Press the hammer icon, or right click your project in the Project Explorer window then click `Build Project` , to build in CCS.

- Verify that the build completes without errors.

Configure the debugger connection

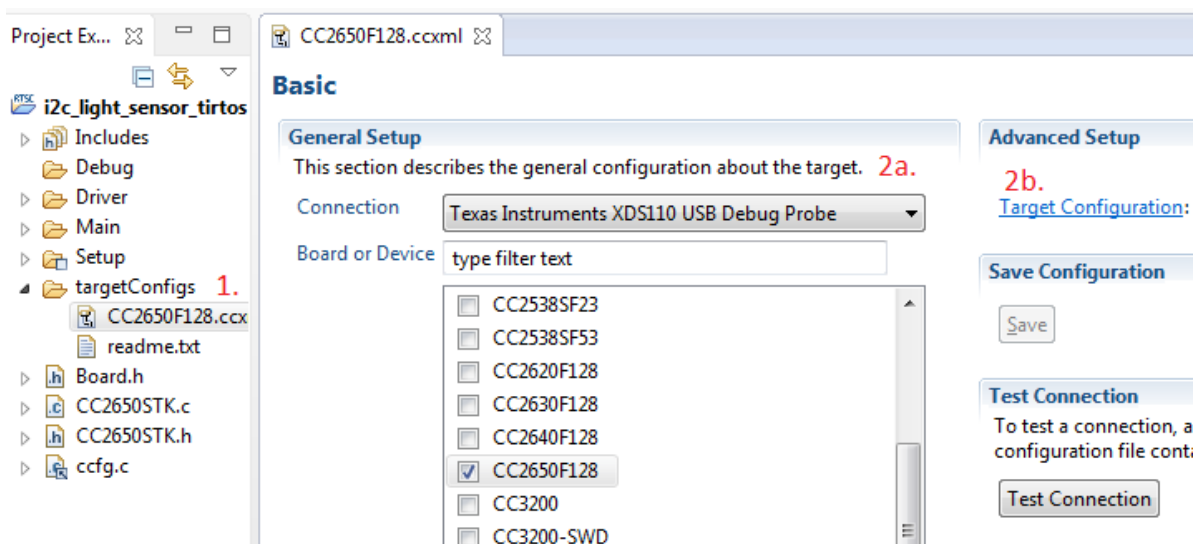
The correct debugger type (XDS110 for LaunchPad) is selected by default when you import the project. If more than one debugger of the same type connected, the GUI will query which debugger you want to use. Simply click on the debugger of choice and press OK.



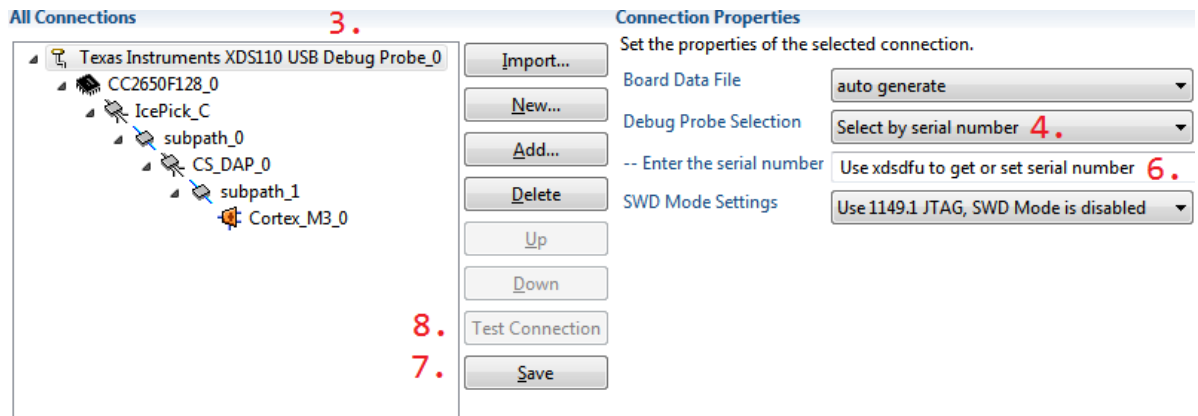
If the debugger connection has already been configured, but you want to change to a different debugger connection, see instructions below for how this can be changed manually.

Expand instructions for manual debugger selection in CCS Desktop

Make sure you only have one debugger of the same type connected when checking the serial number.



- Open the file `targetConfigs/CC2650F128.ccxml`.
- Change the connection to XDS110 USB Debug Probe if needed.
- Click on Target Configuration



- Click on the top-level node (XDS110 USB Debug Probe)
- Choose 'Select by serial number'
- Start command prompt and call
`c:\ti\ccs\ccs_base\common\uscif\xds110\xdsdfu.exe -e` to enumerate the connected debugger. This will give you the serial number of the device.

```
C:\ti>c:\ti\ccs\ccs_base\common\uscif\xds110\xdsdfu.exe -e

USB Device Firmware Upgrade Utility
Copyright (c) 2008-2014 Texas Instruments Incorporated. All rights reserved.

Scanning USB buses for supported XDS110 devices...

<<<< Device 0 >>>>

VID: 0x0451    PID: 0xbef3
Device Name:   XDS110 with CMSIS-DAP
Version:       2.2.4.2
Manufacturer: Texas Instruments
Serial Num:    0000FF01
Mode:          Runtime

Found 1 device.

C:\ti>
```

Finding XDS110 serial number

- Insert the serial number. Above it's 0000FF01 .
- Save the settings.
- Test the connection.
- Do this for both the App and the Stack projects.

You can also change the serial number to something easier to remember, or in case two debuggers have the same serial number.

```
C:\ti>c:\ti\ccs\ccs_base\common\uscif\xds110\xdsdfu.exe -m

USB Device Firmware Upgrade Utility
Copyright (c) 2008-2015 Texas Instruments Incorporated. All rights reserved.

Scanning USB buses for supported XDS110 devices...

<<<< Device 0 >>>>

VID: 0x0451      PID: 0xbef3
Device Name:     XDS110 with CMSIS-DAP
Version:         2.2.4.2
Manufacturer:    Texas Instruments
Serial Num:      BADEABBA
Mode:            Runtime

Switching device into DFU mode.

C:\ti>c:\ti\ccs\ccs_base\common\uscif\xds110\xds110\xdsdfu.exe -s ABCDEF01 -r

USB Device Firmware Upgrade Utility
Copyright (c) 2008-2015 Texas Instruments Incorporated. All rights reserved.

Scanning USB buses for supported XDS110 devices...

Setting serial number to "ABCDEF01"...

c:\ti>
```

Optionally change the serial number

Task 2 - Download and Debug with CCS

- Make sure that your HW is connected as instructed earlier in this guide.
- Press the Debug button (bug icon) or press **F11** to start the debug session.
- Then press Resume (**F8**) to allow the code to run. The ADC will sample the configured input pin (DIO23) at an interval and will notify the main application processor if the measured/converted ADC input value change to either below or above a set window threshold. The main application processor will read out the result and set:
 - Red LED (DIO6) if ADC input > high threshold.
 - Green LED (DIO7) if ADC input < low threshold.

Breakpoints and source browsing in CCS

- In `main_tirtos.c` set breakpoints inside the two if statements as shown below in CCS by double clicking the desired lines or pressing `CTRL+SHIFT+B`.

```
// Wait for an ALERT callback
Semaphore_pend(Semaphore_handle(&semScTaskAlert), BIOS_WAIT_FOREVER); // Waiting for alert from Sensor Controller

// Clear the ALERT interrupt source
scifClearAlertIntSource();

// Indicate on LEDs whether the current ADC value is high and/or low
if (scifTaskData.adcWindowMonitor.output.bvWindowState & SCIF_ADC_WINDOW_MONITOR_BV_ADC_WINDOW_LOW) {
    PIN_setOutputValue(hLedPins, Board_GLED, 1);    // Set breakpoint here (Ctrl+Shift+B)
} else {
    PIN_setOutputValue(hLedPins, Board_GLED, 0);
}
if (scifTaskData.adcWindowMonitor.output.bvWindowState & SCIF_ADC_WINDOW_MONITOR_BV_ADC_WINDOW_HIGH) {
    PIN_setOutputValue(hLedPins, Board_RLED, 1);    // Set breakpoint here (Ctrl+Shift+B)
} else {
    PIN_setOutputValue(hLedPins, Board_RLED, 0);
}
```

- Now run the application (`F8`) and alternate the ADC input (DIO23) between High (3V3) and low (GND) and observe which breakpoint is hit in the code. Press resume (`F8`) after the breakpoint is hit to allow the code to resume from halt. The picture below show you how to connect the ADC input to high or low with a jumper or wire.



- Find the address of the `scifTaskData` struct. Tips: press `CTRL+H` for advanced search options in CCS.

Quiz

Where is the variable `scifTaskData.adcWindowMonitor.output.bvWindowState` located?

Can the Sensor Controller access the Main application RAM?

- Find out in which context the `scTaskAlertCallback` function is called.

Quiz

The HWI function `hwiTaskAlert` will post the semaphore (`semScTaskAlert`), the application is pending on this semaphore. Which event (from the Sensor Controller) will trigger the Hardware Interrupt (HWI) `hwiTaskAlert` handler function?

- Terminate the `adc_window_monitor_launchpad` project (`Ctrl + F2` or red square icon).

Task 3 - Download and Debug with SCS

In this task you will debug the Sensor Controller task directly from Sensor Controller Studio. Sensor Controller Studio taking over the role of the System CPU application and interacting with the Sensor Controller task through the debugger. More detailed information about this can be read in the SCS Help viewer (press `F1` in SCS and go to `Task Testing Panel`). We will now debug the Sensor Controller task without any application running on the main application processor.

Quiz

Is it currently possible to debug both code on the main application processor and the Sensor Controller at the same time (concurrently)?

- In SCS, go to the Task Testing panel by clicking on the tab or pressing **Ctrl + T**.
- In the Task Testing window, if you have multiple projects open; select the ADC Window Monitor for LaunchPad project like shown in the screen shot below.
- Be sure to select **low-level workflow** before connecting to target.

When taking over the role of the System CPU application during task testing and debugging, SCS cannot interact with the Sensor Controller in real-time, and also has no knowledge of what a task iteration (during task testing) actually is. The sequence of events that will occur during a task iteration must therefore be specified, and this is done by listing a sequence of actions. In this example you can simply set up **Run Execution Code**.

Verify your Task Testing window matches the screenshot below before starting task testing.

1 - Select Project

2 - Select Task

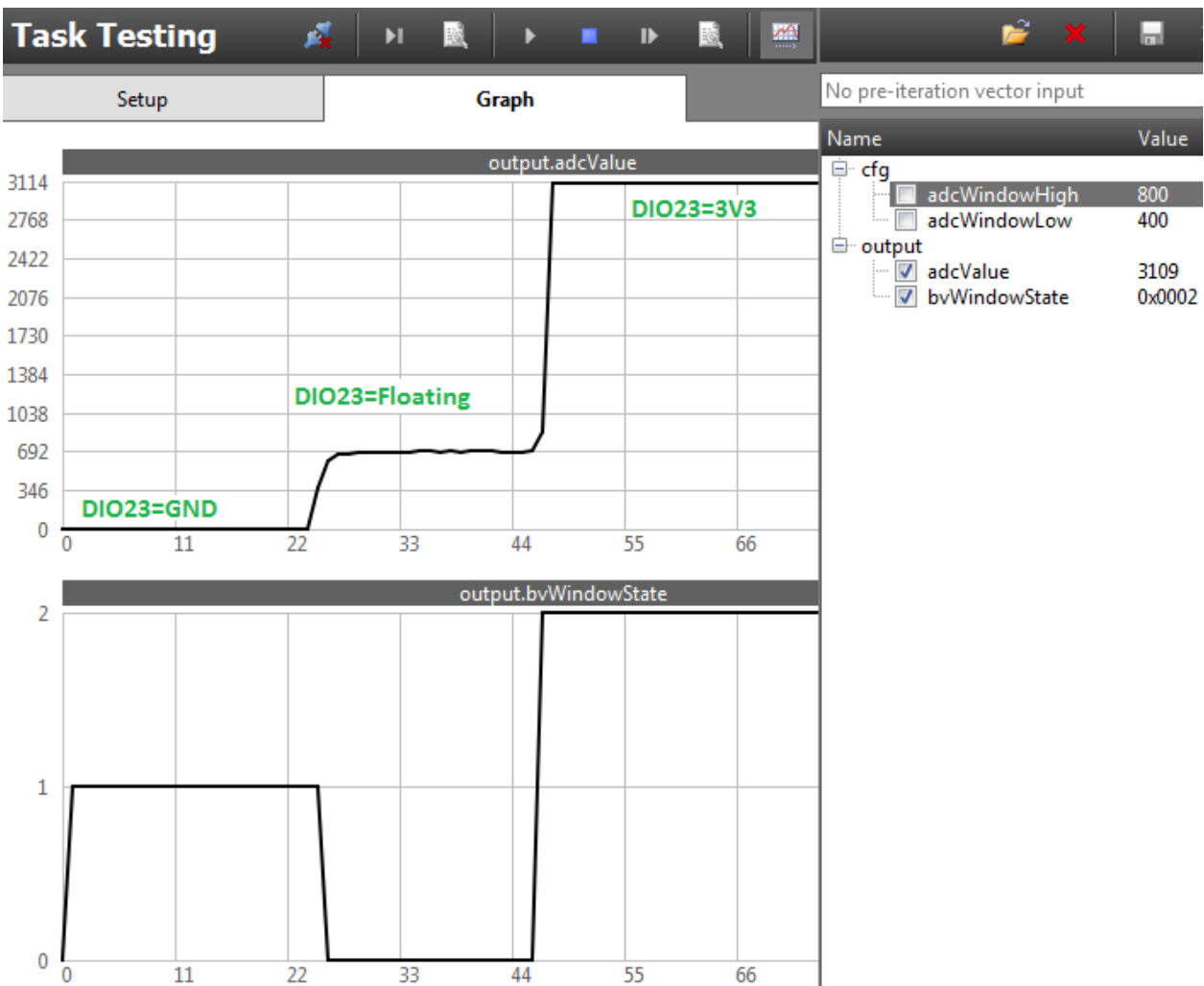
3 - Select workflow

4 - Set up task iteration sequence

- Start a debug session by connecting to target (**F12**).
- Set the threshold values in the **cfg RAM** variables as shown in the screen shot below. These values are set by the main application and therefore needs to be set manually in task testing.

Name	Value
cfg	
adcWindowHigh	800
adcWindowLow	400
output	
adcValue	0
bvWindowState	0x0000

- Run the initialization code once (F6).
- Run task iterations continuously (F5).
- Experiment and adjust the input level on the ADC input. You can use a jumper and connect the DIO23 pin to 3V3 and GND. Observe the adcValue and bvWindowState in the graph window as in the screen shot below.



The icons and hot keys for debugging alter meaning and form during each step which means the hot keys, for example F6 , have different actions during the different phases. This makes it easier to restart the execution code as you can press F6 twice or press the same icon twice to run termination code and initialization code again before starting the execution code again.

Now you will start a task testing (assembler code) debugging session:

- When you are satisfied press stop (F6).

- Run the termination code (F6).
- Run the initialization code once (F6).
- Start a debug session of one task iteration (CTRL + F11).
- Place a breakpoint as shown in the image below where the converted ADC input value is stored to the RAM variable output.value (F9).
- Press run continuously (F5).
- Now you should hit the breakpoint. Single step in the code by pressing F11 and observe the new value being loaded into output.adcValue (seen on right pane). Change the input ADC value, repeat the process and verify the new measured input voltage.

The screenshot displays the 'Task Debugging - Ex' window with three main panes:

- Assembly Code Pane:** Shows assembly instructions with addresses and comments. Key instructions include:
 - 008a ---- fb4c out R7, [#IOP_ADISE]
 - 008b ---- 7003 ld R7, # (ADI16_ADI)
 - 008c ---- fb4c out R7, [#IOP_ADISE]
 - 008d ---- fd47 nop
 - 008e ---- fb4c out R7, [#IOP_ADISE]
 - 008f ---- 1465 jsr AdiDdiRelease
 - 0090 ---- 6403 iobset #0, [#IOP_ANAI]
 - 0091 ---- 001f ld R0, #EVCTL_SCEI
 - 0092 ---- 8b2c out R0, [#IOP_EVCTI]
 - 0093 ---- fdb1 wevl #WEVSEL_PROG
 - 0094 ---- 8902 in R0, [#IOP_ANAI]
 - 0095 ---- 0c76 st R0, [#adcWindow]
 - 0096 ---- 14be jsr AdcDisable
 - 0097 ---- 7000 ld R7, #0
 - 0098 ---- 1875 ld R1, [#adcWindow]
- RAM Variables Pane:** Shows a tree view of variables:
 - cfg: adcWindowHigh (0), adcWindowLow (0)
 - output: adcValue (0), bvWindowState (0x000)
- CPU Registers Pane:** A table showing the state of CPU registers and flags.

CPU	Value	Flag	Value	Control	Value
R0	0x00aa	Z FLAG	1	EXC VECTOR	0x00
R1	0x0000	V FLAG	0	WAKEUP	1
R2	0x0000	C FLAG	0	EVENTS	0x00
R3	0	N FLAG	0	PC	0x0008
R4	0	SLEEP	0	OPCODE	0x4436
R5	0x0000	WEV	0	LOOP START	0x0000
R6	0x0000	SELF STOP	1	LOOP END	0x0000
R7	0x0000	BUS ERROR	0	LOOP COUNT	0x00

Task 4 - Understand

In this task you will try to grasp the basic flow of the ADC Window Monitor example program and solve a quiz to make sure you understand the concepts. Code from the Sensor Controller or the main application is presented in the format shown in the table below.

Code Context	Core	Formatted Example	Format
Application Processor (AP)	32-bit Cortex-M	<i>scifStartRtcTicksNow()</i>	Code (Italic)
Sensor Controller Processor (SCP)	16-bit RISC	fwScheduleTask(1);	Code

Program Flow (ADC Window Monitor for LaunchPad)

Here is a brief step-by-step list on the program flow for the ADC Window Monitor project:

Boot and Initialization

- 1) The complete application including the Sensor Controller driver will be loaded as a **flash image** on to the device.
- 2) On boot, the *main()* function will configure a task, initialize a semaphore and boot the TI-RTOS kernel (*BIOS_start()*).
- 3) In the task function (*taskFxn*) everything needed for the Sensor Controller driver will be configured:
 - (1) Construct Sensor Controller framework semaphore.
 - (2) Interrupt handlers are registered.
 - (3) Load Sensor Controller program image to the Sensor Controller RAM and configure the complete driver setup which includes IO Mapping, Domain Clocks, control and data variables, HWI's and more. For more details refer to the source code.
 - (4) Schedule periodic wakeup interval for the Sensor Controller.
 - (5) Write upper and lower window limits directly to Sensor Controller RAM.
 - (6) The Sensor Controller task is started and the initialization code will run once.

```
// Initialize the Sensor Controller
scifOsalInit(); // (1)
scifOsalRegisterCtrlReadyCallback(scCtrlReadyCallback); // (2)
scifOsalRegisterTaskAlertCallback(scTaskAlertCallback); // (2)
scifInit(&scifDriverSetup); // (3)
scifStartRtcTicksNow(0x00010000 / 8); // (4)

/* Configure and start the Sensor Controller's
 * ADC window monitor task (not to be confused with RTOS tasks)
 */
scifTaskData.adcWindowMonitor.cfg.adcWindowHigh = 800; // (5)
scifTaskData.adcWindowMonitor.cfg.adcWindowLow = 400; // (5)
scifStartTasksNbl(BV(SCIF_ADC_WINDOW_MONITOR_TASK_ID)); // (6)
```













Program Loop Running

- 4) TI-RTOS task will then pend on the semaphore `semScTaskAlert` (device enter sleep/standby).
- 5) The Sensor Controller wakes up every 125 ms (interrupt from RTC channel 2 compare event) and runs the execution code as scheduled (`fwScheduleTask(1);`).
- 6) The execution code will trigger an ADC conversion. If the ADC value is below or above the set window thresholds it wakes up the TI-RTOS task (`fwGenAlertInterrupt();`).
- 7) The triggered HWI calls `scTaskAlertCallback` which posts the `semScTaskAlert` semaphore.
- 8) The TI-RTOS task `taskFxn` iterate once through the while-loop.
- 9) LED's are set/cleared and the interrupt source cleared.
- 10) Repeat steps 5) to 9) forever (while-loop).

Review SCS Project Panes

Review the open project (ADC Window Monitor for LaunchPad) in SCS and review the project panels. Press `F1` and navigate to the Project Panel section where you find all the details for every panel in the project. The table below lists all the project panels with a very short description.

Termination Code will only run once when task is stopped by the application processor (`scifStopTasksNbl()`) or at the end of a single iteration for all task code blocks (`scifExecuteTasksOnceNbl()`). In the "ADC Window Monitor" example the termination code never runs and is therefore left blank.

Panel (Tab) 	Description
 ADC Window Monitor for LaunchPad	Define project name, location, chip/OS configuration, description and add tasks.
 Power and Clock Settings	CC13x2/CC26x2 devices only: Configure power and clock modes for various parts of the project.
 ADC Window Monitor	Configure task name, description and resources for every Task in the project.
 Initialization Code	Code that will run once when task is initialized.
 Execution Code	The Execution Code will run when scheduled by <code>fwScheduleTask()</code> . Is only available if the RTC-Based Execution Scheduling resource is enabled.
 Even Code	The Event Handler Code will run when its corresponding event is triggered. Is only available if any Event Trigger resources are enabled, e.g. GPIO Event Trigger .
 Termination Code	Termination code run once on task exit or never if the task runs forever.
 I/O Mapping	Configure I/O functions to I/O pins.
 Code Generator	Compile task code and generate application framework.
 Task Testing	Test and debug a single Sensor Controller task.
 RTL	Run-Time Logging panel for evaluating and optimizing performance.

Review Sensor Controller Task Code

Browse the different task codes (initialization, execution, event handler, termination) and answer the following quiz.

Quiz

How many times will Initialization Code run in the *adc_window_monitor_launchpad* example?

(Hint: `scifCtrlTasksNb1()` in `scif_framework.c`)

☐ Once☐ Every time the device comes out of STANDBY

How often will Execution Code run in the *adc_window_monitor_launchpad* example?

(Hint: `fwScheduleTask()` procedure in SCS)

☐ every 10 seconds☐ every millisecond☐ every 100 milliseconds☐ every 125 milliseconds

How often will Termination Code run in the *adc_window_monitor_launchpad* example?

(Hint: `fwScheduleTask()`)

☐ Never☐ Before the Device goes in to standby

Great Work!

You completed first module and demonstrated the Sensor Controller!

Below is two optional bonus tasks that will let you get more hands on and learn important key concepts about the Sensor Controller and ADC.

Task 5 - Bonus Tasks (Low to Intermediate Level)

These tasks will let you get hands on and modify Sensor Controller task code, modify application code (TI-RTOS) as well as discuss ADC accuracy.

Bonus Task 1 - Read and Correct ADC Value

In this task you will read the raw ADC value directly from the Sensor Controller RAM in the TI-RTOS main application and correct for ADC gain error and ADC offset error. The ADC offset and gain error are calculated in production test and stored in FCFG1. The driverlib file `aux_adc.h` contains functions that can retrieve and apply these corrections to a raw ADC conversion result as well as reverse a corrected value back to the equivalent raw ADC result

value. These functions must be run in the main application as the computations are too demanding for the Sensor Controller processor. For example, if the driver needs to trigger on a precise ADC value threshold you can calculate the corresponding raw value in the main application and write the value directly to the Sensor Controller RAM. You will do this in the exercise Configure Microvolt Threshold below.

Read ADC Value Directly from Sensor Controller RAM

The framework (`scif.h`) defines structs for all the Sensor Controller RAM variables. Try to read the raw adc result after an ADC conversion takes place. Simply access the correct sub-members of the `scifTaskData` struct. The figure below will guide you to the correct struct member variable.

```

/// ADC Window Monitor: Task output data structure
typedef struct {
    uint16_t adcValue;
} SCIF_ADC_WINDOW_MONITOR_OUTPUT_T;

typedef struct {
    struct {
        SCIF_ADC_WINDOW_MONITOR_OUTPUT_T output;
    } adcWindowMonitor;
} SCIF_TASK_DATA_T;

#define scifTaskData (*((volatile SCIF_TASK_DATA_T*) 0x400E00E8))
  
```

The diagram illustrates the nested structure of the `scifTaskData` variable. It shows that `scifTaskData` is a pointer to a `SCIF_TASK_DATA_T` struct. This struct contains an `adcWindowMonitor` member, which is itself a struct containing an `output` member. The `output` member is of type `SCIF_ADC_WINDOW_MONITOR_OUTPUT_T`, which contains the `adcValue` member. Red circles and arrows highlight the path from `scifTaskData` to `adcValue`.

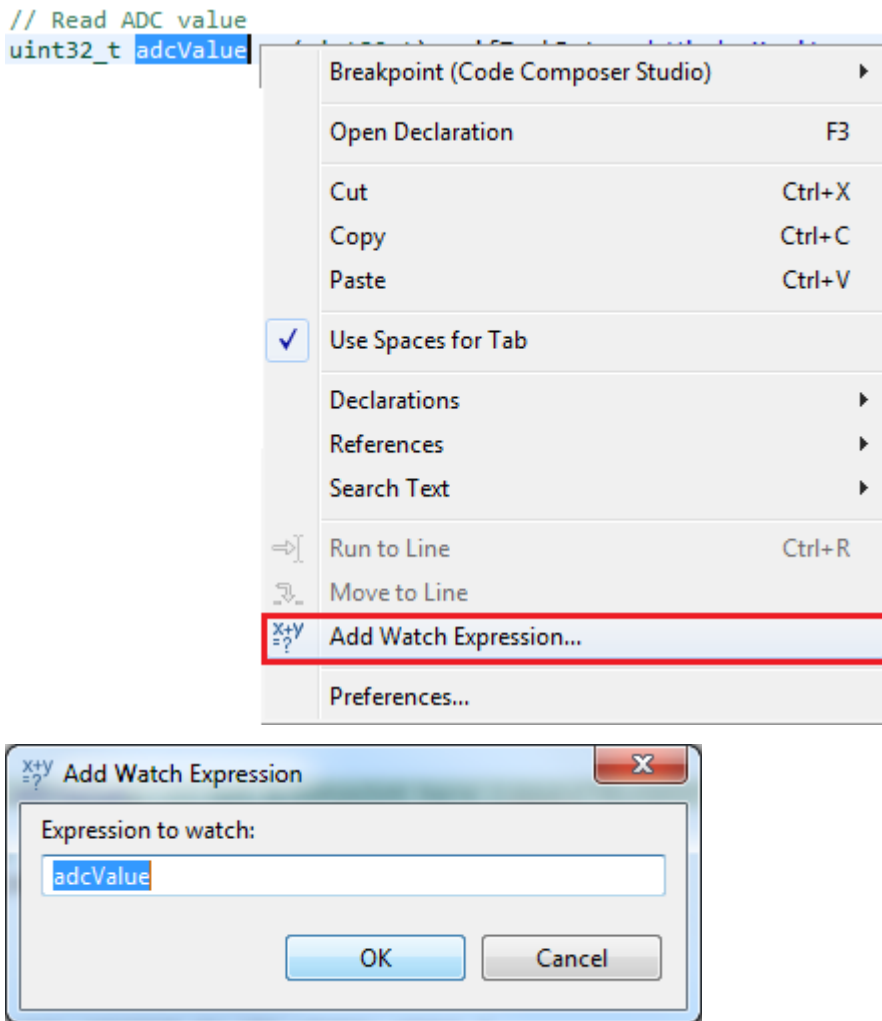
Expand Code Solution (snippet from `main_tirtos.c`):

```

// Wait for an ALERT callback
Semaphore_pend(Semaphore_handle(&semScTaskAlert), BIOS_WAIT_FOREVER);

// Clear the ALERT interrupt source
scifClearAlertIntSource();

// Read ADC value
int32_t adcValue = (int32_t) scifTaskData.adcWindowMonitor.output.adcValue; // Set breakpoint here (CTRL+SHIFT+B.)
  
```



- After you have written code to read the raw ADC value, set breakpoint at the line where the value is read (CTRL+SHIFT+B .)
- Compile, download and debug in CCS by pressing F11 .
- When ready to run, press F8 to run to breakpoint.
- After the the breakpoint is hit, right-click on the read variable and press Add Watch Expression...
- Press ok on the dialog box.
- Single step (over) F6 and observe the value.
- Change the ADC input voltage and repeat the steps above to verify correct behavior.

Correct Raw ADC Value and Convert to Microvolts

In this sub-task you will correct the raw ADC value and modify the main application code to set/clear LED's depending on threshold values given in microvolt. Direct links for specific devices are given below:

- Review the driverlib API documentation for `aux_adc` which contains all the functions you need to solve this task.

- The driverlib API can be found under the docs folder in your SimpleLink SDK installation, i.e.
`<SDK>/docs/driverlib_cc13xx_cc26xx/docs_overview_driverlib_cc13xx_cc26xx.html` .
 - Or, you can find the driverlib API in the TI Resource Explorer. For example, the SimpleLink CC13x0 SDK would have the following path:
 Software → SimpleLink CC13x0 SDK - v:*version* → Documents → DriverLib
 → Driverlib Documentation
- Correct the Raw ADC value (offset error and gain error).
 - Convert the corrected ADC value to microvolts.
 - Compile, download and debug (F11).
 - Add watch expression for both the raw adc value and the corrected version. Single step in debug mode and observe the delta.

Expand Task Code Solution (main_tirtos.c):



```
#include <ti/devices/DeviceFamily.h>
#include DeviceFamily_constructPath(driverlib/aux_adc.h)

void taskFxn(UArg a0, UArg a1) {

    int32_t adcOffset = AUXADCGetAdjustmentOffset(AUXADC_REF_FIXED);
    int32_t adcGainError = AUXADCGetAdjustmentGain(AUXADC_REF_FIXED);
    int32_t adcValue, adcCorrectedValue, adcValueMicroVolt;

    PIN_Handle hLedPins;

    // Enable LED pins
    hLedPins = PIN_open(&ledPinState, pLedPinTable);

    // Initialize the Sensor Controller
    scifOsalInit();
    scifOsalRegisterCtrlReadyCallback(scCtrlReadyCallback);
    scifOsalRegisterTaskAlertCallback(scTaskAlertCallback);
    scifInit(&scifDriverSetup);
    scifStartRtcTicksNow(0x00010000 / 8);

    // Configure and start the Sensor Controller's ADC window monitor task (not to
    be confused with OS tasks)
    scifTaskData.adcWindowMonitor.cfg.adcWindowHigh = 800;
    scifTaskData.adcWindowMonitor.cfg.adcWindowLow = 400;
    scifStartTasksNbl(BV(SCIF_ADC_WINDOW_MONITOR_TASK_ID));

    // Main loop
    while (1) {

        // Wait for an ALERT callback
        Semaphore_pend(Semaphore_handle(&semScTaskAlert), BIOS_WAIT_FOREVER);

        // Clear the ALERT interrupt source
        scifClearAlertIntSource();
```

```

// Read ADC value
adcValue = scifTaskData.adcWindowMonitor.output.adcValue;

// Correct ADC raw value
adcCorrectedValue = AUXADCAdjustValueForGainAndOffset((int32_t) adcValue,
adcGainError, adcOffset);

// Convert ADC value to Microvolts.
adcValueMicroVolt = AUXADCValueToMicrovolts(AUXADC_FIXED_REF_VOLTAGE_NORMA
L,adcCorrectedValue);

// Indicate on LEDs whether the current ADC value is high and/or low
if (scifTaskData.adcWindowMonitor.output.bvWindowState & SCIF_ADC_WINDOW_M
ONITOR_BV_ADC_WINDOW_LOW) {
    PIN_setOutputValue(hLedPins, Board_GLED, 1);
} else {
    PIN_setOutputValue(hLedPins, Board_GLED, 0);
}
if (scifTaskData.adcWindowMonitor.output.bvWindowState & SCIF_ADC_WINDOW_M
ONITOR_BV_ADC_WINDOW_HIGH) {
    PIN_setOutputValue(hLedPins, Board_RLED, 1);
} else {
    PIN_setOutputValue(hLedPins, Board_RLED, 0);
}

// Acknowledge the alert event
scifAckAlertEvents();
}

} // taskFxn

```

The screenshot displays the TI Studio IDE interface during a debug session. The main window is divided into several panes:

- Left Pane (Project Explorer):** Shows the project structure for 'adc_window_monitor_launchpad_tirtos'. The file 'main_tirtos.c' is selected.
- Top Pane (Registers):** Displays the state of registers. The 'Variables' tab is active, showing:

Name	Type	Value	Location
(x)= a0	unsigned int	0	0x20000DC8
(x)= a1	unsigned int	0	0x20000DCC
(x)= adcCorrectedValue	int	3162	0x20000DDC
- Bottom Left Pane (Code Editor):** Shows the source code for 'main_tirtos.c'. The current line of execution is highlighted in green:


```

144 if (scifTaskData.adcWindowMonitor.output.bvWindowSt
145     PIN_setOutputValue(hLedPins, Board_GLED, 1);

```
- Bottom Right Pane (Expressions):** Displays the values of expressions being evaluated:

Expression	Type	Value
(x)= adcValue	int	3129
(x)= adcCorrectedValue	int	3162
(x)= adcValueMicroVolt	int	3320288

Configure Microvolt Threshold

Now you will configure new threshold values based on microvolt. Convert from microvolt to raw value (*AUXADCMicrovoltsToValue*) and calculate the raw threshold values with corrections for gain error and offset error (*AUXADCUnadjustValueForGainAndOffset*).

- Define new thresholds for the window monitor:
 - Window High = 0.8 Volts (800000 microvolts)
 - Window Low = 0.4 Volts (400000 microvolts)
- Calculate the corresponding raw values and write these to the appropriate Sensor Controller configuration variables.
- Compile, download and debug (F11).
- If you have an external power supply you can apply voltages to the ADC input (DIO23) to verify if the Sensor Controller window monitor triggers on the expected voltage values.

Expand Task Code Snippet Solution (main_tirtos.c):



```
/* Set threshold based on microVolt */
int32_t adcThresholdLowMicro = 400000;      // 0.4 Volt
int32_t adcThresholdHighMicro = 800000;     // 0.8 Volt

int32_t adcThresholdLow = AUXADCMicrovoltsToValue(AUXADC_FIXED_REF_VOLTAGE_NOR
MAL, adcThresholdLowMicro);
int32_t adcThresholdHigh = AUXADCMicrovoltsToValue(AUXADC_FIXED_REF_VOLTAGE_NO
RMAL, adcThresholdHighMicro);

int32_t adcThresholdLowRaw = AUXADCUnadjustValueForGainAndOffset(adcThresholdL
ow, adcGainError, adcOffset);
int32_t adcThresholdHighRaw = AUXADCUnadjustValueForGainAndOffset(adcThreshold
High, adcGainError, adcOffset);

// Configure and start the Sensor Controller's ADC window monitor task (not to
be confused with OS tasks)
scifTaskData.adcWindowMonitor.cfg.adcWindowHigh = adcThresholdHighRaw;
scifTaskData.adcWindowMonitor.cfg.adcWindowLow  = adcThresholdLowRaw;
scifStartTasksNbl(BV(SCIF_ADC_WINDOW_MONITOR_TASK_ID));
```

Bonus Task 2: GPIO Event Handler

Now you will change the Sensor Controller driver from being a window monitor to trigger on either low or high input value on DIO23 (ADC input). For this you will use the GPIO Event Trigger resource in Sensor Controller. This will cause the Sensor Controller driver to trigger on interrupt instead of scheduling execution code and therefore all the content in the execution code can be removed. In the default ADC Window Monitor example project (*main_tirtos.c*), periodic wakeups based on RTC compare events is configured:

```
/* Wake up Sensor Controller every 125 ms
 * 1 second (0x00010000) / 8 = 125 ms. */
scifStartRtcTicksNow(0x00010000 / 8);
```

RTC configuration in application code

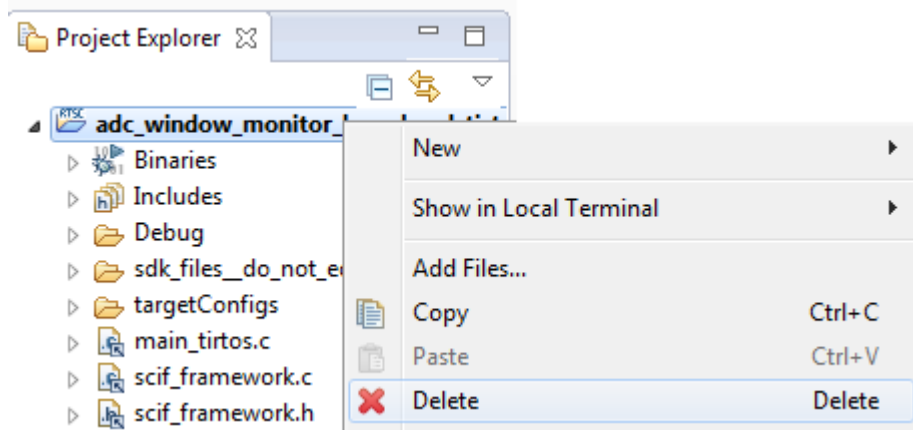
Every time the Sensor Controller wakes up it will check if a task is scheduled to run its execution code. If the previously run Sensor Controller code scheduled an execution with argument 1:

```
/* Schedule execution at the next Sensor Controller
 * RTC scheduled wakeup. */
fwScheduleTask(1);
```

Scheduling of RTC wakeup in Sensor Controller Task code

If there is only execution code in the Sensor Controller driver and every execution task is scheduled to run at every 10 wakeups (`fwScheduleTask(10);`) the Sensor Controller would wakeup 10 times and do nothing else except waste energy. In this case you should reconfigure the wakeups (`scifStartRtcTicksNow(0x00010000);`) and schedule the execution code to run at every wakeup (`fwScheduleTask(1);`). But in this task you do not need scheduling or the execution code as the event handler code is triggered to run directly after every interrupt. Let's start the task with a fresh copy of the ADC Window Monitor example project.

- Delete the modified ADC Window Monitor project in CCS and SCS:
 - To delete a project in CCS, right-click on the project, then press Delete and OK.



- Delete the physical project stored in `C:\ti\scs_academy\adc_window_monitor_launchpad`. If you want to save the work for future references simply rename the folder instead.
 - Close the project in SCS.
- Open clean ADC Window Monitor for LaunchPad project
 - Open the ADC Window Monitor project in SCS as you did in the start of this training. Now a fresh copy of the project will be spawned at the set output base directory.
 - Import the project in CCS.

- In `main_tirtos.c`: disable the Sensor Controller periodic wakeups:

```
// scifStartRtcTicksNow(0x00010000 / 8);
```

- Modify the Sensor Controller driver:

- Open task resources by navigating to the Task Panel for the ADC Window Monitor.
- Press the help viewer button on the GPIO Event trigger. Review the help text.

☒ GPIO Event Trigger

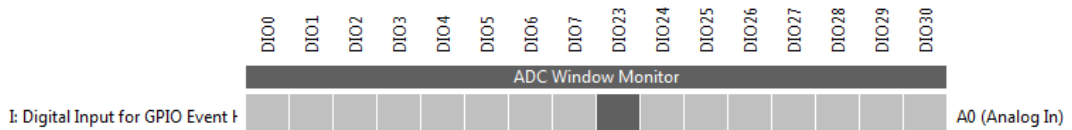


? Use a single GPIO pin to trigger the Event Handler Code

- In task panel under the Task resources, enable "Digital Input Pins", "System CPU Alert" and "GPIO Event Trigger":

Algorithms	
UART Emulator	? Full-duplex UART interface emulator
General-Purpose I/O	
Analog Open-Drain Pins	? General-purpose analog + open-drain pins (can drive to GND, for cap. touch)
Analog Open-Source Pins	? General-purpose analog + open-source pins (can drive to VDDS)
Analog Pins	? General-purpose analog pins, for use with analog peripherals
Differential Output Pins	? General-purpose digital differential output pin pairs
Digital Input Pins	? General-purpose digital input pins, for manual polling
GPIO_IN	Digital Input for GPIO Event Handler
Configuration on initialization	No pull
Configuration on uninitialization	No pull
Pin count	1 (accessed by constant)
Digital Open-Drain Pins	? General-purpose digital open-drain pins (drive low)
Digital Open-Source Pins	? General-purpose digital open-source pins (drive high)
Digital Output Pins	? General-purpose digital output pins
Peripherals	
ADC	? Analog to digital converter
COMP_A	? Continuous time comparator
COMP_B	? Low-power clocked comparator
ISRC	? Current source (for cap. touch and temperature measurement)
Pulse Counter	? Counts rising edges on a digital input pin or the COMP_A output
TDC	? Time to digital converter
Timer 0	? Execution timing for use within task iterations
Serial Interfaces	
I2C Master	? I2C master interface
SPI Chip Select	? SPI master chip select (CSN)
SPI Data Transfer	? SPI master clock and data (SCLK, MOSI and MISO)
System CPU Communication	
Multi-Buffered Output Data Exchange	? Seamless output buffer exchange with ALERT interrupt generation
Peripheral Sharing	? Sharing of peripheral hardware modules with the System CPU
System CPU Alert	? Basic ALERT interrupt generation from task code
Task Event Handling	
COMP_B Event Trigger	? Use COMP_B to trigger the Event Handler Code
GPIO Event Trigger	? Use a single GPIO pin to trigger the Event Handler Code
Timer Event Trigger	? Use timer to trigger the Event Handler Code
Task Execution	
RTC-Based Execution Scheduling	? Use the Real-Time Clock to trigger the Execution Code
Utilities	
Delay Insertion	? Delay insertion in task code
Math and Logic	? Basic mathematical operations
RTC Multi-Event Capture	? RTC capture of every N'th rising edge on a GPIO pin

- Go to the I/O Mapping tab and select DIO23 as input.



- In initialization code, set up gpio trigger to trigger on high match (evhSetupGpioTrigger).
- Copy the behavior from the execution code into the event handler code so that the application defined in main_tirtos.c will behave as before. Also set up the next GPIO trigger.
- Remove all contents in execution code.
- Generate the Sensor Controller driver (CTRL+G).
- Compile run and test project in CCS (F11).

Expand Solution to Sensor Controller Driver



Initialization Code:

```
output.bvWindowState = 0;  
evhSetupGpioTrigger(0, AUXIO_I_GPIO_IN, 1, EVH_GPIO_TRIG_ON_MATCH);
```

Event Handler Code:

```
if(output.bvWindowState == BV_ADC_WINDOW_LOW) {  
    output.bvWindowState = BV_ADC_WINDOW_HIGH;  
    // Set up the next interrupt trigger  
    evhSetupGpioTrigger(0, AUXIO_I_GPIO_IN, 0, EVH_GPIO_TRIG_ON_MATCH);  
} else {  
    output.bvWindowState = BV_ADC_WINDOW_LOW;  
    // Set up the next interrupt trigger  
    evhSetupGpioTrigger(0, AUXIO_I_GPIO_IN, 1, EVH_GPIO_TRIG_ON_MATCH);  
}  
fwGenAlertInterrupt();
```



(<http://creativecommons.org/licenses/by-nc-nd/4.0/>)

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

Introduction

This workshop will show you how to create and integrate a basic Sensor Controller ADC driver with a blank TI-RTOS project. The training is expected to take about 2h to complete. There should be at least an beginner level of knowledge of the C programming language as well experience with embedded software development to be able to complete the tasks.

The Sensor Controller ADC driver will measure an analog input voltage on one pin (DIO29) and set the green LED if the input ADC value is below a set threshold to indicate LOW input. If the ADC input value is above the set threshold it will notify the main application processor which then set the Red LED to indicate HIGH input. To vary the input voltage applied to the pin, an external voltage source can be connected to the analog input pin. In this workshop a jumper from the LaunchPad is used to short the analog input pin with adjacent pins (DIO28 and DIO30). There are two additional bonus tasks, one for *Bluetooth®* low energy and one for proprietary operation. Refer to the table below to find the required HW for each bonus task.



Compatible Connected MCU LaunchPad kits

This workshop can be completed with any one of the SimpleLink™ Wireless MCU with Sensor Controller devices described in the table below. Install the required Associated SimpleLink™ Software Development Kit matching your device. For more details on LaunchPads please visit the LaunchPad overview page (<http://www.ti.com/lstds/ti/tools-software/launchpads/launchpads.page>).

S-W-MCU	DK	SDK	Bonus
CC2640R2F (http://www.ti.com/product/CC2640R2F)	LAUNCHXL-CC2640R2 (http://www.ti.com/tool/launchxl-cc2640r2)	SimpleLink™ <i>Bluetooth®</i> low energy CC2640R2 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC2640R2-SDK)	Bonus Task 1
CC1350 (http://www.ti.com/product/CC1350)	LAUNCHXL-CC1350 (http://www.ti.com/tool/launchxl-cc1350)	SimpleLink™ CC13x0 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK)	Bonus Task 2
CC1310 (http://www.ti.com/product/CC1310)	LAUNCHXL-CC1310 (http://www.ti.com/tool/launchxl-cc1350)	SimpleLink™ CC13x0 Software Development Kit (http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK)	Bonus Task 2

Abbreviations / terminology

Abbreviation / terminology	Definition
CCS	Code Composer Studio
SC	Sensor Controller
SCS	Sensor Controller Studio
LP	LaunchPad
AUX RAM	Sensor Controller Memory
RTC	Real-Time Clock
RTOS	Real-Time Operating System
TI-RTOS	RTOS for TI microcontrollers
DK	Hardware Development Kit
SDK	Software Development Kit
S-W-MCU	SimpleLink™ Wireless Micro Controller Unit

Prerequisites

Completed material

- Sensor Controller Basics - Getting Started (../sc_01_basic/sc_01_basic.html)
- TI-RTOS Basics Lab 1 (../tirtos_01_basic/tirtos_01_basic.html)
- For Bonus Task 1: Bluetooth Low Energy Fundamentals workshop (../ble_01_basic/ble_01_basic.html)
- For Bonus Task 2: Proprietary RF – Basic RX and TX (../prop_01_basic/prop_01_basic.html)

Software for desktop development

- Code Composer Studio CCS 7.2+ installed with support for CC13xx/CC26xx devices (<http://www.ti.com/tool/ccstudio>)
- Sensor Controller Studio 1.4.1 + All Patches (<http://www.ti.com/tool/SENSOR-CONTROLLER-STUDIO>)
- CC2640R2F SDK (<http://www.ti.com/tool/SIMPLELINK-CC2640R2-SDK>) or CC13x0 SDK (<http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK>)
- For Bonus Task 1:
 - Bluetooth mobile app:
 - Android: BLE Scanner by Bluepixel Technology LLP - available on the Google Play store (<https://play.google.com/store/apps/details?id=com.macdom.ble.blescanner&hl=en>)
 - iOS: LightBlue Explorer - *Bluetooth®* Low Energy by Punch Through - available on the App Store (<https://itunes.apple.com/us/app/lightblue-explorer-bluetooth-low-energy/id557428110?mt=8>)

OR

- BTool (located in tools->blestack directory of the SimpleLink CC2640R2 SDK installation)
- For Bonus Task 2:
 - SmartRF Studio 7+ (<http://www.ti.com/tool/smartrfstudio>)

Hardware

- 1x CC2640R2F (<http://www.ti.com/tool/launchxl-cc2640r2>), CC1350 (<http://www.ti.com/tool/launchxl-cc1350>) or CC1310 (<http://www.ti.com/tool/launchxl-cc1310>) LaunchPad
- Micro-USB Cable for LaunchPad
- External variable voltage source, OR
 - Wire or jumper, see Getting started – Hardware ([l#getting-started-ndash-hardware](#))
- For bonus task1:
 - 1x CC2640R2F LaunchPad (<http://www.ti.com/tool/launchxl-cc2640r2>)
 - Mobile device for testing or any other Bluetooth client applications as described in Bluetooth Low Energy Fundamentals workshop ([./ble_01_basic/ble_01_basic.html](#))
- For bonus task2:
 - 2x CC1350 LaunchPad (<http://www.ti.com/tool/launchxl-cc1350>) or CC1310 LaunchPad (<http://www.ti.com/tool/launchxl-cc1310>)

Getting started – Desktop

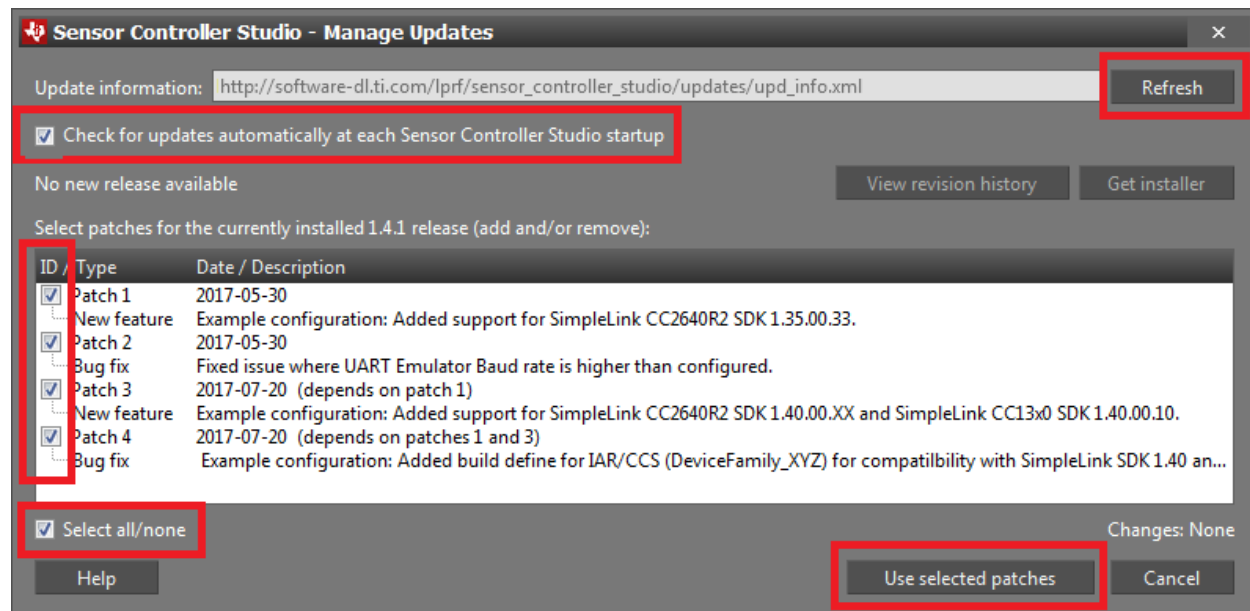
Install the software

1 - Run the Associated SDK installer (refer to table in introduction) to the default directory: C:\ti\

This gives you

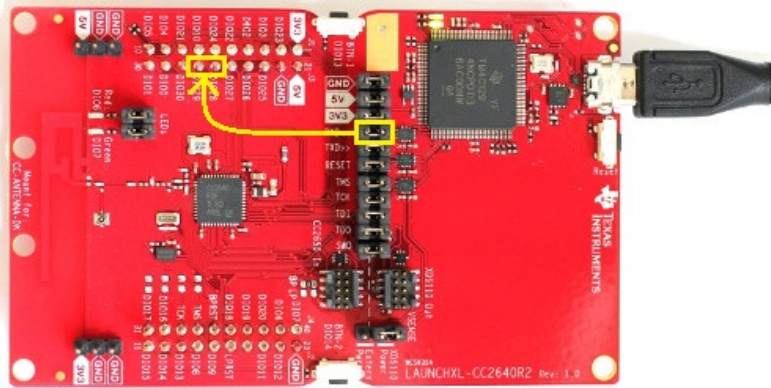
- For CC2640R2F: The SimpleLink CC2640R2 SDK at C:\ti\simplelink_cc2640r2_sdk_1_40_00_xx
- For CC1350 or CC1310: The SimpleLink CC13x0 SDK at C:\ti\simplelink_cc13x0_sdk_1_40_00_xx

2 - Install Sensor Controller Studio (<http://www.ti.com/tool/SENSOR-CONTROLLER-STUDIO>) and enable all patches by selecting Updates → Manage Updates.



Getting started – Hardware

Connect the CC2640R2F/CC1350/CC1310 LaunchPad to your computer via the micro-USB cable. **DI029** on LaunchPad will be the ADC input. Connect the external variable voltage source to **DI029**. As an alternative, a jumper or wire can be used. In this training **DI029** is connected to **DI028** or **DI030** via a jumper, borrowed from **RXD<<** on the LaunchPad. **DI028** will be set as digital output HIGH and **DI030** as digital output LOW, which will be used to test and verify the ADC driver and main application. Note that any jumper that does not interfere with the application can be used. See picture below for reference.



Task 1 – Import, build and download clean TI-RTOS Project

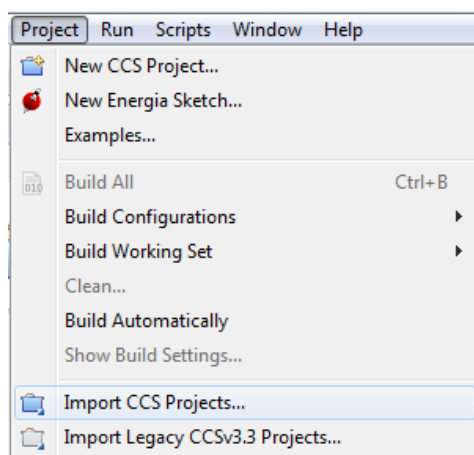
In this task you will import a clean TI-RTOS Driver project called `Empty` and run the program on your LaunchPad. This project will be used as a clean template to integrate the SC driver. It is a minimal TI-RTOS example project that has one single task that toggles the red LED on the LaunchPad once every second.

How to find your CCS workspace path

1. In CCS, go to `Project` → `Properties`, or press `Alt+Enter`.
2. Click on `Build` section to the left.
3. Navigate to the `Variables` tab.
4. Click on `Show system variables` checkbox at the bottom.
5. Scroll down until you find the `WORKSPACE_LOC` variable, and note the value. This is your CCS workspace path for your project.

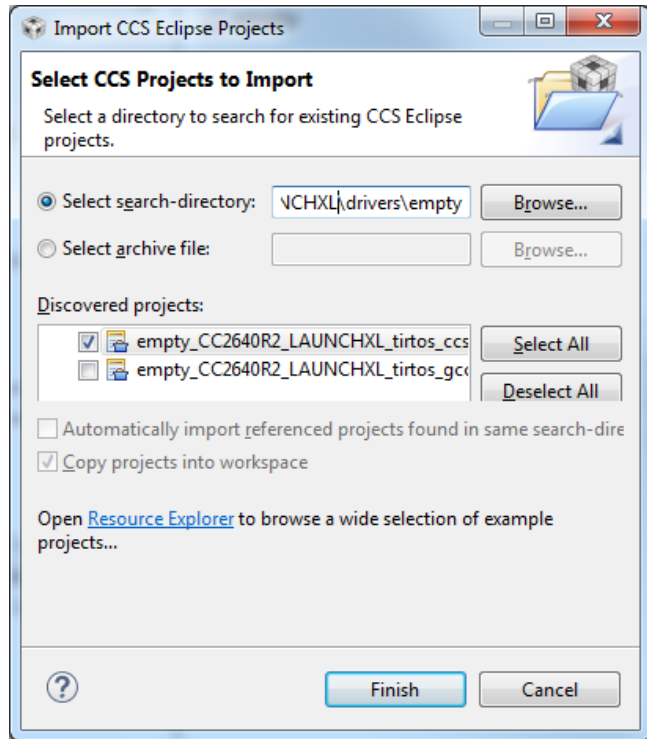
Do the following:

1. Open CCS and press `Project` → `Import CCS Project`



2. Import the `Empty` Project found in the SDK install path folder subdirectory shown in the table below.

Device	Project Path
CC2640R2F	<SDK>\examples\rtos\CC2640R2_LAUNCHXL\drivers\empty
CC1350	<SDK>\examples\rtos\CC1350_LAUNCHXL\drivers\empty
CC1310	<SDK>\examples\rtos\CC1310_LAUNCHXL\drivers\empty



1. Build and download the project to your LaunchPad by pressing **F11**. Then press **F8** to allow the program to run. The red LED should now toggle once every second.



If you failed to download and run the "Empty" Project

- Make sure you have connected the LaunchPad to your computer with a micro-USB Cable.
- Make sure that you have selected the correct project
 - called `empty_CC2640R2_LAUNCHXL_tirtos_ccs` for CC2640R2F.
 - called `empty_CC1350_LAUNCHXL_tirtos_ccs` for CC1350.
 - called `empty_CC1310_LAUNCHXL_tirtos_ccs` for CC1310.
- Unplug and plug in the USB cable again.
- You can try to run a forced mass erase operation with Smart RF Flash Programmer 2 and then try again.

Task 2 – Create and setup SCS project

In this task we will create a new Sensor Controller project in SCS.



SCS Documentation and Help

At any time in SCS, Help → Sensor Controller Studio Help or press **F1** for documentation and help.

Create SCS project

Do the following:

1. Start SCS and open a new project, File → New Project or Ctrl+N.
2. Set the Project Name to ADC Level Trigger.
3. Set the Operating system to TI-RTOS.
4. Set Source code output directory to ./.
5. Set Chip name to
 - CC2640R2F if using CC2640R2F.
 - CC1350 if using CC1350.
 - CC1310 if using CC1310.
6. Set Chip package to QFN48 7x7 RGZ.
7. Add one task by clicking Add new, name it adc level trigger.
8. Save the project, File → Save Project or Ctrl+S, to the CCS project base folder in your CCS workspace. See Task 1 on how to find your CCS workspace path.



Correct project file save path

The path set in Project file **must** be the CCS project base folder located in the CCS workspace.

Refer to the screen shot below for the CC2640R2F LaunchPad.

ADC Level Trigger

Project name:

Project file:

Project description:

Operating system:

Source code prefix:

Source code output directory:

Target Chip

Chip name:

Chip revision:

Chip package:

Compatible chips:

Device	Revision	Package	SCIF Driver	Task Testing
CC2620	-	QFN48 7x7 RGZ	Compatible	Not compatible
CC2630	-	QFN48 7x7 RGZ	Compatible	Not compatible

Sensor Controller Tasks

Task Name	Task ID / Execution Order
adc level trigger	0

Setup SC ADC Driver

Task Properties

We need to specify the resources to be used. Go to Task Properties for the adc level trigger task, which can be accessed by clicking on the task name in the directory on the left hand side, above the Initialization Code. Select the task resources in the list below.

- Analog Pins

- Create one pin and name it `ADC_INPUT`.
- **Digital Output Pins**
 - Create three pins and name them `GREEN_LED`, `LOW`, and `HIGH`. You can add more pins by clicking `Add I/O usage` to the right of the task resource name.
- **ADC**
- **System CPU Alert**
- **RTC-Based Execution Scheduling**



Unnecessary pins

If you are using an external variable voltage source connected to the ADC input, then the two digital pins `LOW` and `HIGH` are not needed.

Make sure the Task resource settings match the screenshot below:

The screenshot shows the Task resource settings window with the following configurations:

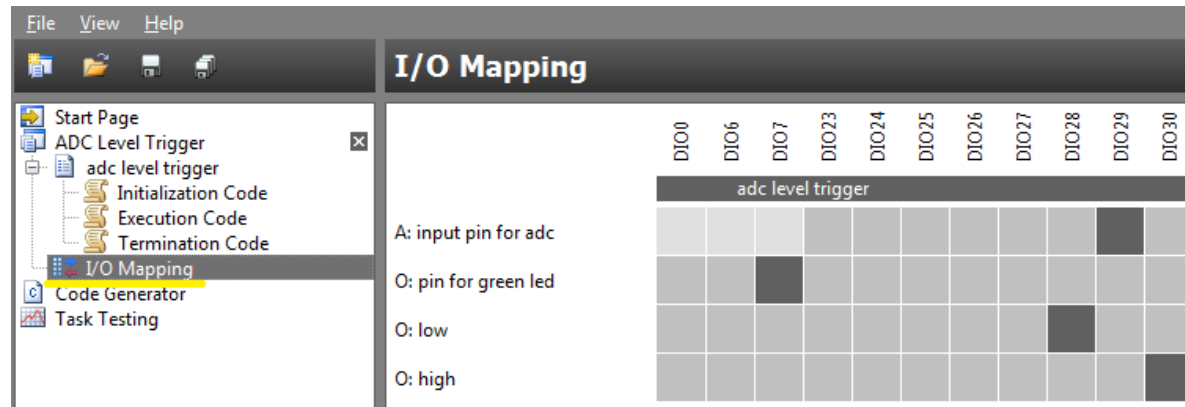
- Analog Open-Source Pins**: ☐ Analog + open-source pins
- Analog Pins**: ☒ General-purpose analog pins, for use with analog peripherals
 - ADC_INPUT**: ☒ Input to internal ADC
 - Pin count: 1 (accessed by constant)
- Differential Output Pins**: ☐ General-purpose digital differential output pin pairs
- Digital Input Pins**: ☐ General-purpose digital input pins, for manual polling
- Digital Open-Drain Pins**: ☐ General-purpose digital open-drain pins (driven low)
- Digital Open-Source Pins**: ☐ General-purpose digital open-source pins (driven high)
- Digital Output Pins**: ☒ General-purpose digital output pins
 - HIGH**: ☒ Digital Output High
 - Configuration on uninitialization: No pull
 - Output value on initialization: 0
 - Pin count: 1 (accessed by constant)
 - LOW**: ☒ Digital Output Low
 - Configuration on uninitialization: No pull
 - Output value on initialization: 0
 - Pin count: 1 (accessed by constant)
 - GREEN_LED**: ☒ Green LED on LP
 - Configuration on uninitialization: Pull-up
 - Output value on initialization: 0
 - Pin count: 1 (accessed by constant)
- Peripherals**
 - ADC**: ☒ Analog to digital converter
 - ☐ COMPA: Continuous time comparator
 - ☐ COMPB: Low-power clocked comparator
 - ☐ ISRC: Current source
 - ☐ TDC: Time to digital converter
 - ☐ Timer 0: Execution timing for use within task iterations
- Serial Interfaces**
- System CPU Communication**
 - ☐ Multi-Buffered Output Data Exchange: Seamless output buffer exchange with ALERT interrupt generation
 - ☐ Peripheral Sharing: Sharing of peripheral hardware modules with the System CPU
 - ☒ System CPU Alert: Basic ALERT interrupt generation from task code
- Task Event Handling**
- Task Execution**
 - ☒ RTC-Based Execution Scheduling: Use the Real-Time Clock to trigger the Execution Code
- Utilities**

I/O mapping

Go to the I/O Mapping window and set the following pins to:

- Analog pin `ADC_INPUT` to `DI029`.
- Digital pin `GREEN_LED` to `DI07`.
- Digital pin `LOW` to `DI028`.
- Digital pin `HIGH` to `DI030`.

The pin order can vary. It should look something like the picture below.



Task 3 – Create Sensor Controller Driver

In this task you will implement and test the ADC driver in Sensor Controller Studio.

Specify the RTC Trigger Period

The RTC period is specified by Minimum task iteration interval in preferences, File → Preferences or Ctrl+P. See picture below for reference. A RTC period of 250 milliseconds is more than enough for this training.

Debugging entry point	Start of task code
Debugging exit point	End of task code
Show firmware framework data structures	No
Task testing	
CSV file separation character	;
Log and display initial data structure values	Yes
Maximum task code execution time before triggering debug mode	1 seconds
Minimum task iteration interval	<u>250 milliseconds</u>
User file locations (leave empty to use default)	
Project file directory	



Task testing specific

Specifying the RTC period in SCS **only** affects the task testing in SCS. This does not set the RTC period when the SC driver is integrated into other projects or applications, as this is done through the functions `scifStartRtcTicks()` and then started with `scifStartRtcTicksNow()`.

RTC scheduling

The RTC period determines the execution interval of the SC task, if the SC task is scheduled by the RTC with the `fwScheduleTask()` function. `fwScheduleTask(N)` schedules the next task iteration in `N` RTC ticks. So for instance, let's say the RTC is operating at 200Hz. Scheduling the SC task with `fwScheduleTask(1);` would run the execution code at a 200Hz rate. With `fwScheduleTask(5);` the execution code would run at a $(200/5)=40\text{Hz}$ rate, and so on.

However, to minimize current consumption it is recommended to select the highest possible tickPeriod so that the argument in `fwScheduleTask()` can be as low as possible.



Quiz

If a given SC task is scheduled with `fwScheduleTask(2);`, and the RTC period is set to 250 ms, what is the SC task period?

125 ms

250 ms

500 ms

1000 ms

If a SC task period is 700 ms, and the RTC is set at 20 Hz, for which `n` is the SC task scheduled with the function `fwScheduleTask(N);` ?

N=7

N=10

N=14

N=20

If a SC task is scheduled with `fwScheduleTask(3);`, and the SC task period is 1500 ms, what is the RTC period?

2 Hz

3 Hz

5 Hz

6 Hz

Adding a data structure member

For a given SC Task, data structure members are used in a SC task to store variables in AUX RAM. This allows the SC Task to store data between task iterations, and to exchange data between the SC and the main application processor. There are four different types of data structures that represent different types of data, shown in the table below. All data structure types have the same format (16 bit word).

Data structure	Intended use
cfg	Configuration of SC Task
input	Input data for SC Task
output	Output data from SC Task
state	Internal state of SC Task

To add a data structure member, locate the **Data structures** box to the right in either code window. This is the same for both **Initialization Code**, **Execution Code**, and **Termination Code**. Click **Add**. Select which **Data structure**, and set **Member name** and **Value**. Optionally, but not necessary, give a **Description** and choose a **Type**. See example picture for ADC output value below.

Add Data Structure Member

Data structure:

Member name:

Description:

Type:

Value:

Array size:

OK, value is 0

OK Cancel



Quiz

A SC Task is keeping track of how many ADC readings it has done between each task iteration. In which data structure should the variable be placed in?

A SC task is set to notify the main application when a given threshold is crossed, determined by the main application. In which data structure should the threshold be placed in?

Implement the SC ADC driver

It is time to create the ADC driver. The Sensor Controller ADC driver shall measure the analog voltage on DIO28 periodically and set/clear the green LED when the measured input voltage is below/above a set threshold value. The SC will alert the main application when the threshold is crossed, and the main application (TI-RTOS application in this case) will set/clear the red LED if the ADC input value was above/below the threshold.

The threshold is specified by the main application, stored in a `cfg` structure data member. Being under/above the threshold sets the state to be either low/high. The state is stored in a state structure member. The analog sensor value is sampled by the ADC and stored in an output structure data member.



Create data structure members

You need to create three data structure members: `threshold` in `cfg` structure, `adcValue` in output structure, and `high` in state structure. See Task 3 on how to create data structure members.

Initialization code

The initialization code will run once when the task is started by the TI-RTOS application. The initialization code should set the digital values for the `HIGH` and `LOW` pins, select the sampling pin for the ADC, and schedule the first execution of the execution code to the next RTC tick. See Task 2 for a more detailed explanation on how RTC scheduling and how `fwScheduleTask()` works.

The initialization code is presented below. Copy and paste the code snippet into the `Initialization Code` in SCS.

```
// Set `DIO28` High
gpioSetOutput(AUXIO_O_HIGH);

// Set `DIO30` Low
gpioClearOutput(AUXIO_O_LOW);

// Select ADC input
adcSelectGpioInput(AUXIO_A_ADC_INPUT);

// Schedule the first execution
fwScheduleTask(1);
```

Initialization code

Execution code

The execution code should enable the ADC, sample and store the ADC value, compare the converted ADC input value with the threshold, and alert the main application if a transition happens. As explained in Task 3, it is scheduled through the `fwScheduleTask()` function every RTC tick.

Copy and paste the code snippet below into the `Execution Code` in SCS. **Note:** There is an intentional error in the code, see next section.

```
// Enable the ADC
adcEnableSync(ADC_REF_FIXED, ADC_SAMPLE_TIME_2P7_US, ADC_TRIGGER_MANUAL);

// Sample the analog sensor
adcGenManualTrigger();
adcReadFifo(output.adcValue);

// Disable the ADC
adcDisable();

U16 oldState = state.high;
if (input.adcValue > cfg.threshold) {
    state.high = 1; // High input -> High state
    gpioClearOutput(AUXIO_O_GREEN_LED);
} else {
    state.high = 0; // Low input -> Low state
    gpioSetOutput(AUXIO_O_GREEN_LED);
}

if (oldState != state.high) {
    // Signal the application processor.
    fwGenAlertInterrupt();
}

// Schedule the next execution
fwScheduleTask(1);
```

adc level trigger Execution Code


Termination code

The termination code will run once when the SC task is either terminated by the main application, or executed without scheduling. The main purpose for the termination code is to do necessary cleanup.

For this task, there is no necessary cleanup, and the termination code is therefore left empty.

Compilation Error

With the SC driver implemented, it is time for test. However, nobody makes perfect code. In the execution code above there is a subtle syntax error. At first glance it's not obvious where the error is, but SCS can help. Go to the `Task Testing` window, `View` → `Task Testing` or `Ctrl+T`. Select the `ADC Level Trigger` project if this is not already set.

You should now be present with an event log, such as the picture below. It shows the different stages of validating and compiling the project. This event log is only shown in this window  when something went wrong during validation and compiling. As we see at the bottom of the log, during compilation of the execution code, it encountered a syntax error at line 12. Note that the line number can vary. See picture below for reference.

Setup		Graph
Select project: ADC Level Trigger		Select task: adc level trigger
Event	Time / Line	Description
■ Selecting project	17:42:26.074	ADC Level Trigger
■ Validating project	17:42:26.074	ADC Level Trigger
■ Reading source template files	17:42:26.074	ADC Level Trigger
■ Validating task	17:42:26.104	adc level trigger
■ Processing task	17:42:26.104	adc level trigger
■ Compiling task code	17:42:26.104	Initialization Code
■ Compiling task code	17:42:26.114	Execution Code
■ Execution Code	12	Invalid syntax in expression

So, go back to the execution code window and find the specified line number. If you are observant, you should see that the code line

```
if (input.adcValue > cfg.threshold) {
```

is accessing the `adcValue` from the **input** structure, and not the **output** structure. A subtle bug indeed. Fix the line by changing `input` to `output`.

Now, when going back to the **Task Testing** window, the event log should no longer be there and you should be presented with the actual **Task Testing** screen.



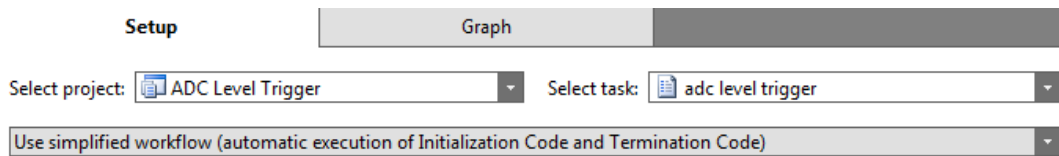
More code errors?

If you got more syntax errors present, then you might have forgotten to create or name the pins correctly in the **Task Properties** window, or to create or name the data structure members in the code windows. Go back and double check.

Sensor Controller Task Testing

Now we test the task to verify it works as intended. Do the following:

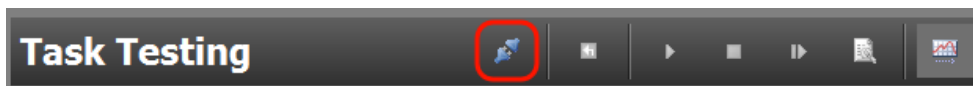
1. Go to the **Task Testing** window if not already there, **View** → **Task Testing** or **Ctrl+T**.
2. Select the **ADC Level Trigger** project if not already set.
3. Select the **Use simplified workflow**. See picture below.



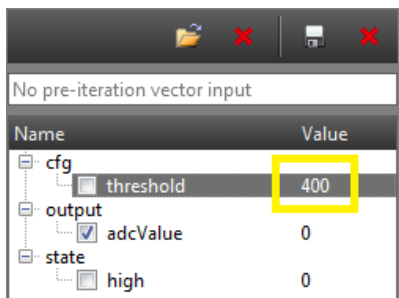
4. Add **Run Execution Code** to the **Task iteration action sequence**.



5. Click on **Connect** **F12**. You should now be in the graph tab.



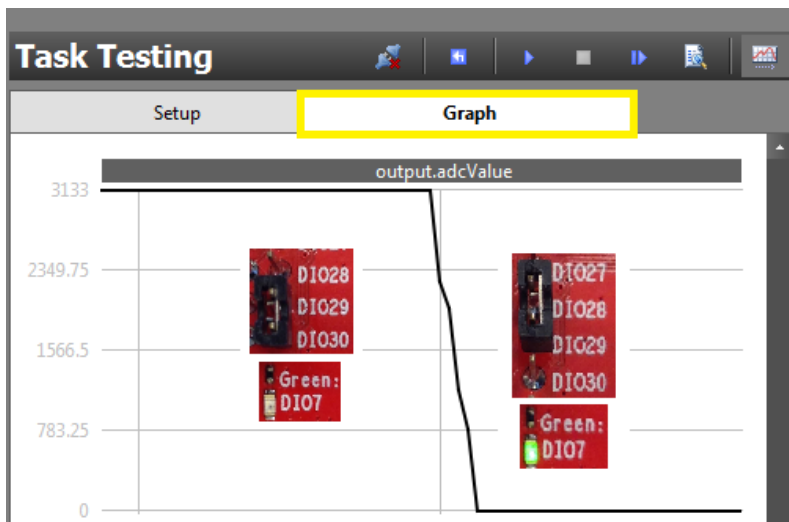
6. On the right side of the graph window you should find all data structure members created. If there are no variables you might have forgotten to create them. Configure the `cfg.threshold` value by double clicking the variable, set it to 400, and press **Enter**.



7. Then, click **Run Task Iterations Continuously** **F5**.



8. Select `output.adcValue` in the member struct box. A graph of the variable should appear.
9. Place the wire or jumper between `DIO29` and `DIO30`. The `output.adcValue` should be high and `state.high` should be 1. Now move the wire or jumper between `DIO28` and `DIO29`. The `output.adcValue` should drop down to about 0 and `state.high` should be 0. The green LED should also turn on. See the picture below on what you should observe.



10. When you are content that the driver is working, click `Disconnect` `F11`.

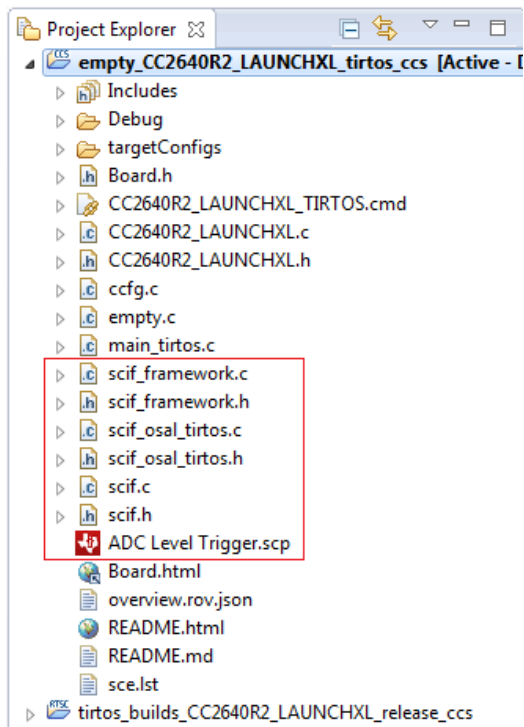


Generate Sensor Controller Driver

Now that everything is working well, the next step is to generate the driver from the SCS project.

Do the following:

1. Go to the `Code Generator` window, `View` → `Code Generator` or `Ctrl+G`.
2. Select `ADC Level Trigger` as `Current project`.
3. Either select `Output automatically` to auto generate code each time visiting the window, or click `Generate driver source code`.
4. Click `View output directory` and **double check** it is the project base folder of the `Empty Project`. The files should also be visible in the project explorer in CCS, shown in the picture below. Try refreshing the project `F5` if they do not show up.



5. Build the project and see that it compiles without errors.



SCIF files do not appear in CCS project folder?

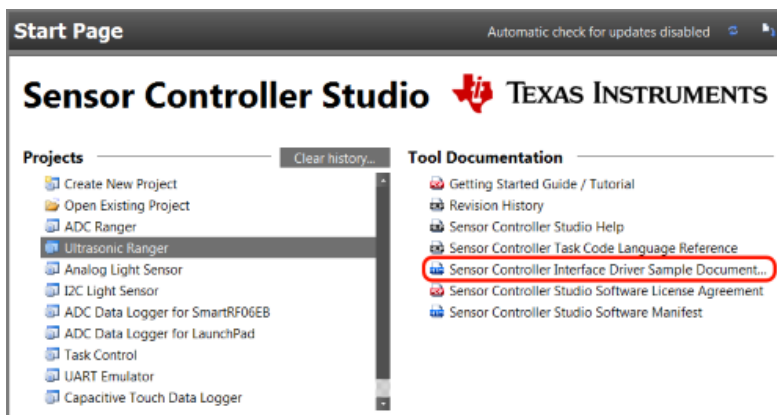
If no files are generated during code generation, then the output directory is most likely wrong. Go back to project settings and make sure **Project file** and **Source code output directory** path are correct. See Task 2 for help.

Task 4 – Implement TI-RTOS Application

Now we will write the TI-RTOS application and integrate the sensor controller driver you created in Task 3. This task will show you how to communicate with the SC, and the different ways to process the SC data, with focus on execution context.

Code documentation regarding the SC can be found at

- `scif_framework.h` file from code generation.
- SC Interface documentation (doxygen), found on the start page.




Refactoring and Cleanup TI-RTOS "empty" Project

Rename the single thread defined in `main_tirtos.c` called `mainThread` to a more descriptive and relevant name for this project. Refactor the function to `tirtosScThread`, by right click function and `Refactor` → `Rename...`, or click function and `Alt+Shift+R`.



TI-RTOS Task and POSIX Thread

The TI-RTOS example use POSIX threads instead of TI-RTOS tasks, but the POSIX thread has an underlying Task object and has support for many of the TI-RTOS kernel task APIs. This workshop will refer to the two terms, thread and task, interchangeably. To summarize, a POSIX thread in TI-RTOS is practically a TI-RTOS task and a sensor controller task is not related to this as it describes a small program running on the sensor controller. Click here for more info about use of POSIX Thread in TI-RTOS (http://processors.wiki.ti.com/index.php/SYS/BIOS_POSIX_Thread_%28pthread%29_Support)

Also, replace the main loop  in the newly renamed `tirtosScThread()` task with an empty loop `while (1) {}`.

I/O configuration

The green LED is controlled by the SC. Because of this the TI-RTOS application GPIO Driver must not initialize and open the green LED pin. To fix this, go to CCS and comment out the `GPIOCC26XX_DIO_07` line in `gpioPinConfigs`, `CC2640R2_LAUNCHXL_PIN_GLED` line in `BoardGpioInitTable` and `CC2640R2_LAUNCHXL_GPIO_LED_GREEN` in `CC2640R2_LAUNCHXL_GPIOName`. See code snippets below for reference (do not copy and paste, but comment out lines with `Ctrl+Shift+/*`).

For CC2640R2:

```
GPIO_PinConfig gpioPinConfigs[] = {
    /* Input pins */
    GPIOCC26XX_DIO_13 | ..., /* Button 0 */
    GPIOCC26XX_DIO_14 | ..., /* Button 1 */

    /* Output pins */
    // GPIOCC26XX_DIO_07 | ..., /* Green LED */
    GPIOCC26XX_DIO_06 | ..., /* Red LED */
};
```

gpioPinConfigs in `CC2640R2_LAUNCHXL.c`

```
const PIN_Config BoardGpioInitTable[] = {
    CC2640R2_LAUNCHXL_PIN_RLED | PIN_GPIO_OUTPUT_EN | ...
    // CC2640R2_LAUNCHXL_PIN_GLED | PIN_GPIO_OUTPUT_EN | ...
    ....
    PIN_TERMINATE
};
```

BoardGpioInitTable in `CC2640R2_LAUNCHXL.c`

```
typedef enum CC2640R2_LAUNCHXL_GPIOName {
    CC2640R2_LAUNCHXL_GPIO_S1 = 0,
    CC2640R2_LAUNCHXL_GPIO_S2,
    // CC2640R2_LAUNCHXL_GPIO_LED_GREEN,
    CC2640R2_LAUNCHXL_GPIO_LED_RED,
    CC2640R2_LAUNCHXL_GPIOCOUNT
} CC2640R2_LAUNCHXL_GPIOName;
```

CC2640R2_LAUNCHXL_GPIOName in `CC2640R2_LAUNCHXL.h`

For CC13x0:

```
GPIO_PinConfig gpioPinConfigs[] = {
    /* Input pins */
    GPIOCC13XX_DIO_13 | ..., /* Button 0 */
    GPIOCC13XX_DIO_14 | ..., /* Button 1 */

    /* Output pins */
    // GPIOCC13XX_DIO_07 | ..., /* Green LED */
    GPIOCC13XX_DIO_06 | ..., /* Red LED */
};
```

gpioPinConfigs in CC1310_LAUNCHXL.c or CC1350_LAUNCHXL.c

```
const PIN_Config BoardGpioInitTable[] = {
    CC1310_LAUNCHXL_PIN_RLED | PIN_GPIO_OUTPUT_EN | ...
    // CC1310_LAUNCHXL_PIN_GLED | PIN_GPIO_OUTPUT_EN | ...
    ....
    PIN_TERMINATE
};
```

BoardGpioInitTable in CC1310_LAUNCHXL.c or CC1350_LAUNCHXL.c

```
typedef enum CC1310_LAUNCHXL_GPIOName {
    typedef enum CC1310_LAUNCHXL_GPIOName {
        CC1310_LAUNCHXL_GPIO_S1 = 0,
        CC1310_LAUNCHXL_GPIO_S2,
    // CC1310_LAUNCHXL_GPIO_LED_GREEN,
        CC1310_LAUNCHXL_GPIO_LED_RED,
        CC1310_LAUNCHXL_GPIOCOUNT
    } CC1310_LAUNCHXL_GPIOName;
```

CC13x0_LAUNCHXL_GPIOName in CC1310_LAUNCHXL.h or CC1350_LAUNCHXL.h

SC driver interaction and processing

There are two aspects with SC interaction and processing that needs to be addressed before implementation: how the SC and main application interact, and how to exchange data.

SC interaction

SC interaction is exclusively intended to be done through the two interrupt callbacks `Control READY` and `Task ALERT`. This way the SC can signal the main application at appropriate times.

The `Control READY` interrupt is signaled when one of the non-blocking task control functions, such as `scifStartTasksNbl()`, has executed successfully on the SC. However, the interrupt is encapsulated in the `scifWaitOnNbl()` function, and is in most cases, such as this training, not necessary to handle.

The `Task ALERT` interrupt is signaled when a SC task calls one of the functions `fwGenAlertInterrupt()`, `fwGenQuickAlertInterrupt()` or `fwSwitchOutputBuffer()`, as seen done in Task 3. This interrupt is used to signal the main application when some type of computation is finished, or some event has happened. This is what is interesting for us, as this allows us to free the main application from waiting for the SC driver to finish or some certain SC event to happen.

In the SC initialization part for this project, we handle both signals by registering a callback for each interrupt. This is only done for the sake of completeness. As only the `Task ALERT` signal is of importance for this project, we are only processing that signal.



Quiz

Which interrupt signal(s) is associated with the task control function `scifStopTasksNb1()`, issued from the main application?

Which interrupt signal(s) can a SC Task explicitly generate through procedure calls?

SC task data structure

See Task 3 for an overview of the data structures.

Accessing data stored in the SC AUX RAM can be done in two ways: either direct or indirect access. Which type of access is determined by the type of data structures used: single or multi-buffered. The only requirement is for multi-buffered data, where only indirect access can be used. Single-buffered can use either way. However, direct access is preferred as this is much faster.

Direct access is done through the variable `scifTaskData` structure, defined in `scif.h`. The hierarchy in which the data structure members are stored are as such: `scifTaskData` as base, next is the name of the SC task in camelCase, then the type of the data structure, and lastly the name of the data structure member.

For instance, the data structure member `adcValue`, located in the `output` structure, defined in the SC task `ADC`, is accessed as such: `scifTaskData.adcLevelTrigger.output.adcValue`. For a complete view of the `scifTaskData` structure, see the `scif.h` file.

The hierarchy and structure of the `scifTaskData` struct is highly dependent on the SC task implementation, and will vary from SC driver to SC driver.

Indirect access can be done through the Task data structure access functions defined in the `scif_framework.h` file, mainly by the `scifGetTaskStruct` function. By specifying the SC Task ID and the structure type, the relevant structure pointer is returned. As mentioned, this must be used to access multi-buffered data structures.



Quiz

A data structure member is defined in a SC Task, and is called `counter`. It is stored in the `output` structure. The SC Task is called `Pin Counter`. What is the resulting direct access code?

Initialize and setup of SC driver

To initialize and setup the SC driver a couple of necessary steps need to be done. First, add

```
#include "scif.h"
#define BV(x)    (1 << (x))
```

Scif header and macro

at the top of the `empty.c` file. The `scif.h` file is the main interface to the SC driver compiled and generated from the SCS project. `BV()` is a bit vector macro, which will be useful soon.

Create callback functions

The initialization code for the SC driver uses two callback functions to handle the two interrupt signals Task ALERT and Control READY . Therefore, copy and paste the code snippet above the `tirtosScThread()` , shown below.

```
void scCtrlReadyCallback(void)
{

} // scCtrlReadyCallback

void scTaskAlertCallback(void)
{

} // scTaskAlertCallback
```

SC callback functions

Initialize driver and register callbacks

Then, in the `tirtosScThread()` , before the main loop `while(1)` , copy and paste the following code snippet. It is important to note that the initialization code should preferably be run in a TI-RTOS context, such as a task context.

```
// Initialize the Sensor Controller
scifOsalInit();
scifOsalRegisterCtrlReadyCallback(scCtrlReadyCallback);
scifOsalRegisterTaskAlertCallback(scTaskAlertCallback);
scifInit(&scifDriverSetup);

// Set the Sensor Controller task tick interval to 1 second
uint32_t rtc_Hz = 1; // 1Hz RTC
scifStartRtcTicksNow(0x00010000 / rtc_Hz);

// Configure Sensor Controller tasks
scifTaskData.adcLevelTrigger.cfg.threshold = 600;

// Start Sensor Controller task
scifStartTasksNbl(BV(SCIF_ADC_LEVEL_TRIGGER_TASK_ID));
```

SC Driver Initialization

So what does the code do? Let's study each code line.

```
// Initialize the Sensor Controller
scifOsalInit();
scifOsalRegisterCtrlReadyCallback(scCtrlReadyCallback);
scifOsalRegisterTaskAlertCallback(scTaskAlertCallback);
scifInit(&scifDriverSetup);
```

Driver specific initialization

The first line simply initializes the OSAL of the `scif` framework. The next two lines registers two callbacks for the two interrupt signals Control READY and Task ALERT from the SC. This is how the main application can communicate and work together with the SC, explained in the Section above. The fourth line initializes the SC with our SC task driver created in Task 3.

```
// Set the Sensor Controller task tick interval to 1 second
uint32_t rtc_Hz = 1; // 1Hz RTC
scifStartRtcTicksNow(0x00010000 / rtc_Hz);
```

RTC initialization

The next line configures the RTC tick interval. This is more thoroughly explained in Task 3.

For this training we are setting the RTC interval to 1 second. However, feel free to play around with the RTC interval by modifying the `rtc_Hz` variable or the function argument directly. The function `scifStartRtcTicksNow()` takes a 32-bit value as an argument. The argument represents a tick interval, where bits 31:16 are the seconds, and bits 15:0 are the 1/65536 of a second.

```
// Configure Sensor Controller tasks
scifTaskData.adcLevelTrigger.cfg.threshold = 600;

// Start Sensor Controller task
scifStartTasksNbl(BV(SCIF_ADC_LEVEL_TRIGGER_TASK_ID));
```

Task initialization

Next line is the SC task configuration. This part is optional, and can be done at multiple appropriate times depending on the SC task. We configure the `cfg.threshold` variable, just as we did in Task 3. The SC task configuration can practically be done at any time in the main application, but is highly recommended to be done at appropriate times, such as during initialization or before SC task execution.

The last line starts the actual execution of the SC task. All `scif` functions handling task IDs, such as `scifStartTasksNbl()`, takes a bit vector as an argument. The macro `BV()` performs this transformation, and this is why it was included. The SC Task IDs can be found in the generated `scif.h` file.



Quiz

Which statement(s) are true?

One can only set a `cfg` data member before the SC task is started.

One must register callbacks for both the Control READY and Task ALERT signals.

RTC does not always have to be configured with the `scifStartRtcTickNow()` function.

Multiple SC tasks can be active at the same time, but the tasks only run one at a time (single processor core).

Implement SC driver application processing

Now we can begin implementing the SC processing. We will do this in increments, implementing it in different application contexts and methods, and at the end discuss the pros and cons for each solution.

All three solutions are interrupt based, where the idea is the same:

1. Wait for a Task ALERT signal.
2. Clear the interrupt source.
3. Process the SC task.
4. Acknowledge the ALERT event to the `scif` framework.

The actual task processing shall be encapsulated in a function called `processTaskAlert()`. Copy and paste the code shown below.

```
void processTaskAlert(void)
{
    // Clear the ALERT interrupt source
    scifClearAlertIntSource();

    // Do SC Task processing here

    // Acknowledge the ALERT event
    scifAckAlertEvents();
} // processTaskAlert
```

SC Task Alert Handling

The processing is simple: fetch the `state.high` variable and copy the value to the Red LED pin. See Task 4 on how to access SC data. Copy and paste the code snippet below in `processTaskAlert()` above, where the SC task processing is marked.

```
// Fetch 'state.high' variable from SC
uint8_t high = scifTaskData.adcLevelTrigger.state.high;
// Set Red LED state equal to the state.high variable
GPIO_write(Board_GPIO_RLED, high);
```

adc level trigger processing

This will not run just yet, as we need to connect the processing to the interrupt signal. Below are the three different solutions presented.

Solution 1 – HWI

For the first solution the SC task alert handling and processing will be done in the `scTaskAlertCallback()` function. The `scTaskAlertCallback()` is executed in a HWI context, as the title suggests. So in `scTaskAlertCallback()`, call the `processTaskAlert()` function. See code snippet below.

```
void scTaskAlertCallback(void)
{
    // Call process function
    processTaskAlert();
} // scTaskAlertCallback
```

HWI SC Task alert process

Build and debug the project. Move the wire or jumper between `DI028` and `DI030` on the LaunchPad. You should see the green and red LED toggle in-between each other.

The application is now doing nothing until the SC task generates a `Task ALERT` signal. The HWI callback `scTaskAlertCallback()` is then called. In the HWI context the callback clears the interrupt source, read the state variable `state.high`, set or clears the red LED, and acknowledges the ALERT event.

Solution 2 – SWI

Doing extensive processing in a HWI process is not advised, as it blocks other high priority processes while running. For this solution, we will move the processing to a SWI process, and the HWI process will signal the SWI process.

At the top of the `empty.c` file, add the following

```
#include <ti/sysbios/knl/Swi.h>

// SWI Task Alert
Swi_Struct swiTaskAlert;
Swi_Handle hSwiTaskAlert;

// Function prototype
void processTaskAlert(void);

void swiTaskAlertFxn(UArg a0, UArg a1)
{
    // Call process function
    processTaskAlert();
} // swiTaskAlertFxn
```

SWI variables and function

In the top of the `tirtosScThread` task, add the following code snippet to initialize the SWI process.

```
// SWI Initialization
Swi_Params swiParams;
Swi_Params_init(&swiParams);
swiParams.priority = 3;
Swi_construct(&swiTaskAlert, swiTaskAlertFxn, &swiParams, NULL);
hSwiTaskAlert = Swi_handle(&swiTaskAlert);
```

SWI initialization

Now, copy and paste the following into `scTaskAlertCallback()`. Whenever the `Task ALERT` signal is raised, the HWI callback will signal the SWI process, where the entirety of the processing will be done.

```
void scTaskAlertCallback(void)
{
    // Post a SWI process
    Swi_post(hSwiTaskAlert);
} // scTaskAlertCallback
```

SWI signalling

Again, build and debug the project. Move the wire or jumper on the LaunchPad. You should observe the same behavior from the HWI solution.

But why was this whole process of moving the task processing to SWI context necessary if the behavior was the same? This would be much more obvious if the actual task processing was much more extensive computational wise. The current task is substantially small, and therefore will not impact the execution whatever the context it runs in. It is still advised to run the task processing at least in SWI context.

Solution 3 – Task (Thread)

Next solution is to move the processing in a task context. The idea is to signal the main task when the HWI is run, and then in turn run the processing in the task. We will use a Semaphore to synchronize the signaling.



Remove SWI Code

The SWI relevant code, such as the struct and handle variables, `swiTaskAlertFxn()` function, and initialization, can now be commented out or removed. This does **not** include the `processTaskAlert()` function.

First up, at the top of the `empty.c` file, add the relevant header file and variables.

```
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/BIOS.h>

// Main loop Semaphore
Semaphore_Struct semMainLoop;
Semaphore_Handle hSemMainLoop;
```

Semaphore variables

Next, in the top of the `tirtosScThread` task, initialize the semaphore struct and store the handle.

```
// Semaphore initialization
Semaphore_Params semParams;
Semaphore_Params_init(&semParams);
Semaphore_construct(&semMainLoop, 0, &semParams);
hSemMainLoop = Semaphore_handle(&semMainLoop);
```

Semaphore initialization

Now, in the `scTaskAlertCallback()` function, you will post to the Semaphore which should trigger the execution in the `tirtosScThread` while-loop.

```
void scTaskAlertCallback(void)
{
    // Post to main loop semaphore
    Semaphore_post(hSemMainLoop);
} // scTaskAlertCallback
```

Semaphore signalling

Now in the main loop in `tirtosScThread()`, we wait on the Semaphore indefinitely and afterwards call `processTaskAlert()`, see code snippet below.

```
while (1) {  
    // Wait on sem indefinitely  
    Semaphore_pend(hSemMainLoop, BIOS_WAIT_FOREVER);  
  
    // Call process function  
    processTaskAlert();  
}
```

tirtosScThread while-loop

Build and debug the project. Move the wire or jumper on the LaunchPad. The same behavior again should be observed. This way of using the HWI process to signal a task process (or a SWI process in solution 2) is the way to go. Not only does it free up high priority processes, but it also streamlines the application processing, which is important in bigger applications.

Expand Complete Code Solution with processing in Task/thread.



main_tirtos.c:

```

/*
 * ===== main_tirtos.c =====
 */
#include <stdint.h>

/* POSIX Header files */
#include <pthread.h>

/* RTOS header files */
#include <ti/sysbios/BIOS.h>

/* Example/Board Header files */
#include "Board.h"

extern void *tirtosScThread(void *arg0);

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

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

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

    /* Set priority and stack size attributes */
    pthread_attr_init(&pAttrs);
    priParam.sched_priority = 1;

    detachState = PTHREAD_CREATE_DETACHED;
    retc = pthread_attr_setdetachstate(&pAttrs, detachState);
    if (retc != 0) {
        /* pthread_attr_setdetachstate() failed */
        while (1);
    }

    pthread_attr_setschedparam(&pAttrs, &priParam);

    retc |= pthread_attr_setstacksize(&pAttrs, THREADSTACKSIZE);
    if (retc != 0) {
        /* pthread_attr_setstacksize() failed */
        while (1);
    }

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

    BIOS_start();

    return (0);
}

```

empty.c:

```

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

```

```

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

/* Driver Header files */
#include <ti/drivers/GPIO.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"

#include "scif.h"
#define BV(x) (1 << (x))
#include <ti/sysbios/kl/Swi.h>
#include <ti/sysbios/kl/Semaphore.h>
#include <ti/sysbios/BIOS.h>

// Main loop Semaphore
Semaphore_Struct semMainLoop;
Semaphore_Handle hSemMainLoop;

// SWI Task Alert
Swi_Struct swiTaskAlert;
Swi_Handle hSwiTaskAlert;

void processTaskAlert(void)
{
    // Clear the ALERT interrupt source
    scifClearAlertIntSource();

    // Do SC Task processing here
    // Fetch 'state.high' variable from SC
    uint8_t high = scifTaskData.adcLevelTrigger.state.high;
    // Set Red LED to the state variable
    GPIO_write(Board_GPIO_RLED, high);

    // Acknowledge the ALERT event
    scifAckAlertEvents();
} // processTaskAlert

void scCtrlReadyCallback(void)
{
} // scCtrlReadyCallback

void scTaskAlertCallback(void)
{
    // Post to main loop semaphore
    Semaphore_post(hSemMainLoop);
} // scTaskAlertCallback

/*
 * ===== mainThread =====
 */
void *tirtosScThread(void *arg0)
{
    // Semaphore initialization
    Semaphore_Params semParams;

```

```

Semaphore_Params_init(&semParams);
Semaphore_construct(&semMainLoop, 0, &semParams);
hSemMainLoop = Semaphore_handle(&semMainLoop);

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

// Initialize the Sensor Controller
scifOsaiInit();
scifOsaiRegisterCtrlReadyCallback(scCtrlReadyCallback);
scifOsaiRegisterTaskAlertCallback(scTaskAlertCallback);
scifInit(&scifDriverSetup);

// Set the Sensor Controller task tick interval to 1 second
uint32_t rtc_Hz = 1; // 1Hz RTC
scifStartRtcTicksNow(0x00010000 / rtc_Hz);

// Configure Sensor Controller tasks
scifTaskData.adcLevelTrigger.cfg.threshold = 600;

// Start Sensor Controller task
scifStartTasksNbl(BV(SCIF_ADC_LEVEL_TRIGGER_TASK_ID));

while (1) {
    // Wait on sem indefinitely
    Semaphore_pend(hSemMainLoop, BIOS_WAIT_FOREVER);

    // Call process function
    processTaskAlert();
}
}

```

 Add Watch Expression

 Add Watch Expression Dialog

Solutions summary

We have now implemented and tested three different solutions.

HWI Context: This gave us the fastest possible response time, as the actual processing was done as close to the Task ALERT signal as possible. However, this is never advised, as any HWI process should be kept to minimal execution time and not block other high priority processes from executing.

SWI Context: This solution does not have as fast response time as the HWI solution, since the HWI process must signal the SWI process. This is however preferred over the HWI solution, as it does not block other high priority processes.

Task Context: This solution can result in slower response time than the SWI context solution, but does not block other high priority processes. This solution can safely scale with increasingly more complex projects.

Solution two (SWI) is a viable method, but solution 3 (Task) is the preferred solution in most cases. It is flexible as the task priority can be adjusted, scalable with complex projects and can handle computational heavy processing without blocking time-critical high priority processes or interrupts.



Quiz

Which context(s) gives the fastest response time?

Which context(s) are preferred for computational heavy processing?

Which context(s) is the **NOT** recommended for computational heavy processing?



Bonus Tasks 1 – Integrate with BLE (CC2640R2F)

In this training, we are porting our simple application into the BLE project `Project Zero`. It is meant to show you how the application would operate within a bigger and more complex project. We are going to use the already existing service `Data Service` to communicate the current state of the ADC input, high or low.



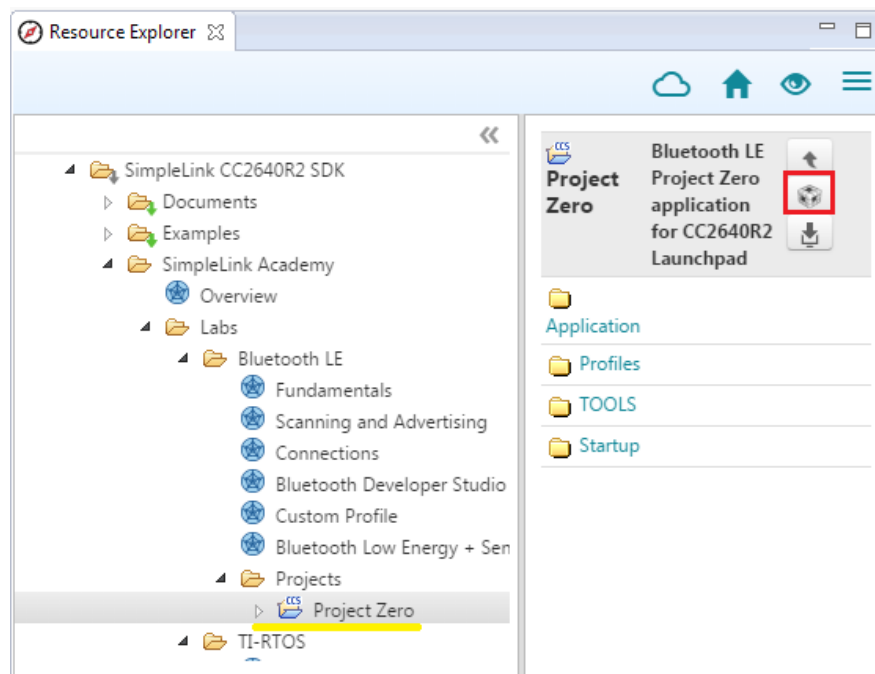
Hardware requirement

This bonus task requires a SimpleLink™ *Bluetooth®* low energy CC2640R2F wireless MCU LaunchPad™ development kit (LAUNCHXL-CC2640R2) (<http://www.ti.com/tool/launchxl-cc2640r2>).

Import and modify Project Zero

First, import a fresh version of `Project Zero` from the Resource Explorer in CCS. It can be found in the `SimpleLink Academy` package, under the `Bluetooth LE` section. Expand box below for reference.

Resource Explorer: Project Zero App and Stack project path



Do the following:

- Copy and paste (you can drag and drop with the mouse in CCS) all 6 `scif` related code files from the `empty` project into the `Application/` folder in `Project Zero`.
- Open the `project_zero.c` file.
- Add `#include "scif.h"` at the top.
- Add `APP_MSG_SC_CTRL_READY` and `APP_MSG_SC_TASK_ALERT` enum in the `app_msg_types_t` enum typedef.
- Find the `ledPinTable[]` array. Comment out the `Board_GLED` member. Remember that this is the LED that the SC task controls.
- Add the following function declarations

```
// Sensor Controller functions
static void scCtrlReadyCallback(void);
static void scTaskAlertCallback(void);
static void processTaskAlert(void);
```

Sensor Controller Function Declarations

- Copy and paste the SC Driver initialization into `ProjectZero_taskFxn()`.
- In `user_processApplicationMessage()`, add the following case in the switch statement
 - Note that a case for `APP_MSG_SC_CTRL_READY` is not added because it is not needed in this application.

```
case APP_MSG_SC_TASK_ALERT:
    processTaskAlert();
    break;
```

Sensor Controller Switch Case

- Add the following functions

```
static void scCtrlReadyCallback(void)
{
    // Notify application `Control READY` is active
    user_enqueueRawAppMsg(APP_MSG_SC_CTRL_READY, NULL, 0);
} // scCtrlReadyCallback

static void scTaskAlertCallback(void)
{
    // Notify application `Task ALERT` is active
    user_enqueueRawAppMsg(APP_MSG_SC_TASK_ALERT, NULL, 0);
} // scTaskAlertCallback

static void processTaskAlert(void)
{
    // Clear the ALERT interrupt source
    scifClearAlertIntSource();

    // Get 'state.high', and set highStr to appropriate string
    uint16_t high = scifTaskData.adcLevelTrigger.state.high;
    char *highStr = (high != 0) ? "HIGH" : "LOW";
    // Set the highStr to the String characteristic in Data Service
    DataService_SetParameter(DS_STRING_ID, strlen(highStr), highStr);

    // Set/clear red LED.
    PIN_setOutputValue(ledPinHandle, Board_GPIO_RLED, high);

    // Acknowledge the ALERT event
    scifAckAlertEvents();
} // processTaskAlert
```

Sensor Controller Functions

- Build and debug the project. The first build may take a while.

Test BLE device

There is only a specific guide here for using the iOS app LightBlue Explorer, but any of the other test solutions described in Bluetooth Low Energy Fundamentals workshop ([../ble_01_basic/ble_01_basic.html](#)) can also be used.

Expand instructions for connecting using LightBlue Explorer

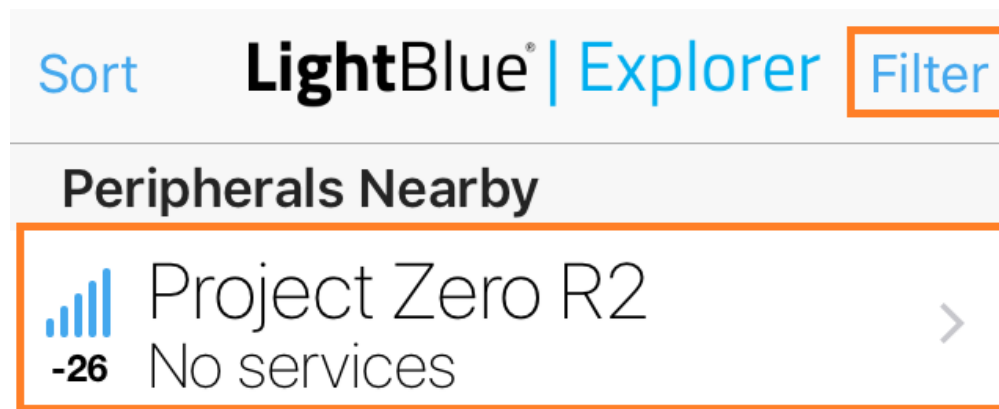
1. Start LightBlue Explorer

Using your iOS device, find the LightBlue Explorer app and open it.



2. Scan for BLE devices

The app should begin scanning for BLE devices automatically but you can refresh the list by pulling down.



You should see Project Zero advertising as "Project Zero R2". Connect to the device by clicking on the name. You can press the filter button in the top right corner to filter out all devices out of reach by setting a RSSI threshold (for example -50 dBm).

[< Back](#)

Peripheral

[Clone](#)

Project Zero R2

UUID: 837F89EA-C3BD-415E-98A8-A5255CC23146

Connected

ADVERTISEMENT DATA [Show](#)

Device Information

System ID

<704faa00 002d0798>

SCROLL DOWN



Model Number String

Model Number



Serial Number String

Serial Number



Firmware Revision String

3. **Find Data Service (UUID 1130)** Scroll down until you find the data service and open the string characteristic which is marked with an orange box below.



[< Back](#)

Peripheral

[Clone](#)

0xF0001111-0451-4000-B000-000000000000 >

Properties: Read Write

0xF0001112-0451-4000-B000-000000000000 >

Properties: Read Write

UUID: F0001120-0451-4000-B000-000000000000

0xF0001121-0451-4000-B000-000000000000 >

Properties: Read Notify

0xF0001122-0451-4000-B000-000000000000 >

Properties: Read Notify

UUID: F0001130-0451-4000-B000-000000000000


0xF0001131-0451-4000-B000-000000000000 >

Properties: Read Write

0xF0001132-0451-4000-B000-000000000000 >

Properties: Write Without Response Notify

4. Read String Characteristic In the top right corner, change the format to UTF-8 and then press the "Read again" button to read the current characteristic value. This value will mirror the state.high variable in the sensor controller driver taht indicate either HIGH or LOW input value onn the ADC input pin. Move the header to change the ADC input between high and low input and press the "Read again" button to retrieve the new state.


 **0xF0001131-0451-40...** **UTF-8**

Project Zero R2
0xF0001131-0451-4...
UUID: F0001131-0451-4000-
B000-000000000000
Connected


READ VALUES

Read again

"LOW"
15:19:38.009

 **Green LED**

"HIGH"
15:19:10.920

 **Red LED**

WRITTEN VALUES

Write new value

Bonus Tasks 2 – Integrate with Proprietary RF (CC1350 or CC1310)

In this training, we are again porting our simple application. However, now we are connecting it with Proprietary RF instead of BLE. We are modifying an already existing Proprietary Tx example project, called **RF Packet TX**. The application will be the transmitter, reading and storing the down state in a RF packet, while the second device will be the receiver, using SmartRF Studio to listen for the packet.



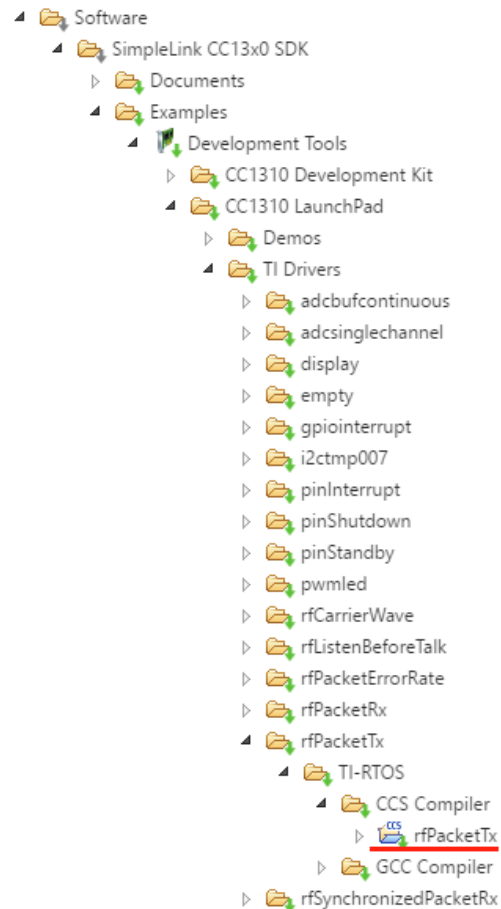
Hardware requirement

This bonus task requires **two** CC1350 or CC1310 LaunchPad; one for Tx and one for Rx.

Import and modify RF Packet TX

First, import a fresh version of the project RF Packet TX from the Resource Explorer in CCS. It can be found in the SimpleLink CC13x0 SDK package. Expand box below for reference.

Resource Explorer: RF Packet TX project path



In CCS, do the following:

- Copy and paste all 6 `scif` related code files from Empty (Minimal) Project into the project base folder in RF Packet TX.
- Open up `rfPacketTx.c` in CCS.
- Copy and paste the following code snippet at the top of the file.

```
#include <string.h>    // strlen() and memcpy()
#include <ti/sysbios/knl/Semaphore.h>
#include "scif.h"

#define BV(x)    (1 << (x))

Semaphore_Struct semMainLoop;
Semaphore_Handle hSemMainLoop;
```

Header Includes and Variable Declarations

- Go to the already existing `pinTable[]` array and change the first PIN instance from `Board_LED1` to `Board_LED0`. Remember that `LED1` is controlled by the SC.
- Copy and paste the SC callbacks.

```
void scCtrlReadyCallback(void)
{
    // Do nothing
} // scCtrlReadyCallback

void scTaskAlertCallback(void)
{
    // Signal main loop
    Semaphore_post(hSemMainLoop);
} // scTaskAlertCallback
```

SC Callbacks

- In `TxTask_init()`, add initialization for the semaphore

```
// Main loop Semaphore initialization
Semaphore_Params semParams;
Semaphore_Params_init(&semParams);
semParams.mode = Semaphore_Mode_BINARY;
Semaphore_construct(&semMainLoop, 0, &semParams);
hSemMainLoop = Semaphore_handle(&semMainLoop);
```

Semaphore Initialization

- In `txTaskFunction()`, copy and paste the SC Driver initialization at the top of the function.
- At the bottom of `txTaskFunction()`, replace the whole main loop with the code below

```

// Main loop
while(1) {
    // Wait for signal
    Semaphore_pend(hSemMainLoop, BIOS_WAIT_FOREVER);

    // Clear the ALERT interrupt source
    scifClearAlertIntSource();

    // Get 'state.high', and set highStr to appropriate string
    uint16_t high = scifTaskData.adcLevelTrigger.state.high;
    char *highStr = (high != 0) ? "HIGH" : "LOW";
    uint16_t highStrLen = strlen(highStr);

    // Populate packet, and set pktlen
    packet[0] = (uint8_t)(seqNumber >> 8);
    packet[1] = (uint8_t)(seqNumber++);
    memcpy(packet + 2, highStr, highStrLen);
    RF_cmdPropTx.pktLen = 2 + highStrLen;

    // Send packet Tx
    RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropTx, RF_PriorityNormal, NULL, 0);

    // Toggle pin
    PIN_setOutputValue(ledPinHandle, Board_LED0, high);

    // Acknowledge the ALERT event
    scifAckAlertEvents();
}

```

Application Main Loop

- Build and debug project.

Test Proprietary RF application

To test the application, do the following:

- While the transmitter is running on one of the devices, open up SmartRF Studio. If both devices show up on the list of connected devices, disconnect the transmitter while setting up the receiver.
- Double click the available device, and choose `Proprietary Mode`.
- Choose the `50 kbps, 2-GFSK, 25 KHz deviation` setting at the top. This is usually the default.
- Go to the `Packet RX` tab.
- In the tab, check of `infinite packet count`, set viewing format to `Text`, and click `Start`.
- Try moving the wire or jumper on the transmitter. Packets with the dawn state should show up in SmartRF Studio.

You should see something like the picture below. Note the red markings.

Continuous TX Continuous RX Packet TX **Packet RX**

Expected Packet Count: 100 ☒ Infinite

Viewing Format: Text

Length Config: Variable

Sync Word: 0x930b51de Sync Word Length: 32 Bits

No address check ☐ 0xAA or ☐ 0xBB

☒ Seq. Number Included in Payload

10:20:36.077	0000	HIGH	-15
10:20:49.074	0001	LOW	-16
10:20:52.076	0002	HIGH	-18
10:21:00.074	0003	LOW	-19
10:21:05.072	0004	HIGH	-19
10:21:25.072	0005	LOW	-20
10:21:26.073	0006	HIGH	-19
10:21:29.072	0007	LOW	-20
10:21:30.071	0008	HIGH	-27
10:21:32.069	0009	LOW	-21
10:21:36.070	0010	HIGH	-21



(<http://creativecommons.org/licenses/by-nc-nd/4.0/>)

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).