RA0L1 Group

FPB-RA0L1 Tutorial

Introduction

This application note explains how to create a new project, setting touch sensors and debug a program using the e^2 studio integrated development environment for the FPB-RA0L1 board with Renesas' RA0L1 group MCU.

Target Device

RA0L1 Group

Related Documents

[1] RA0L1 Group User's Manual: Hardware (R01UH1143)
[2] RA0L1 Group Fast Prototyping Board for RA0L1 Microcontroller Group FPB-RA0L1 v1 User's Manual (R20UT5601)
[3] Renesas e^2 studio 2023 -10 or Higher Quick Start Guide User's Manual RA Family Renesas MCU (R20UT5210)

Contents

1. Development environment

This application note explains using the following development environment.

1.1 Hardware environment

Use the following hardware:

- board : FPB-RA0L1(Product part number: RTK7FPA0L1S000010BJ, Installed Device: R7FA0L1074CFL) Connect the board and PC using the USB Type A to Type C cable included with the product.

1.2 Software environment

This document uses the following software. Please install the software in advance.

- Integrated development environment
  — e^2 studio : V2025-07 or later
- C compiler
  — GNU ARM Embedded : 13.3.1.arm-13.24 or later
- Flexible Software Package
  — Version : 6.1.0 or later
- QE for Capacitive Touch: Development Assistance Tool for Capacitive Touch Sensors
  — Version : 4.2.0 or later

2. Software overview

This section explains the specifications of the program created using this application note.

2.1 Program

Using the touch sensor function of the RA0L1 Group, create a program that performs the following controls.

- Control the lighting status of LED5 and LED6 according to the touch detection status of the Touch Button
- Touch detection on Touch Button1 controls the start/stop of blinking of LED1 and LED2.
- Touch detection on Touch Button2 change the blinking cycle the LEDs.

Indicates the board used and the location of the LEDs and touch sensors.

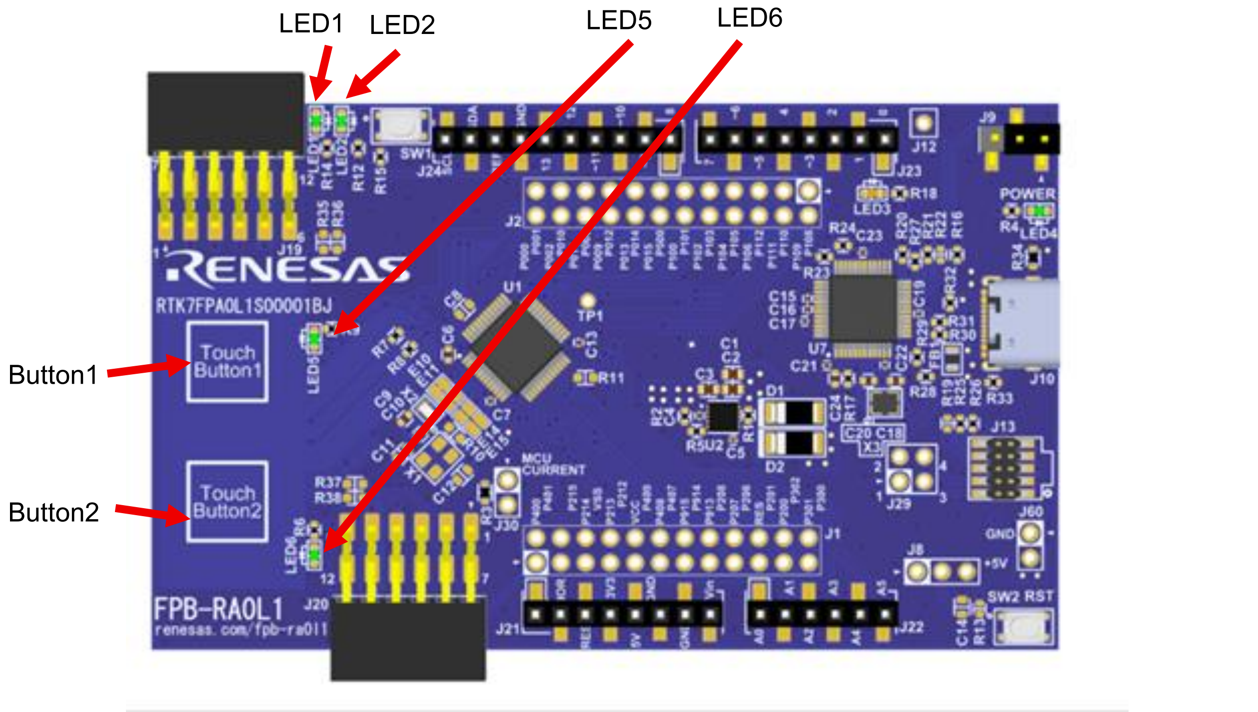


Figure 2-1. Position of LED on board

2.2 Resources

This section describes the resources consumed by the program described in this application note.

2.2.1 Clock Generation Circuit

HOCO clock frequency: 32MHz

HOCO clock division: 1 division

ICLK selection source: HOCO

ICLK frequency: 32MHz

TAU CK00 supply clock:32MHz

2.2.2 Low Voltage Detection (LVD)

Low voltage detection circuit used: Voltage monitoring 0

Detection voltage level: 1.86V

2.2.3 Capacitive Touch Sensing Unit (CTSU2SIa)

Touch sensor Functions used： TouchButton1， TouchButton2

FSP software stack used： rm_touch， r_ctsu

2.2.4 Timer Array Unit (TAU)

Timer function used： TAU channel 0

TAU clock supplied to channel 0： CK00

Channel 0 operating mode： Interval timer mode

Interval timer period： 1ms

FSP software stack used： r_tau

2.2.5 I/O Ports

Ports used： P002(LED1) / P104(LED2) / P401(LED5) / P400(LED6)

FSP software stack used： r_ioport

3. Generate new project

This section explains how to generate project.

3.1 Project launch

1. From the e² studio menu bar
   Select "New" → "Renesas C/C++ Project" → "Renesas RA".

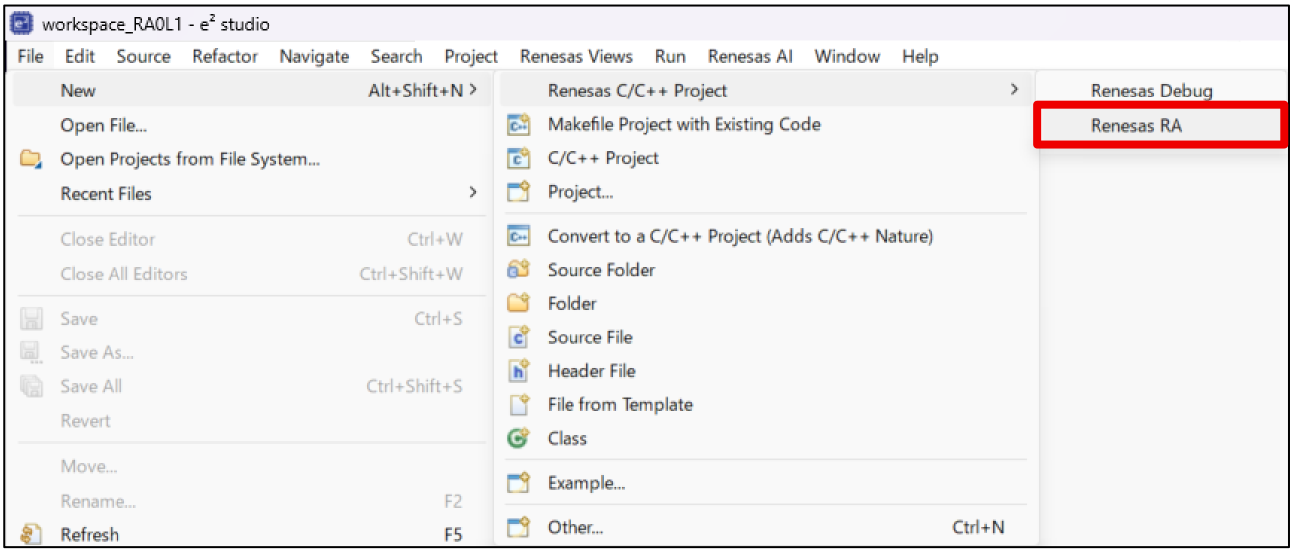


Figure 3-1. Project launch

2. Select "Renesas RA C/C++ Project", then click "Next".

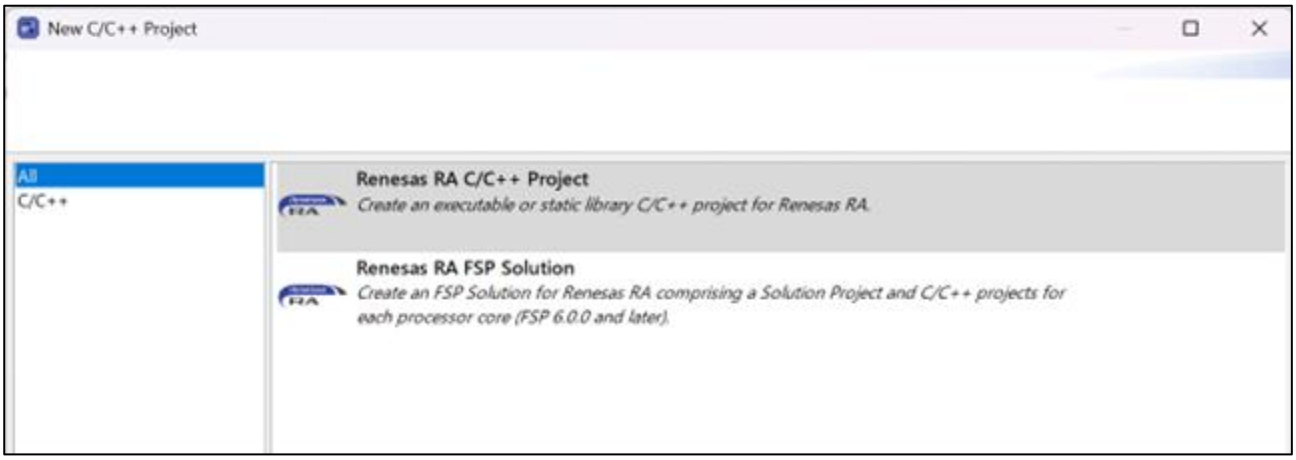


Figure 3-2. Project launch

3.2 Setting the project name

Set any name to the project name, click “next”. (In this application note, the name is FPB_RA0L1_Tutorial)

When “Using default location” is checked, the project will be created in the path shown. If you want to create it in a different location, uncheck it and set the path.

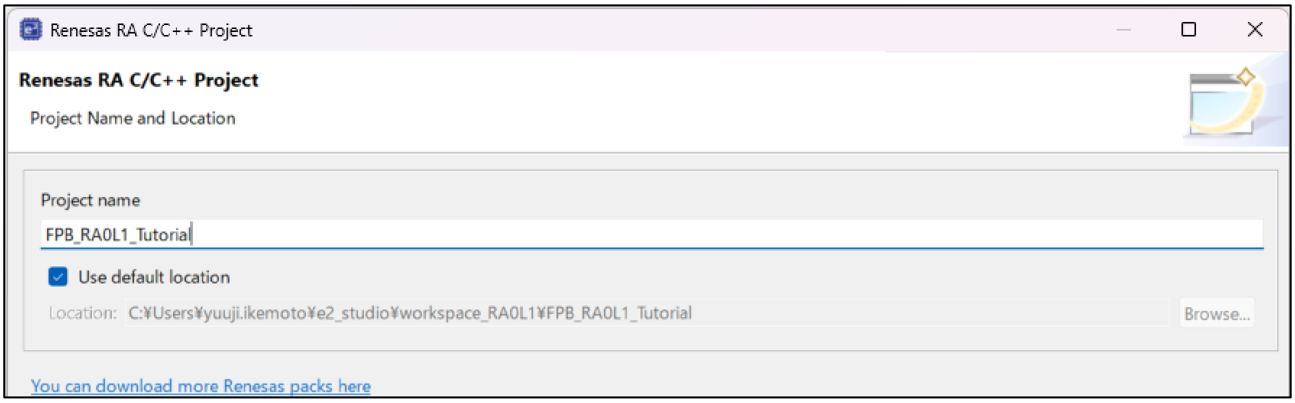


Figure 3-3. Setting the project name

3.3 Device/Tool Configuration

The following steps outline the configuration of the projects target device and tooling.

1. FSP Version: Select the latest version
2. Board: Select “RA0” → “RA0L1” → “FPB-RA0L1”
3. Device: Device is set automatically by selecting Board
4. Language: Select C

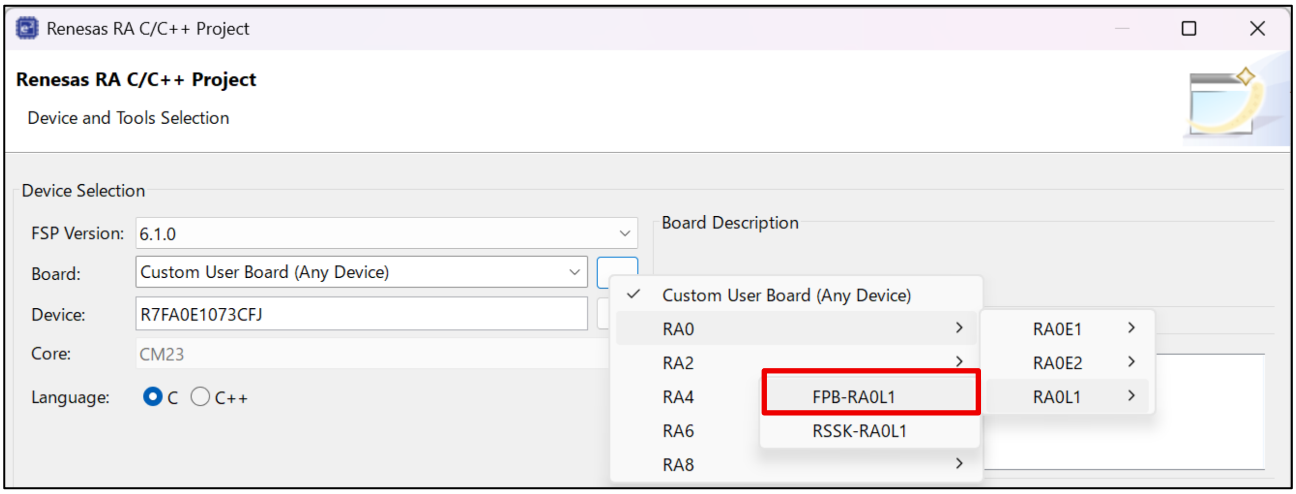


Figure 3-4. Device/Tool Configuration

5. Toolchains settings: Select “GNU ARM Embedded”. Select the latest version from the list of version.

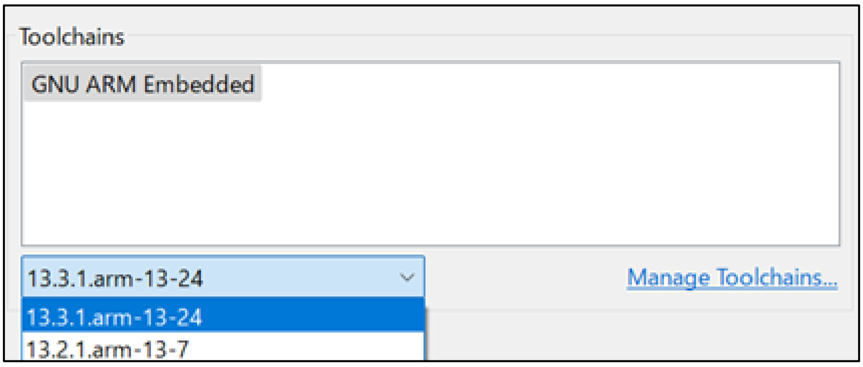


Figure3-5. Device/Tool Configuration

6. Debugger settings: Select J-Link ARM

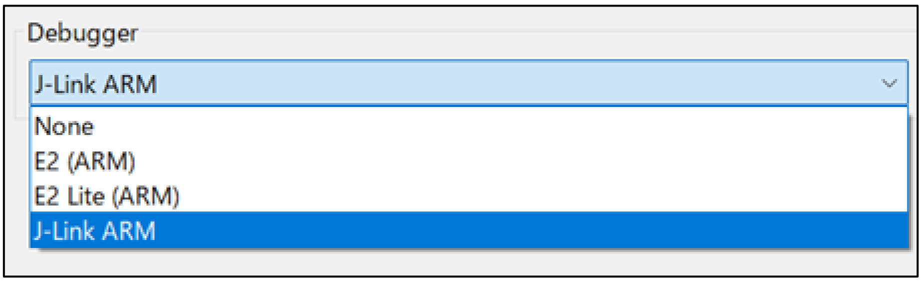


Figure3-6. Device/Tool Configuration

The settings are complete.

7. Confirm the setting in the red frame are and click “Next”.

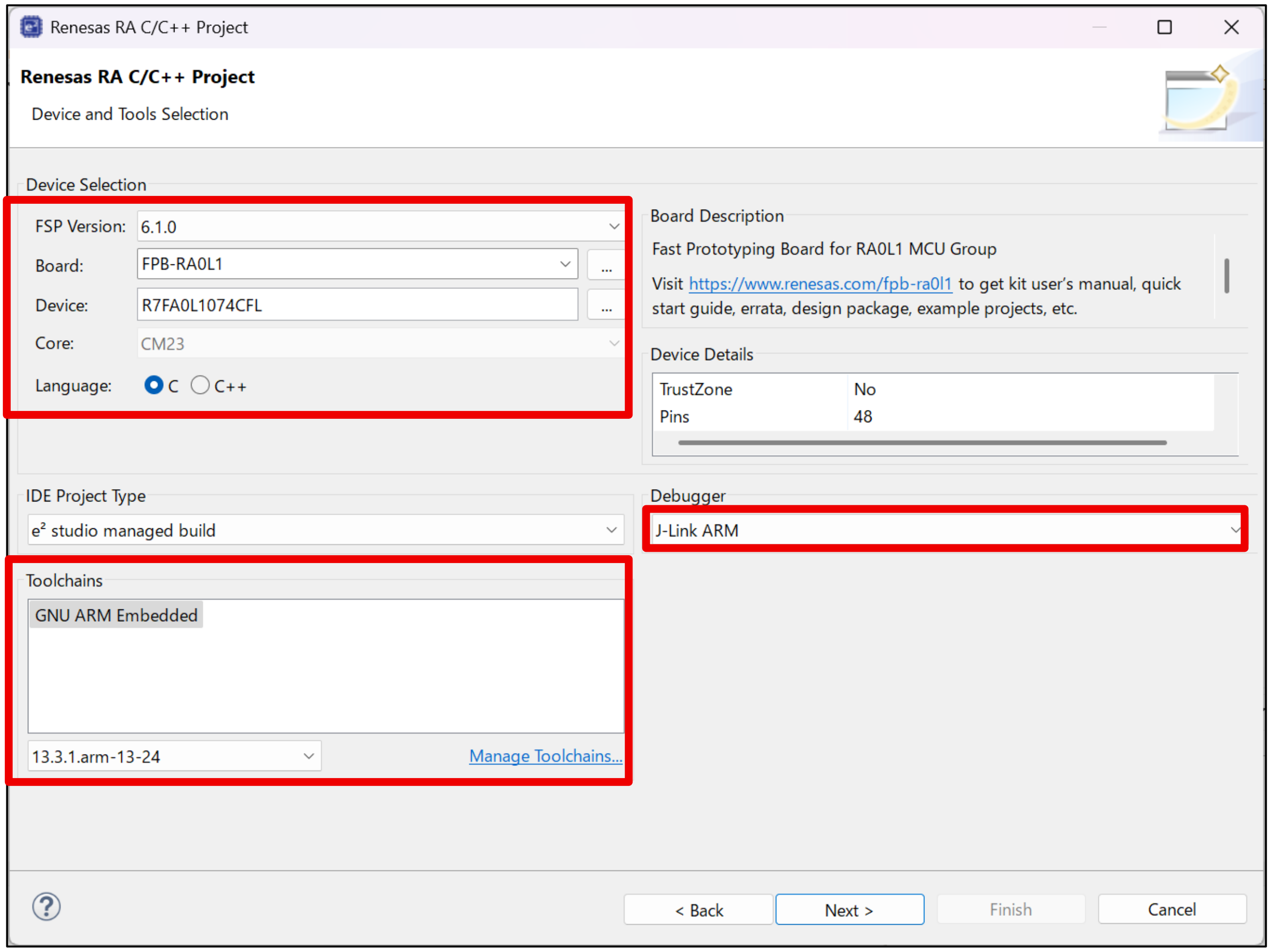


Figure3-7. Device/Tool Configuration

3.4 Setting Preceding Project or Smart Bundle Selection

Confirm "None" has selected and click "Next".

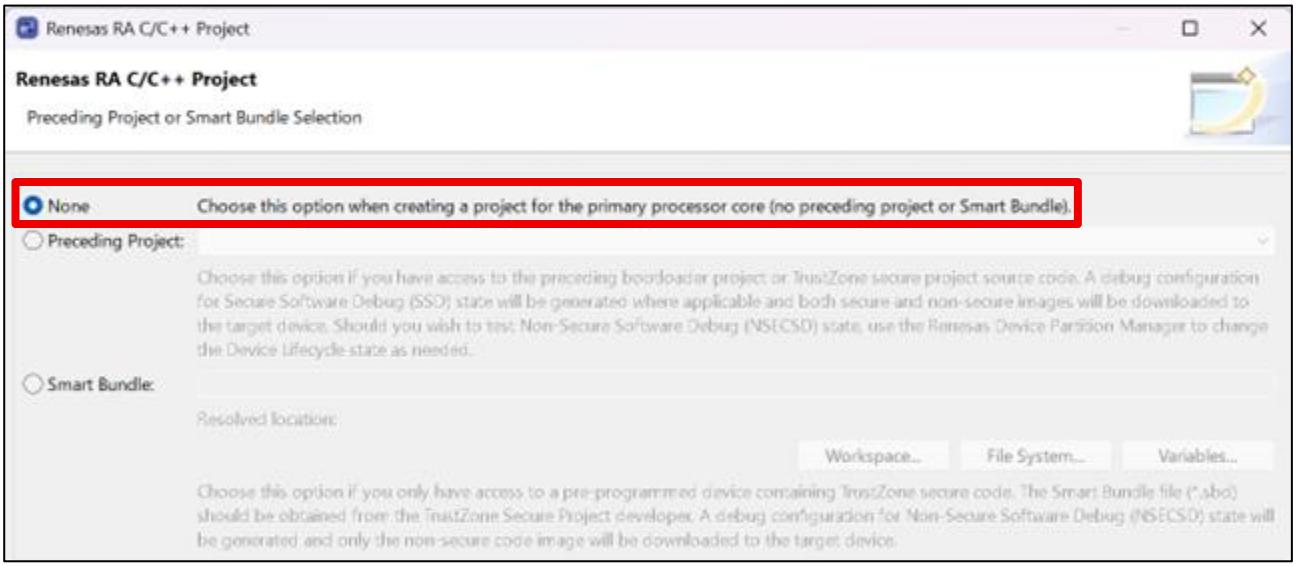


Figure3-8. Setting Preceding Project or Smart Bundle Selection

3.5 Build artifact settings

In this application note, we will generate an executable file from the program, so select "Executable".

Select "No RTOS" since "RTOS" is not used. After selecting both options, click "Next".

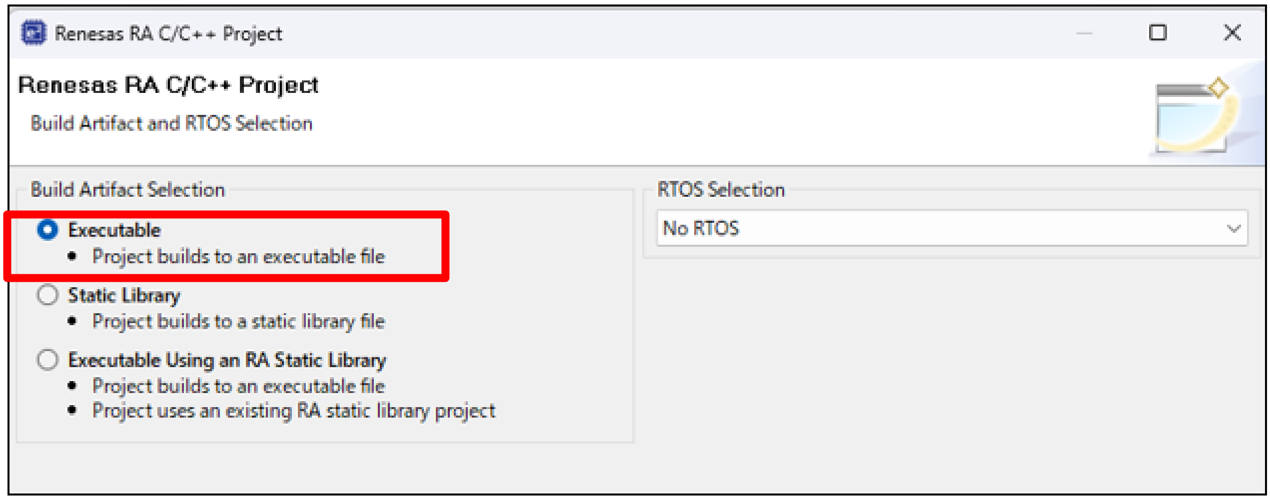


Figure3-9. Build artifact settings

3.6 Template type settings

1. In this application note, we will explain how to use FSP, so this time we will select Bare Metal - Minimal. Then click "Finish".

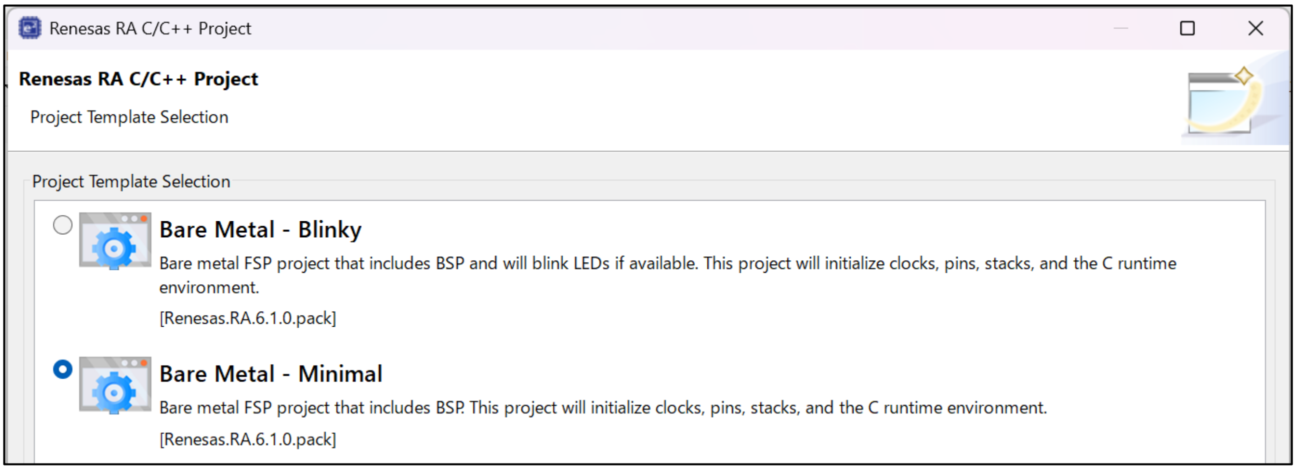


Figure3-10. Template type settings

2. Click “Open Perspective” when the "FSP Configuration" Perspective which optimizes the FSP Configuration workflow is pop-up.
   * If you click no, it can be accessed again by selecting "Open New Perspective" ... then images/instructions of this process are provided.

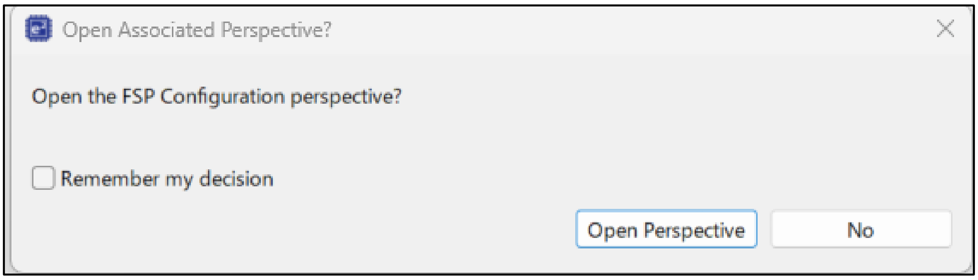


Figure3-11. Setting template type

3. Project creation is complete.

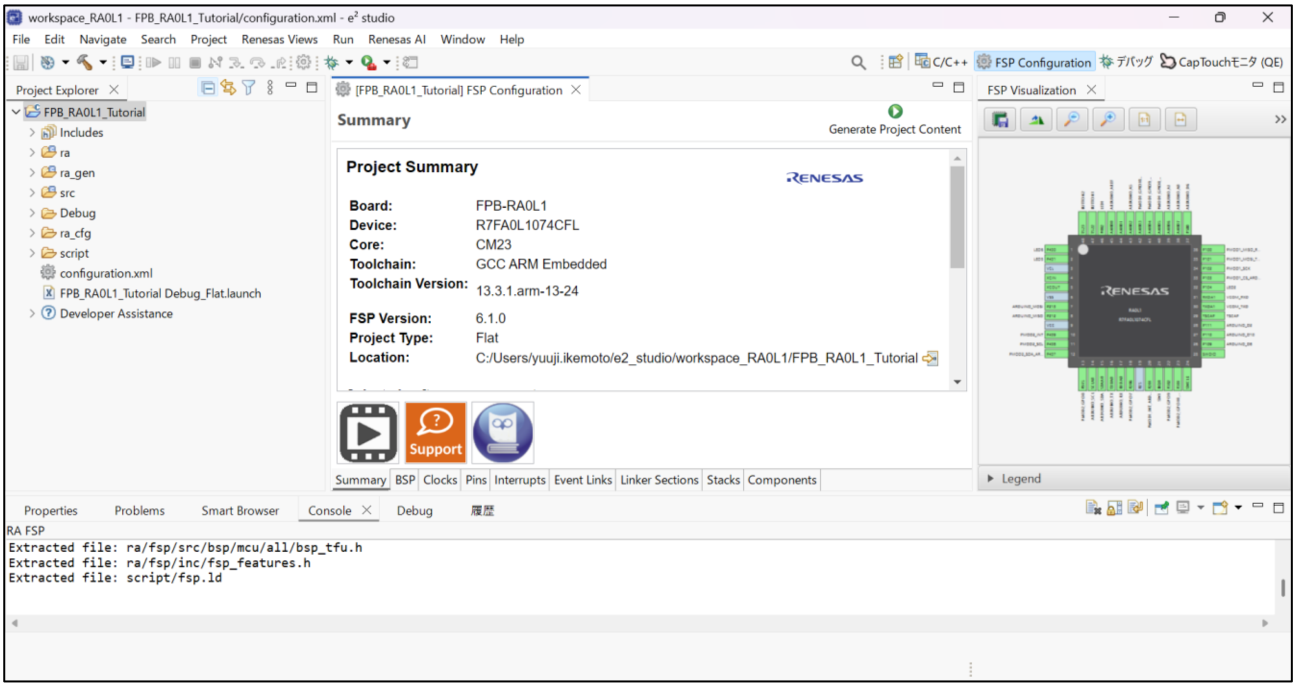


Figure3-12. Template type settings

4. Tuning and monitoring touch sensors using QE Touch

4.1 Setting FSP Configurator of touch IF

Use FSP Configurator to configure the system clock and initial settings for peripheral functions.

4.1.1 How to open FSP Configurator window

Double-click the configuration.xml file in the Project Explorer to open the FSP Configurator window.

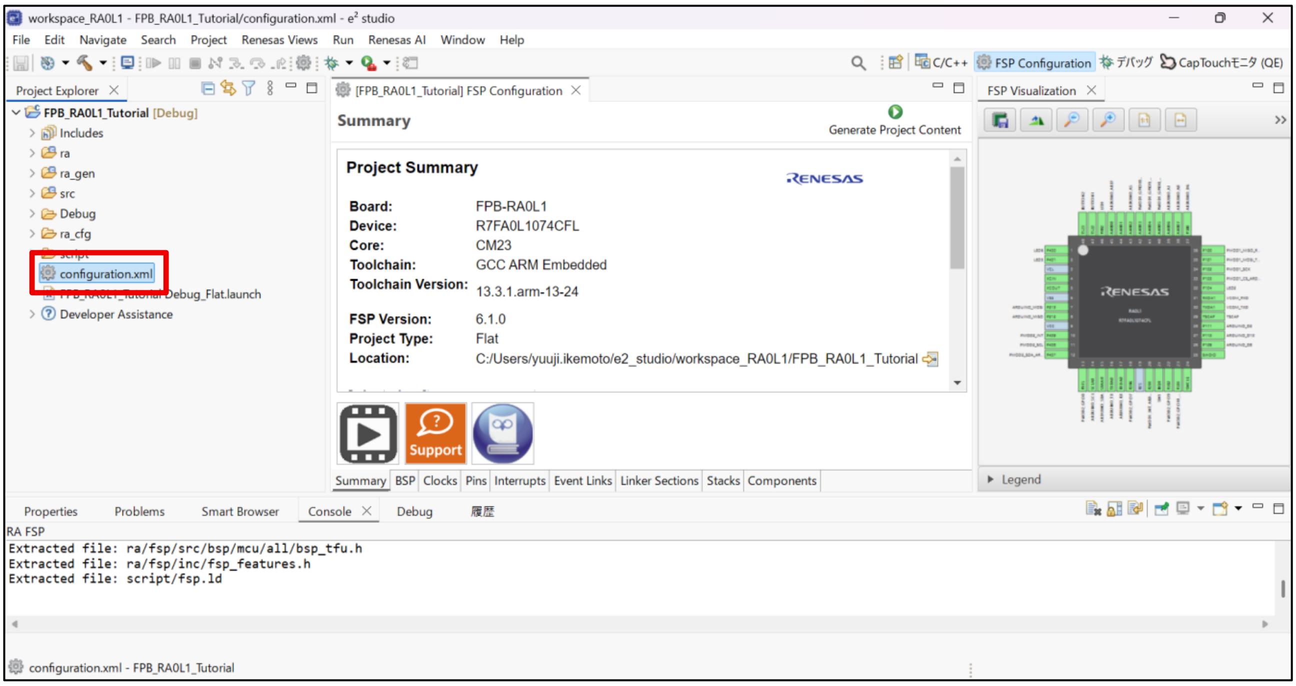


Figure4-1. How to open FSP Configurator window

4.1.2 Clock configuration

Select the “Clocks” tab to open clock setting window. Verify that the default screen looks like the one below. In this Application note, clock frequency is 32 MHz, so the setting is not needed to change. Verify that the clock supplied to TAU CK00 is 32 MHz.

The path of each clock is indicated by a thick arrow.

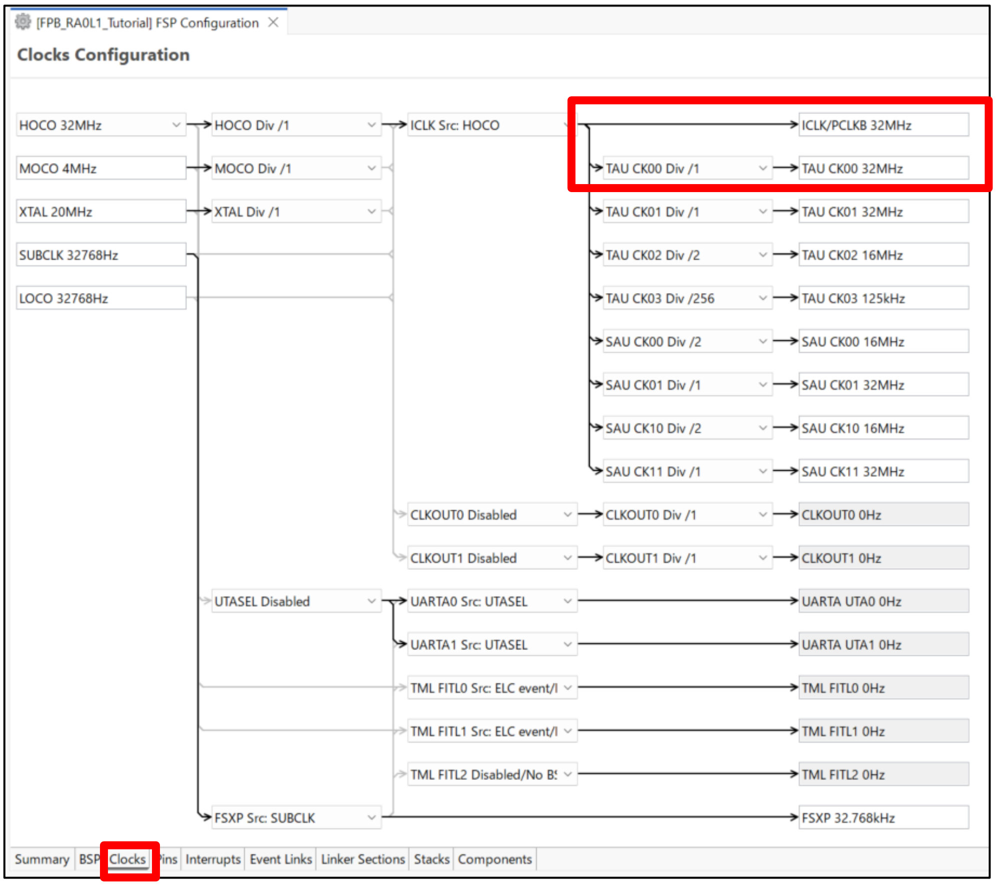


Figure4-2. Clock configuration window

4.1.3 Low Voltage Detection configuration

1. Select the “BSP” tab to open the Board Support Package Configuration window. Then select the “Properties” tab in the lower panel to display the Properties window.

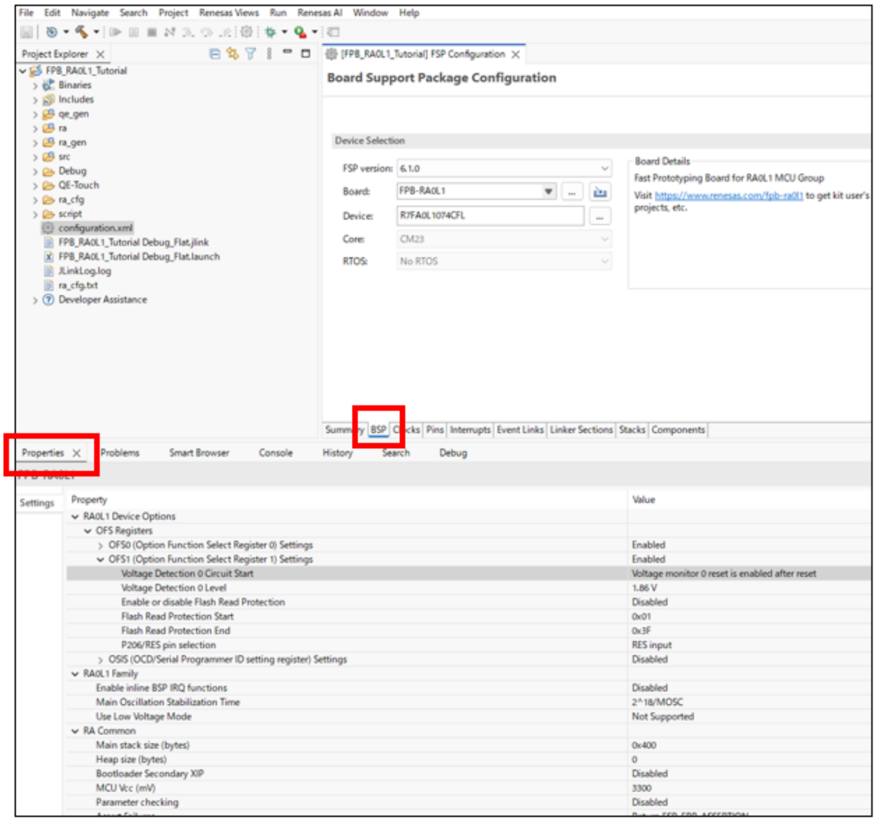


Figure4-3. Displaying the “Properties” window

2. In the Properties window, set “Voltage Detection 0 Circuit Start” to “Voltage monitor 0 reset is enabled after reset” and “Voltage Detection 0 Level” to “1.86V”.

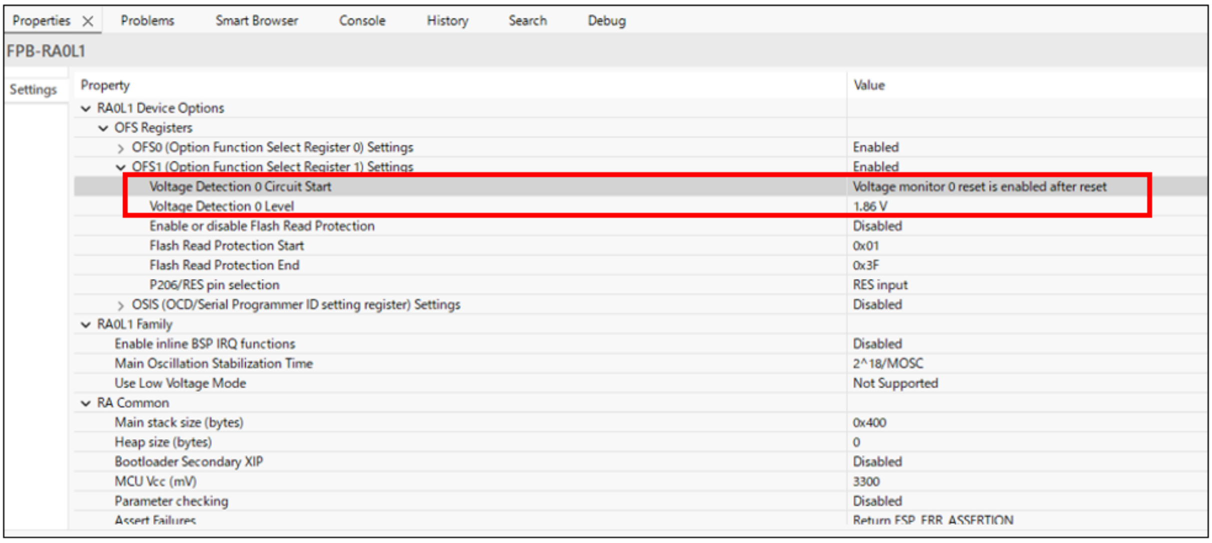


Figure4-4. Low Voltage Detection Circuit Settings window

4.1.4 Touch driver configuration

1. In the FSP Configurator window, open the “Stack” tab, add “New Stack” → “CapTouch” → “Touch (rm_touch)”

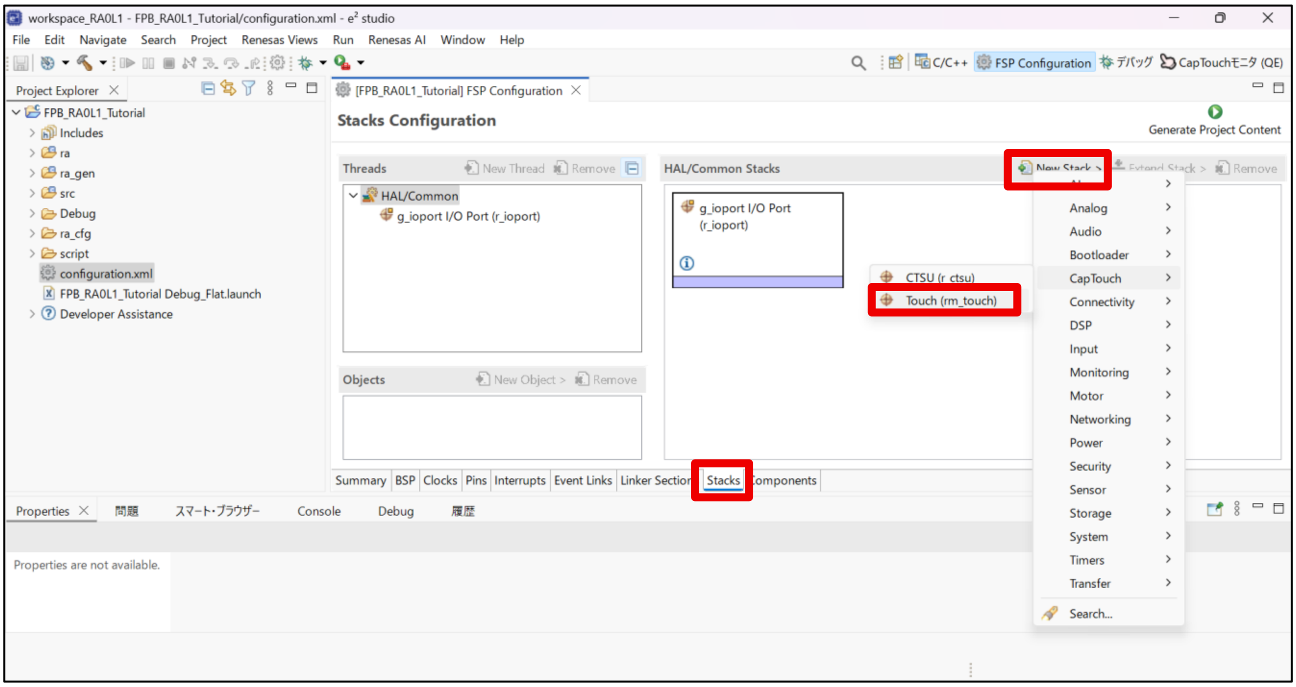


Figure4-5. Touch driver configuration

2. After confirming that the stack has been added, the Touch driver configuration is complete.

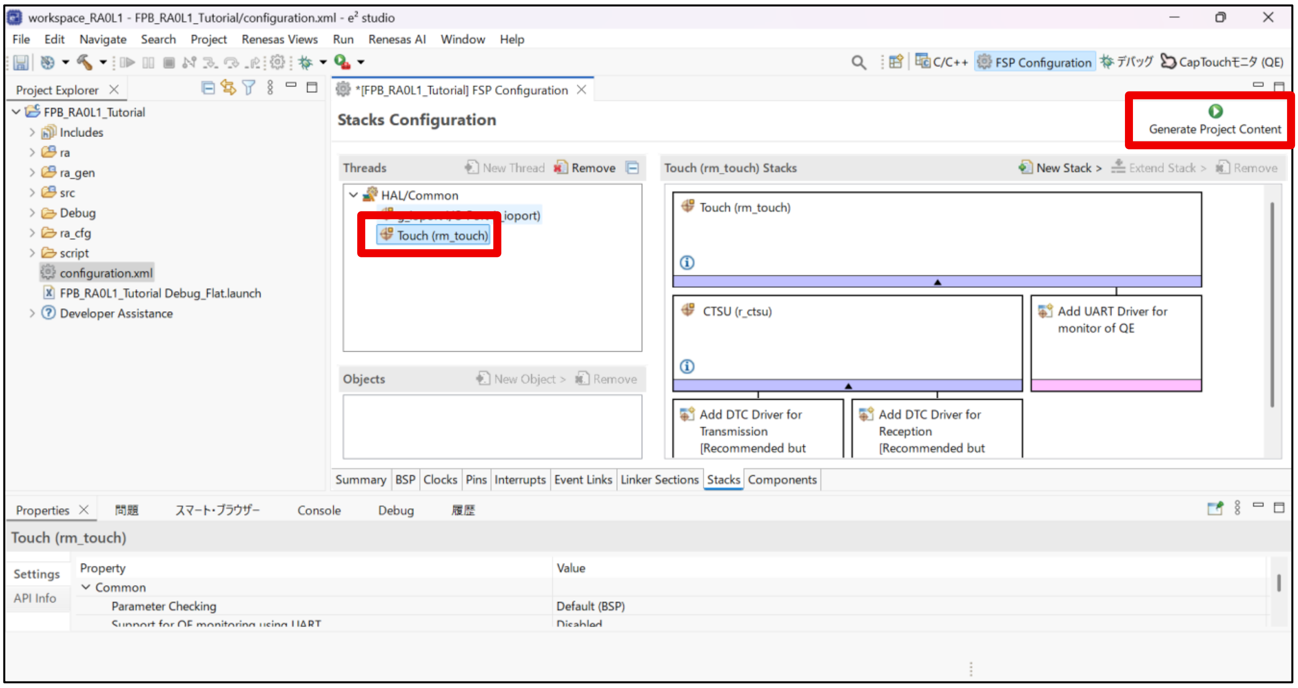


Figure4-6. Touch driver configuration

4.1.5 Confirm Pin configuration

1. In the FSP Configurator window, open “Pins” tab, and open “Pin Selection” → “Peripherals” → “HMI:CTSU” → ”CTSU”.
   Confirm that “TS22”, “TS23” and “TSCAP” are checked(*), then click “Generate Project Content” to generate code.
   * These are automatically set by selecting board at “3.3 Device/Tool Configuration”.

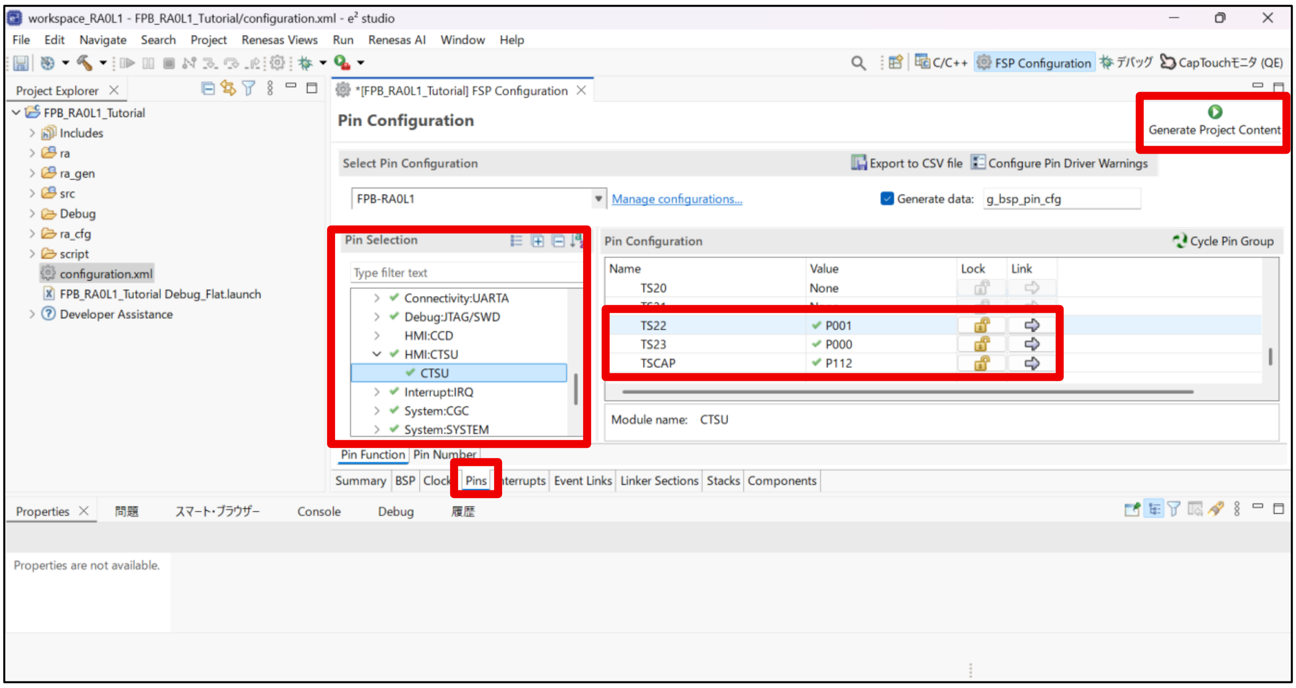


Figure4-7. Confirm Pin configuration

A pop-up will appear asking you to save your settings before code generation. Please click “Proceed”.

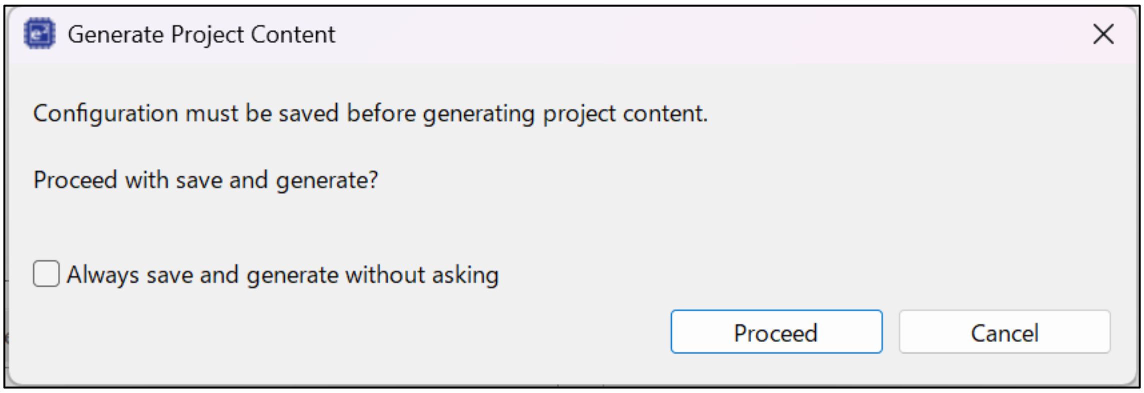


Figure4-8. Confirm pin configuration

4.2 Tuning touch sensor by using QE Touch

4.2.1 Preparing for Capacitive Touch

(1) Select a project

From the e² studio menu bar click “Renesas Views” → “Renesas QE” → “CapTouch workflow”.

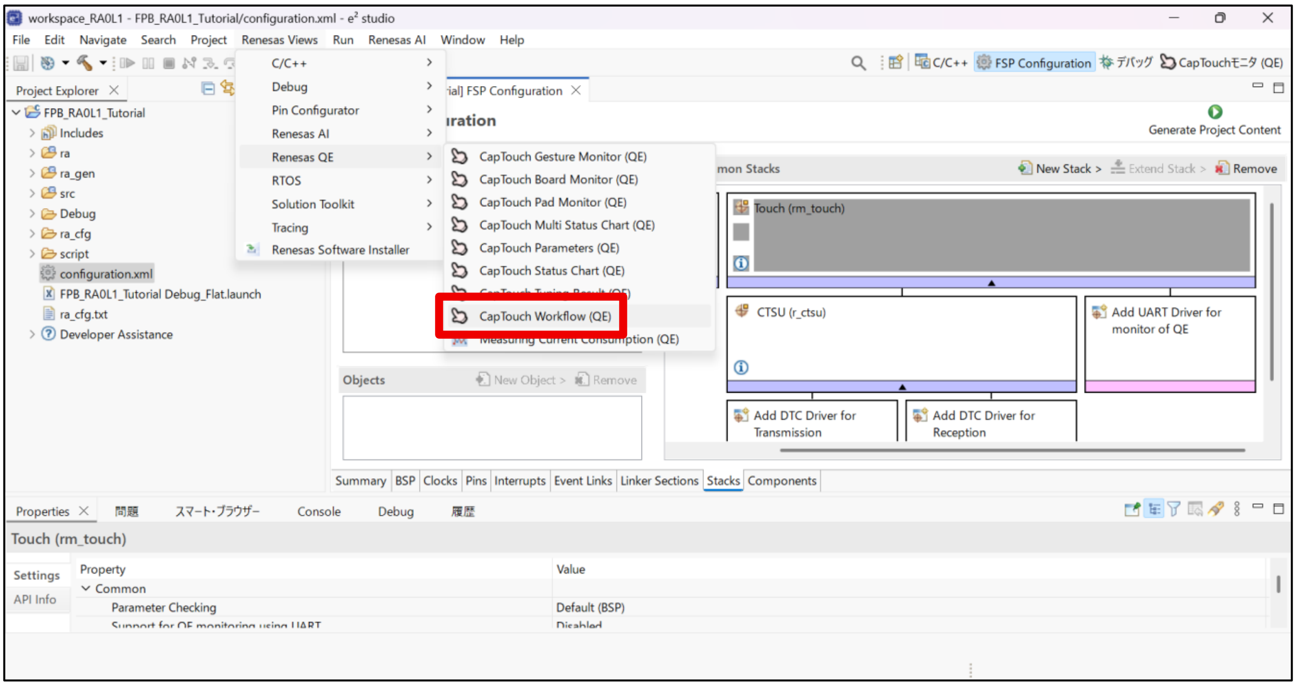


Figure4-9. Selecting the Touch workflow

Select the target project from “Select a Project”.

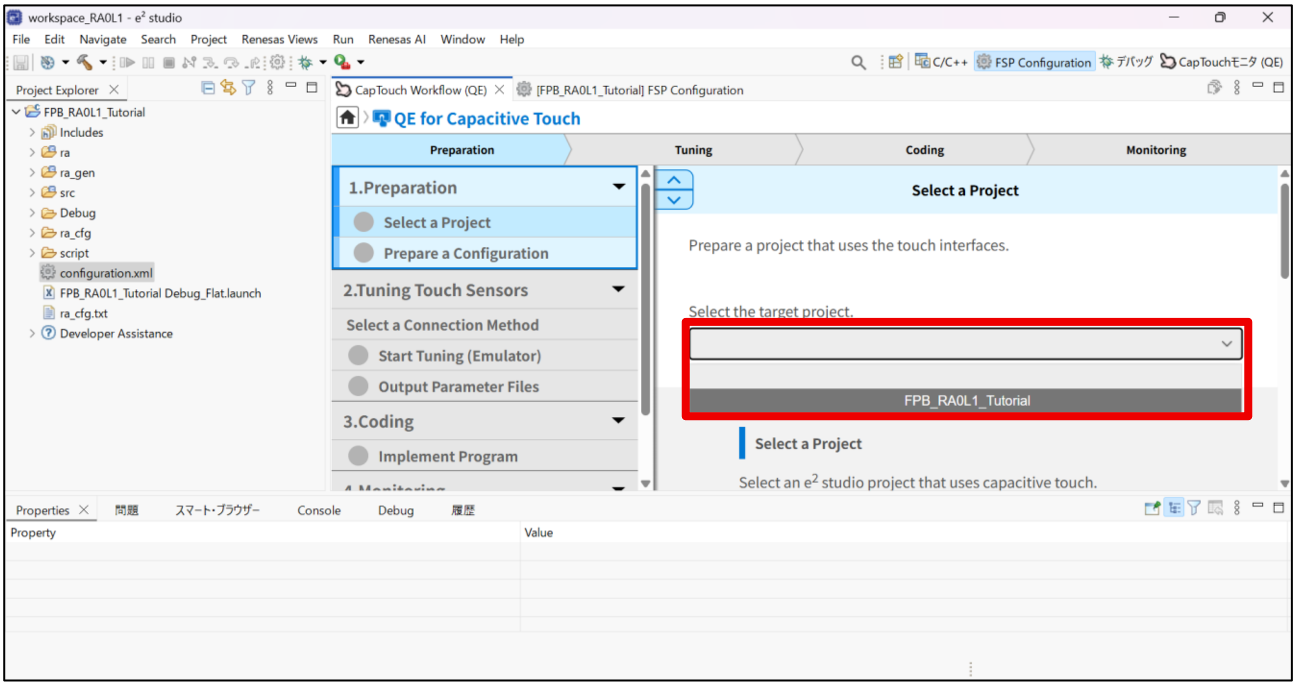


Figure4-10. Selecting a project

(2) Prepare a configuration

Click “Create a new configuration” in “Prepare a Configuration”.

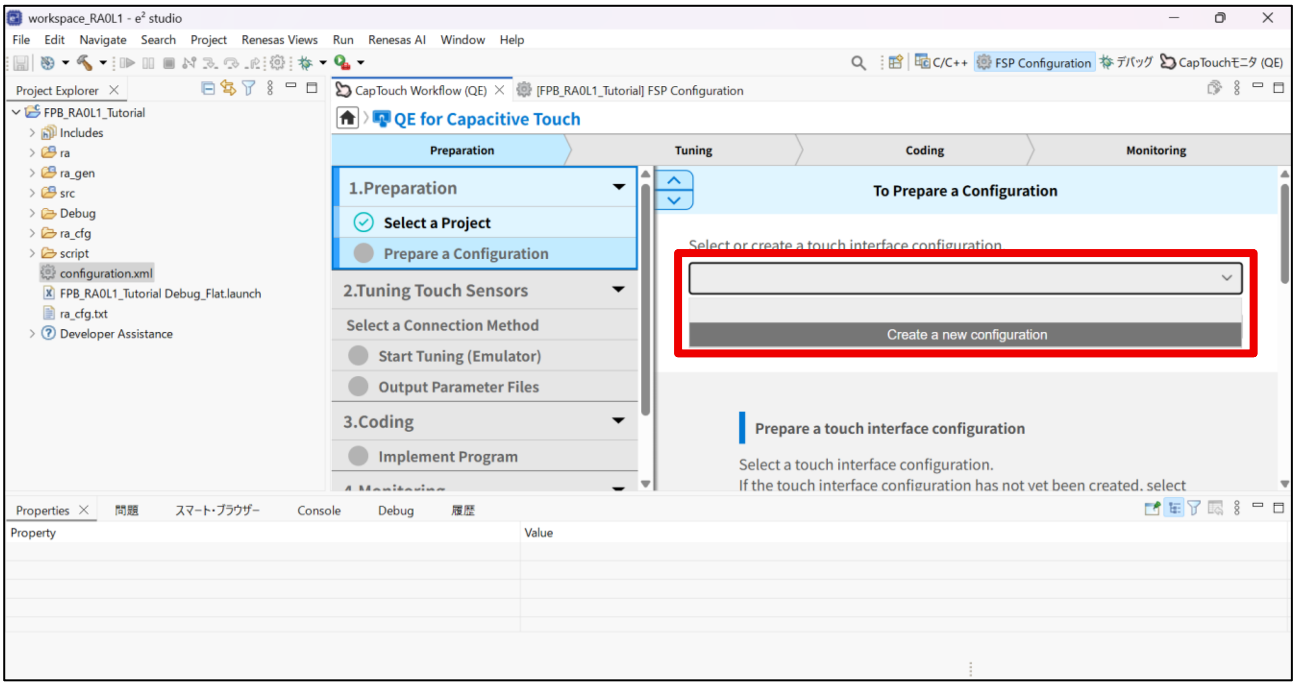


Figure4-11. Preparing a configuration

The "Create Configuration of Touch Interface" window will open. Select interface to use from Touch I/F list on the right and configure the setup by clicking within the fields in the red frame.

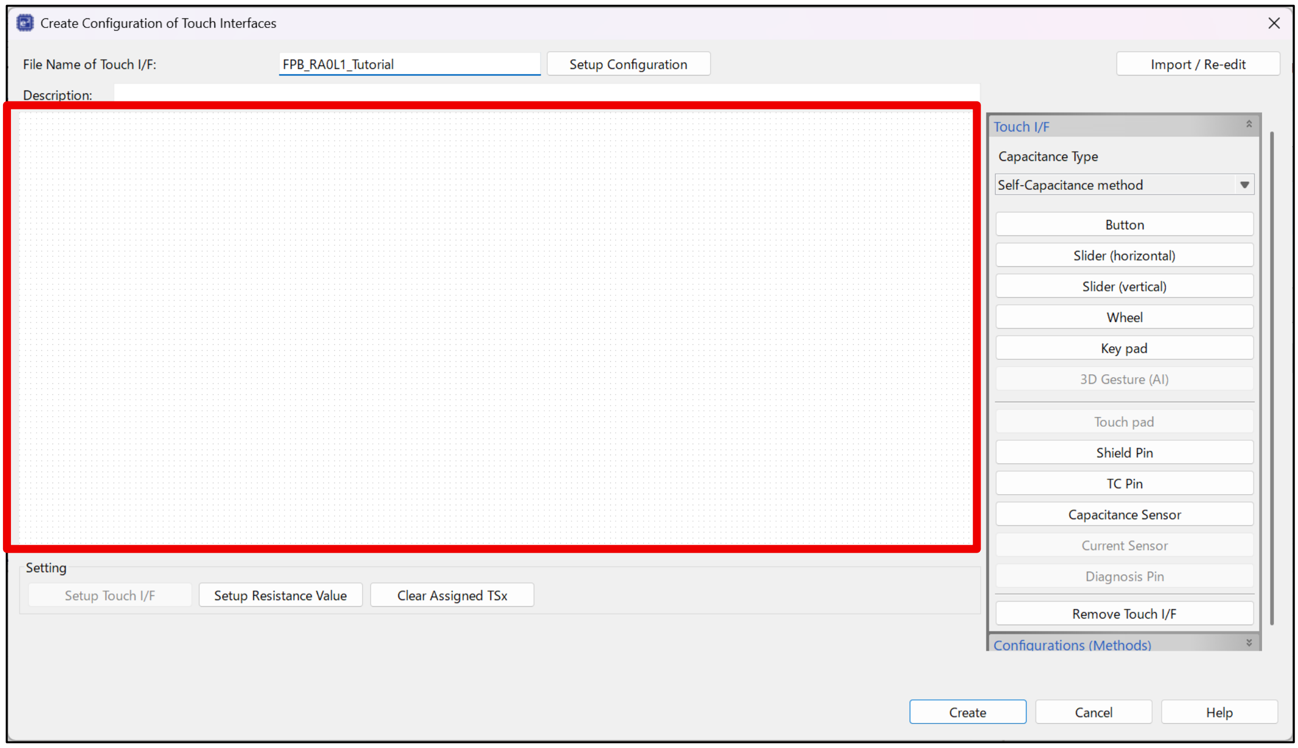


Figure4-12. Creating a touch interface configuration

This project uses 2 buttons. Select the buttons from the Touch I/F list on the right and place two of them on the field.

After placing the two buttons, select “Button” in the touch interface again or press the Esc key to deselect it.

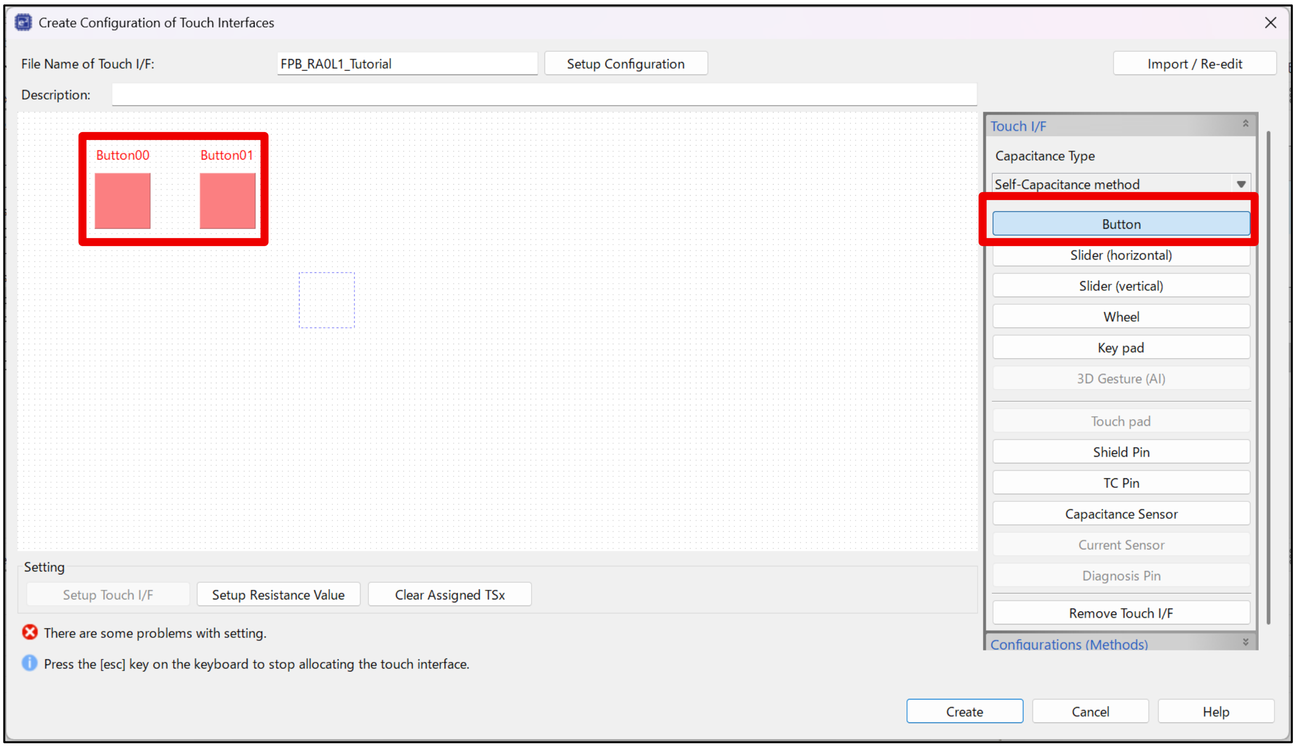


Figure4-13. Creating a touch interface configuration

When the placed button is bubble clicked, [Setup Touch Interface] pup-up window will appear.

Configure the first touch interface as follows.

- Name: Button1
  *The names specified here will be used as part of the macro names in the subsequent sample program.
- Touch sensor: TS22

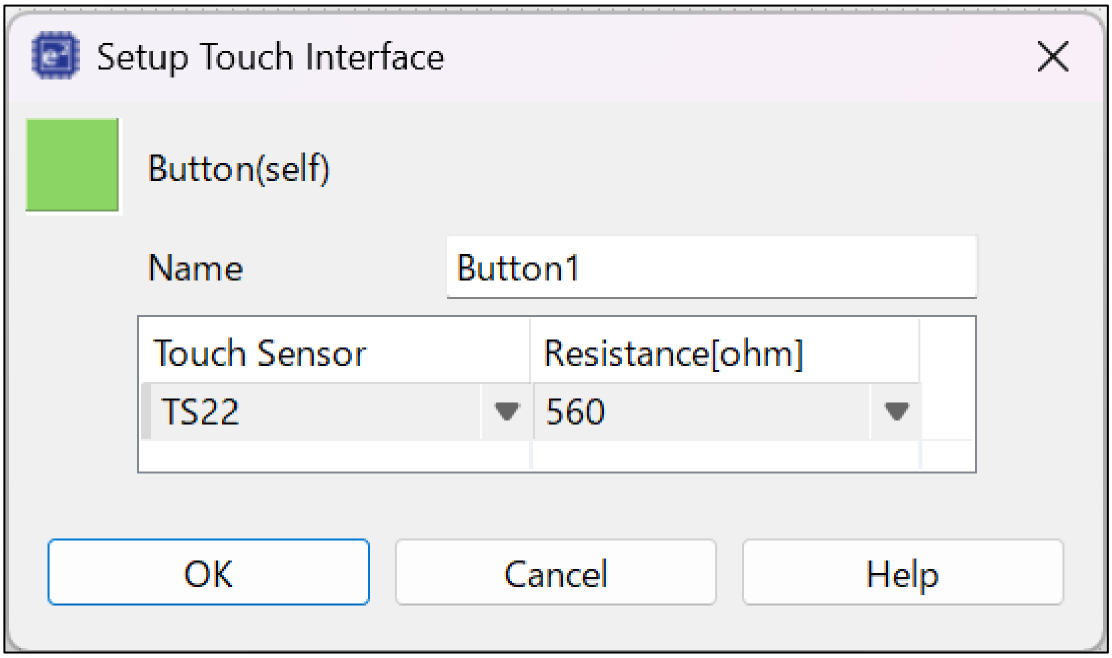


Figure4-14. Setting up the touch interface

Configure the second touch interface as follows.

- Name: Button2
- Touch sensor: TS23

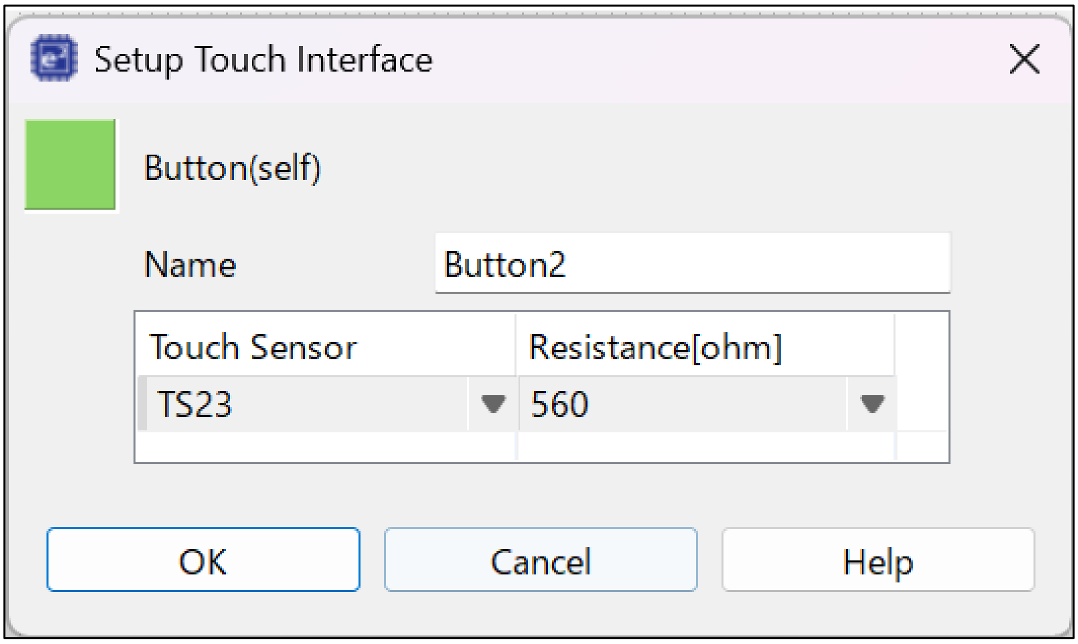


Figure4-15. Setting up the touch interface

The TS pins to use can be identified by referring to the board manual or circuit diagram.

Please refer to “Figure 2-1. Position of LED on board” for the position of the buttons.

Table 4-1 FPB-RA0L1 Capacitive touch button control port

Button number	RA MCU PORT
Touch Button1	P001/TS22
Touch Button2	P000/TS23

When touch buttons setting has been done, the buttons indicate green. If no error message is displayed in the bottom left, the setup is complete. Then click “Create” to finish the configuration of touch interface.

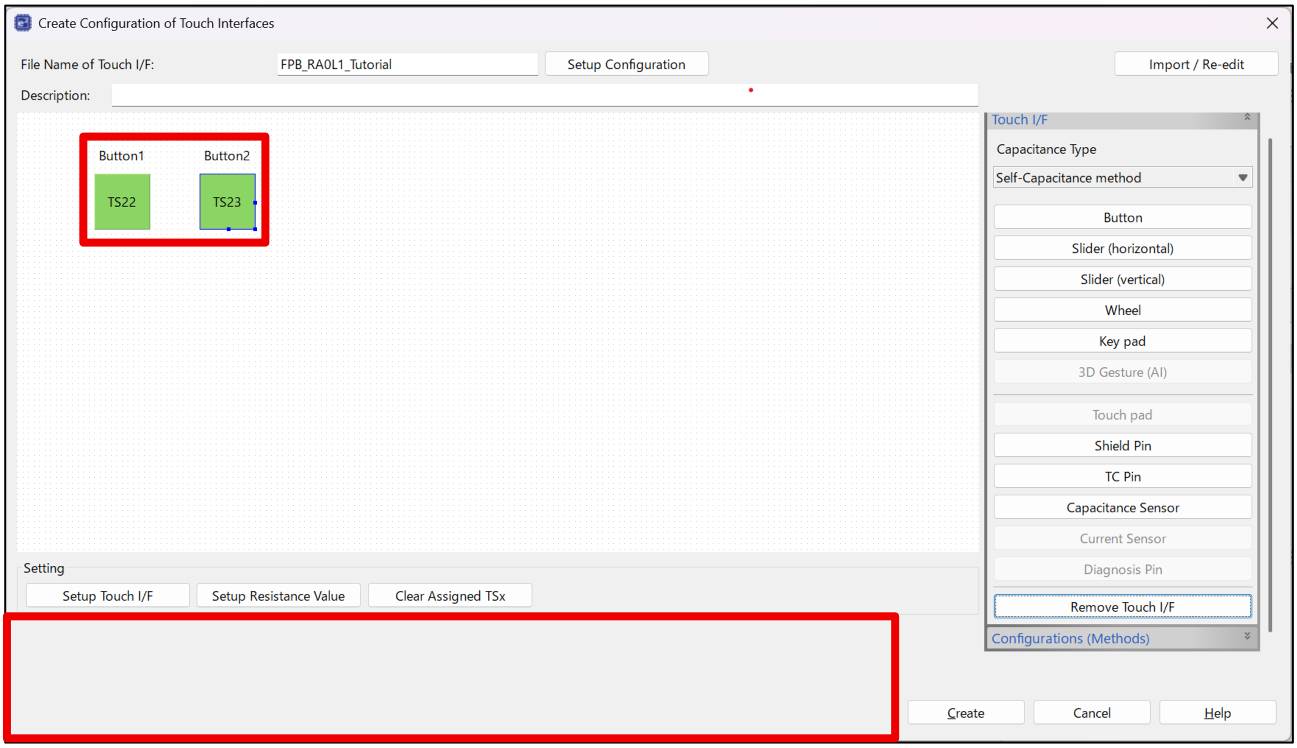


Figure4-16. Touch interface configuration

4.2.2 Tuning touch sensor

(1) Tuning (connect emulator)

Connect the board and PC using the USB cable included with the product. Then click “Start Tuning”.

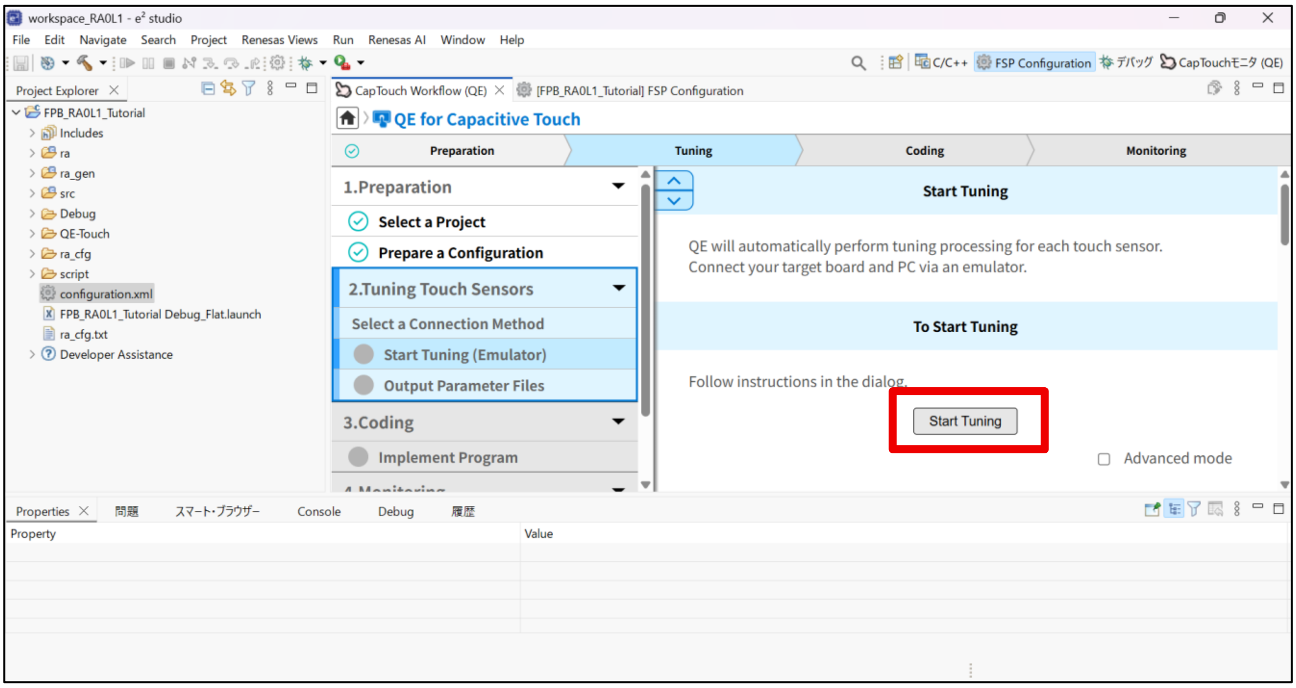


Figure4-17. Tuning (connect emulator)

Set the peripheral module clock frequency to 32MHz and click “OK”.

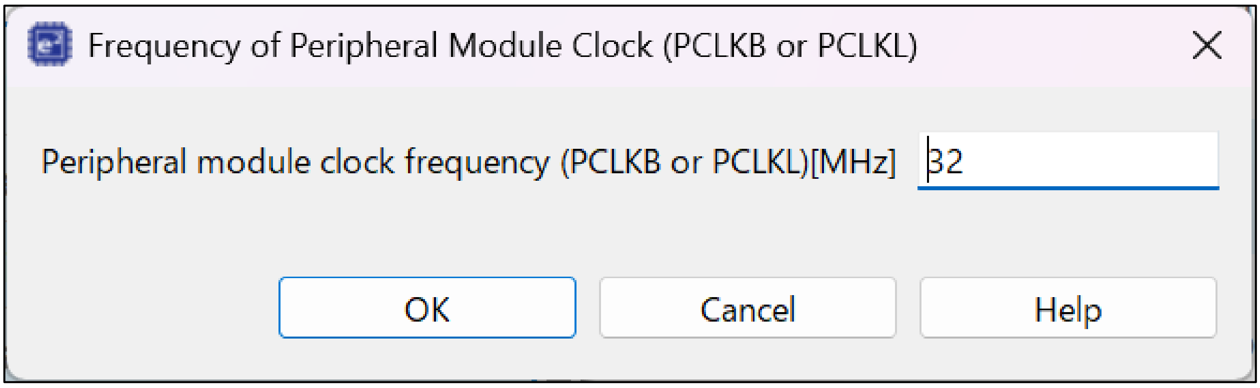


Figure4-18. Tuning (connect emulator)

In the “Select resource to save” window, check the configuration.xml for this project and click “OK”.

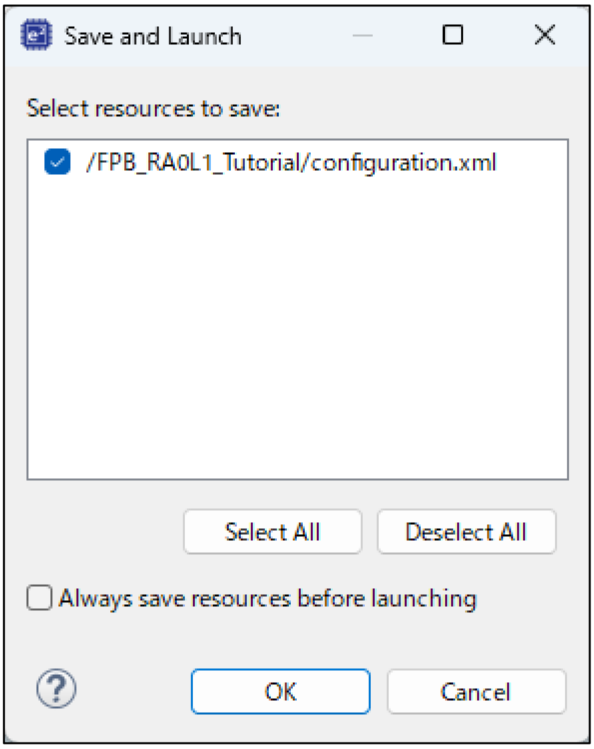


Figure4-19. Tuning (connect emulator)

Click “Switch” when “Confirm Perspective Switch” pop-up window is displayed.
* Clicking "No" is also acceptable.

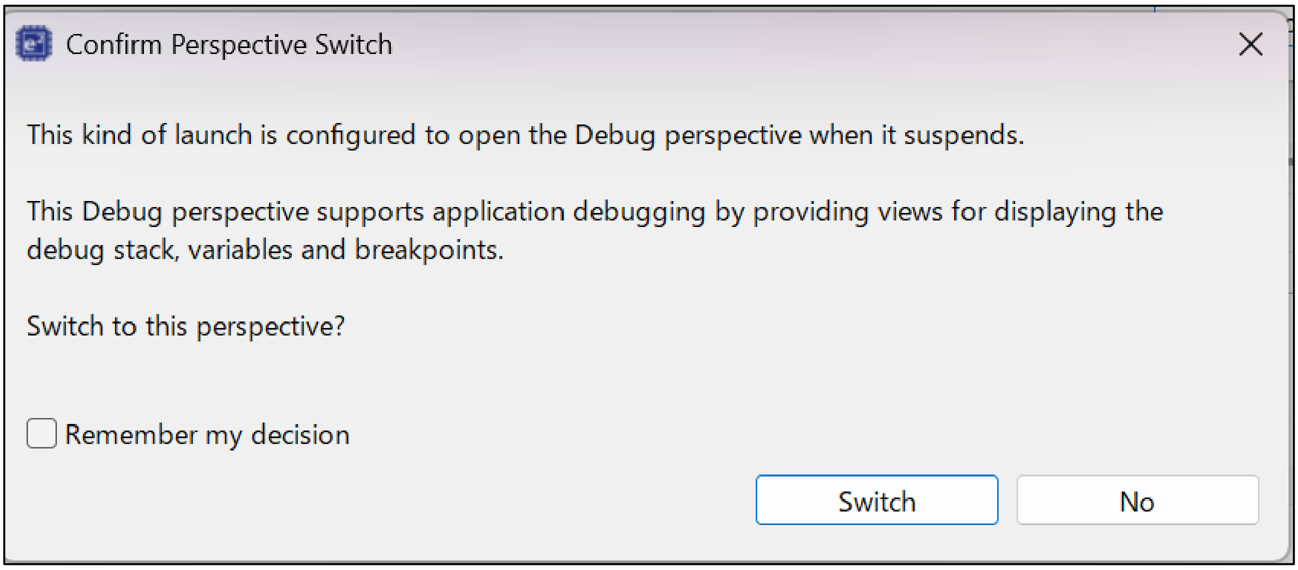


Figure4-20. Tuning (connect emulator)

A pop-up window will appear to tune the touch sensor sensitivity.

This pop-up is for tuning sensitivity of Button1. Touch button1 on the board.

Press any key on the PC keyboard to accept the sensitivity measurement. This completes the sensitivity tuning for Button1.

[Not touch Button1]

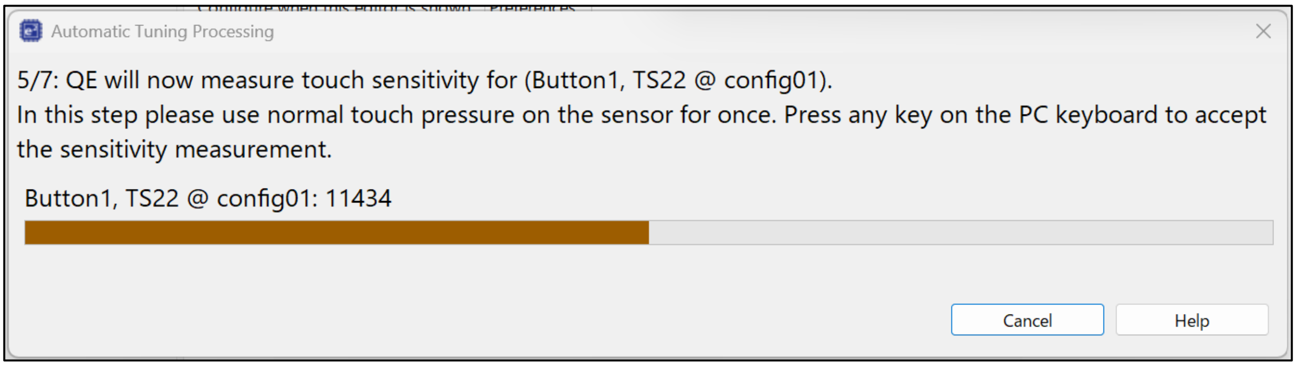


Figure4-21. Tuning (connect emulator)

[Touch Button1]

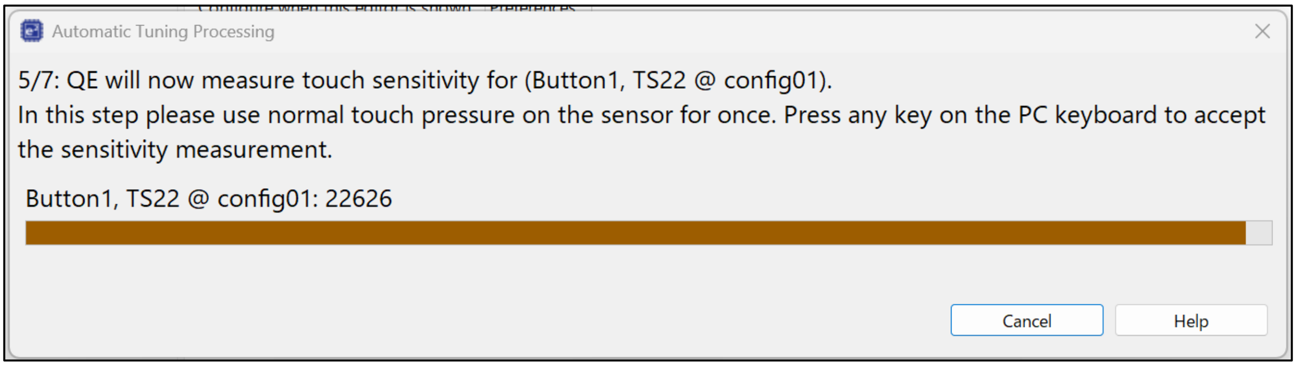


Figure4-22. Tuning (connect emulator)

Next, a pop-up window will appear to tune the sensitivity of Button2. Tune sensitivity in the same way as for Button1.

[Not touch Button2]

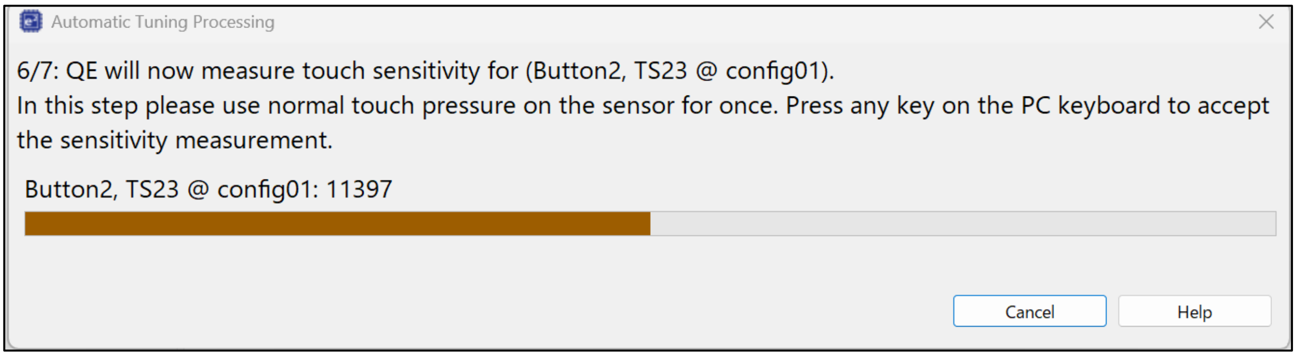


Figure4-23. Tuning (connect emulator)

[Touch Button2]

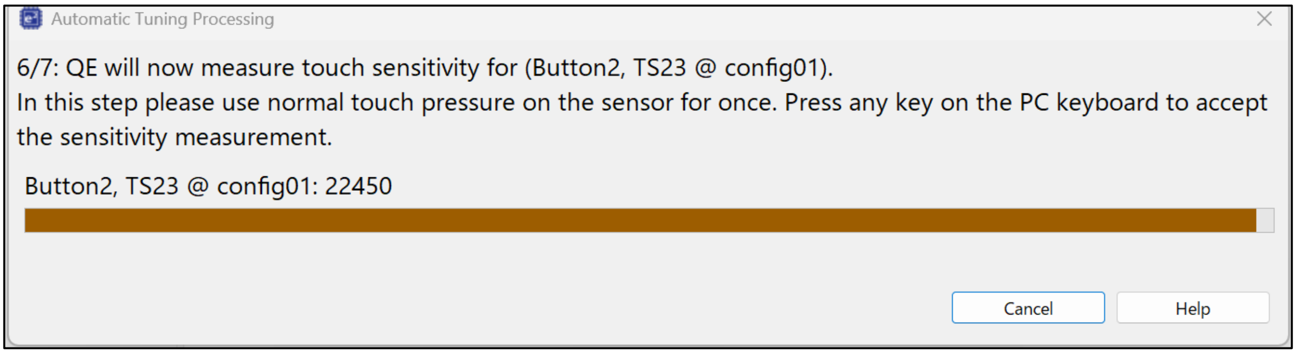


Figure4-24. Tuning (connect emulator)

When sensitivity tuning is completed, a pop-up window displaying the tuning results will appear.

Check for “Overflow” and “Warning / Error”. If there is any message, check “Select the target” and retry the tuning process.

If there is no message, tuning has been completed successfully. Click “Continue the Tuning Process”.

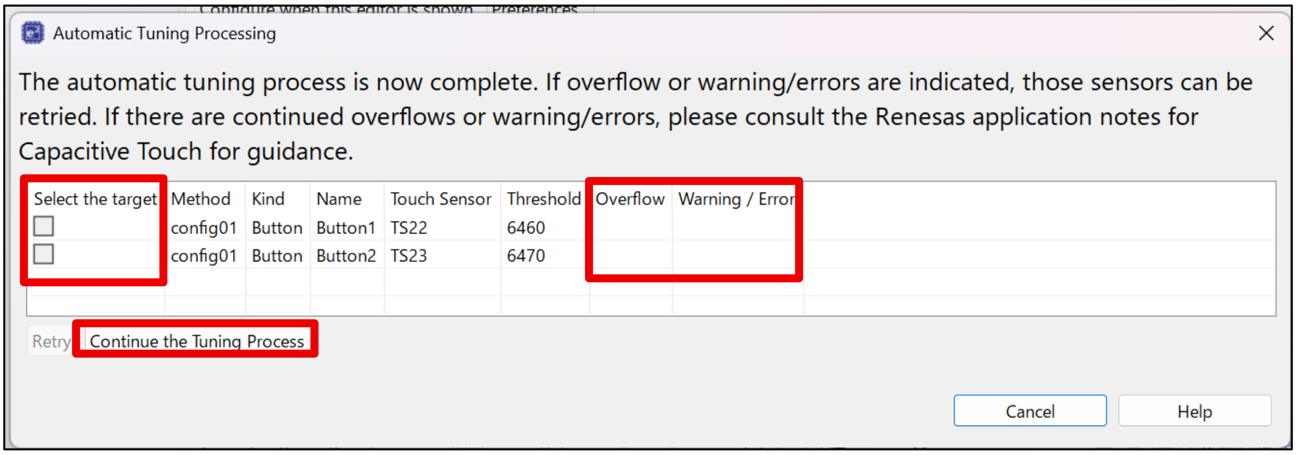


Figure4-25. Tuning (connect emulator)

(2) Output Parameter Files

Click “Output Parameter Files” from to Output Parameter Files.

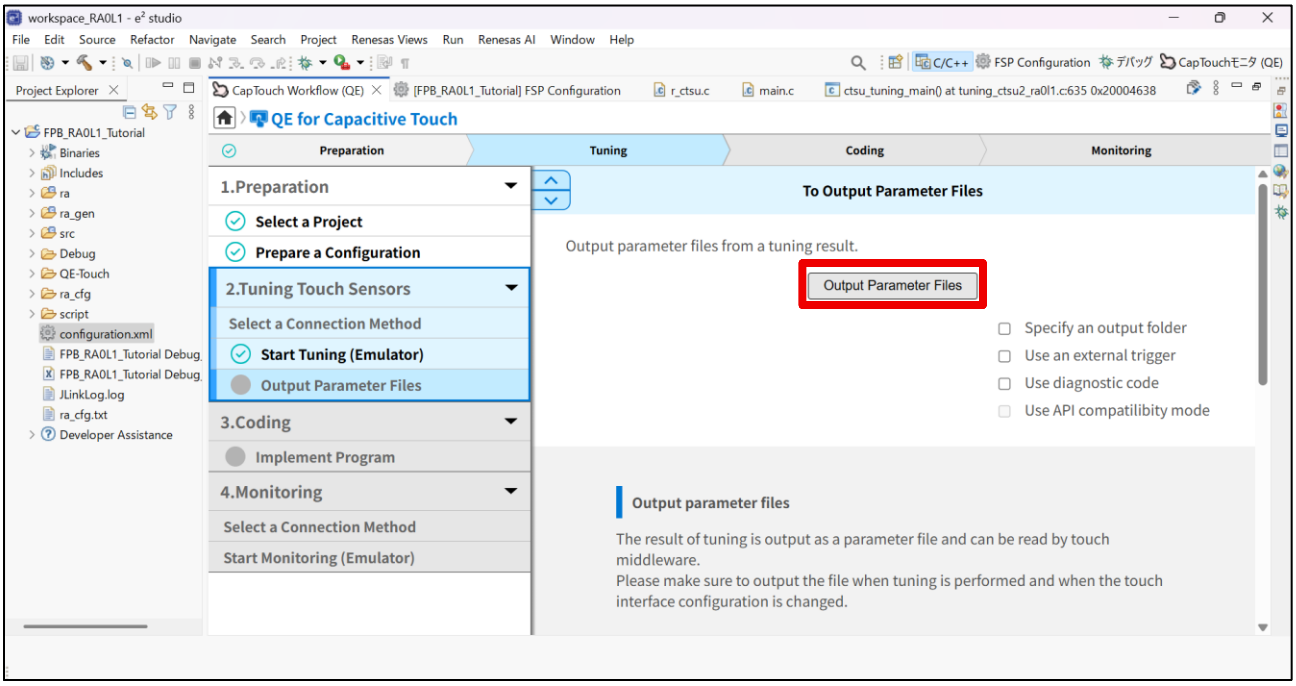


Figure4-26. Output Parameter Files

Confirm that “Output Parameter Files” is checked on CapTouch workflow, and the tuning is complete.

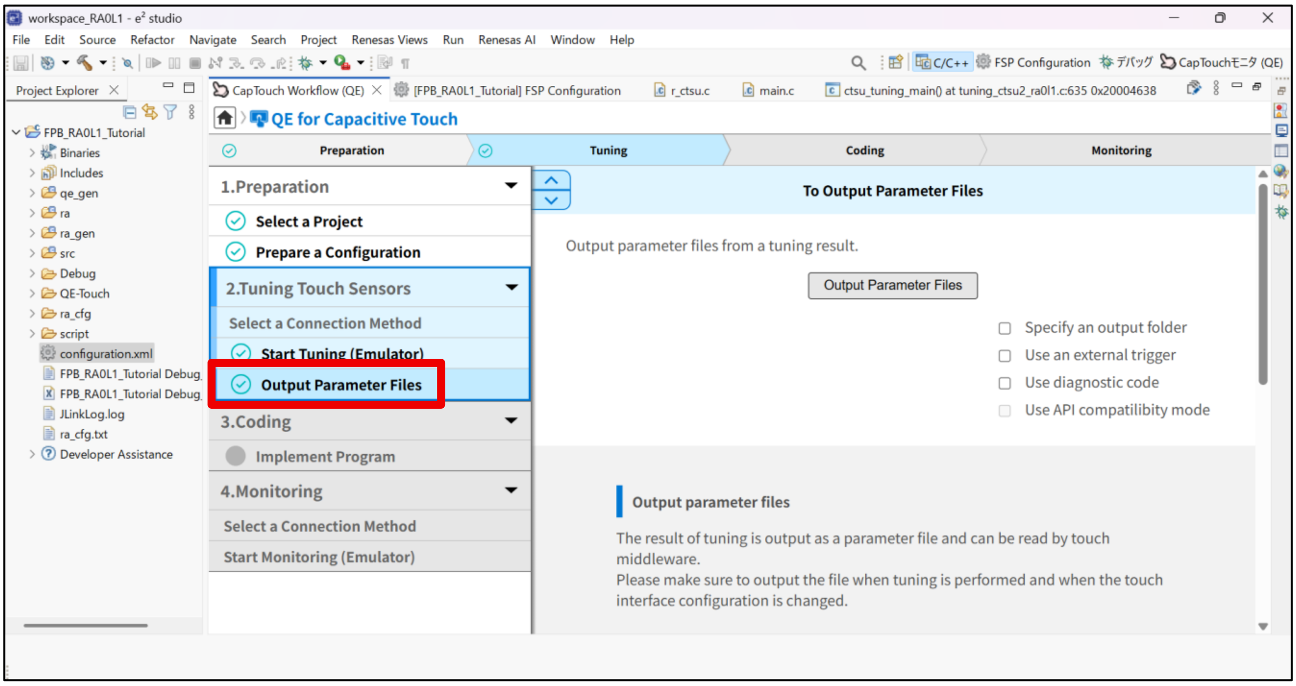


Figure4-27. Output Parameter Files

4.2.3 Program Implementation

(1) Sample program implementation

Click “Show Sample” from “Implement Program”.

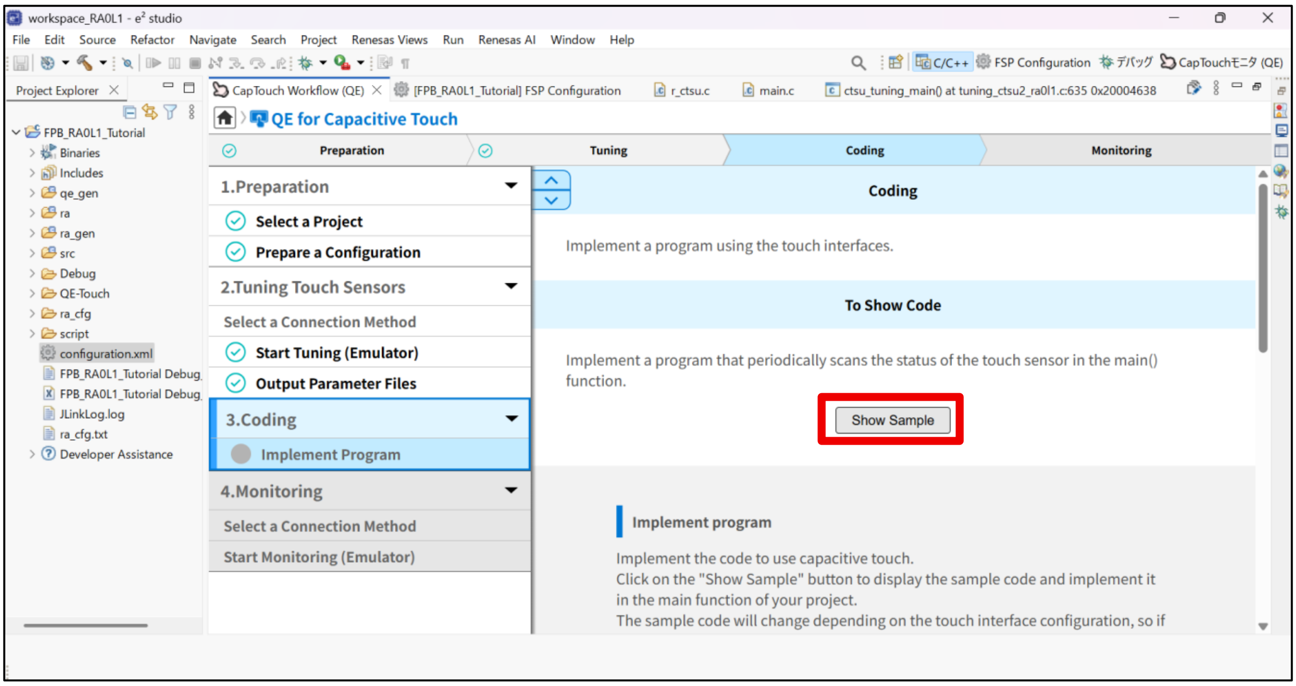


Figure4-28. Sample Program Implementation

The sample code will be displayed. Click "Output to a File," then "OK."

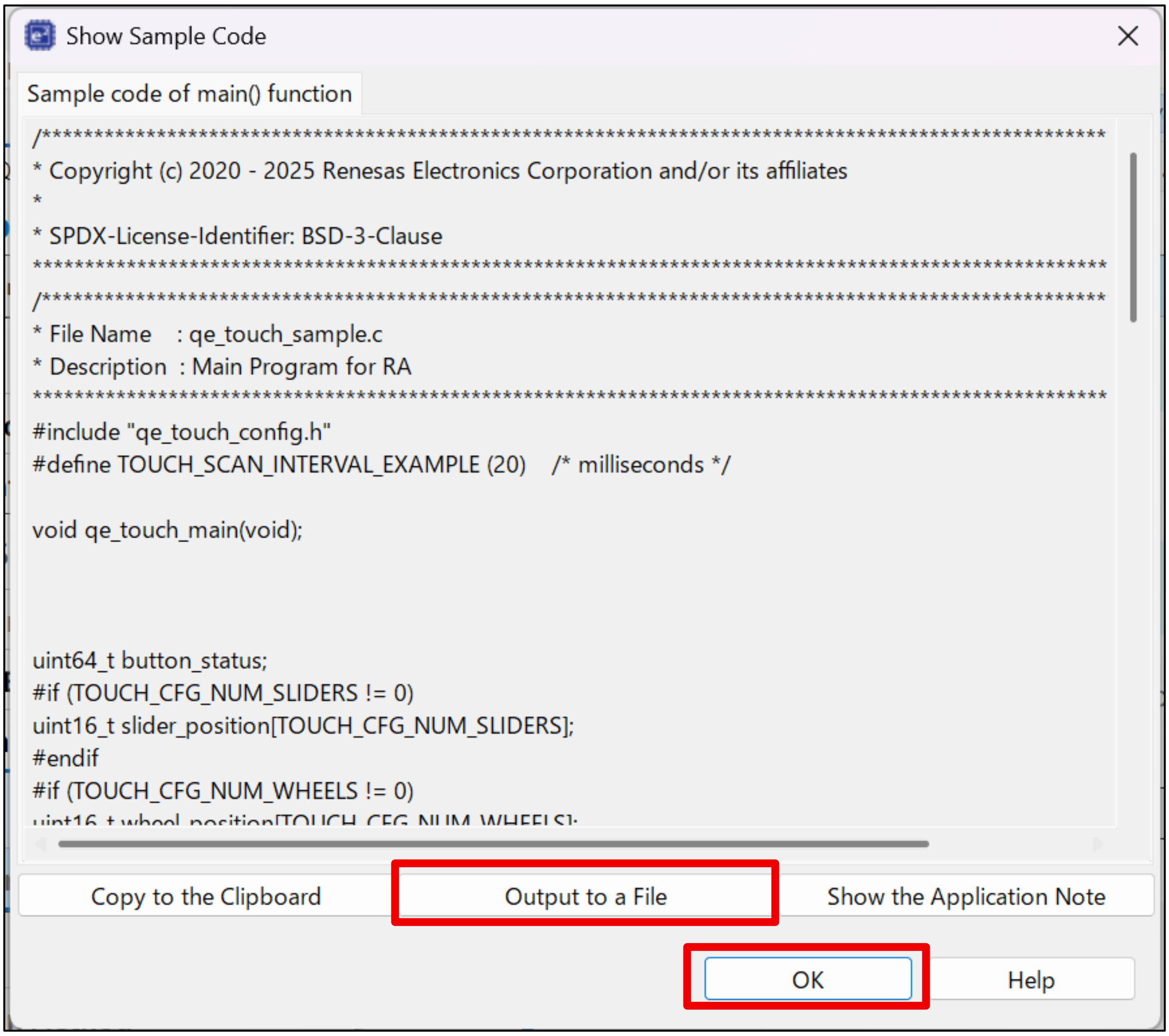


Figure4-29. Sample Program Implementation

If "FPB_RA0L1_Tutorial" → "qe_gen" → "qe_touch_sample.c" is visible in the Project Explorer, the output is complete.

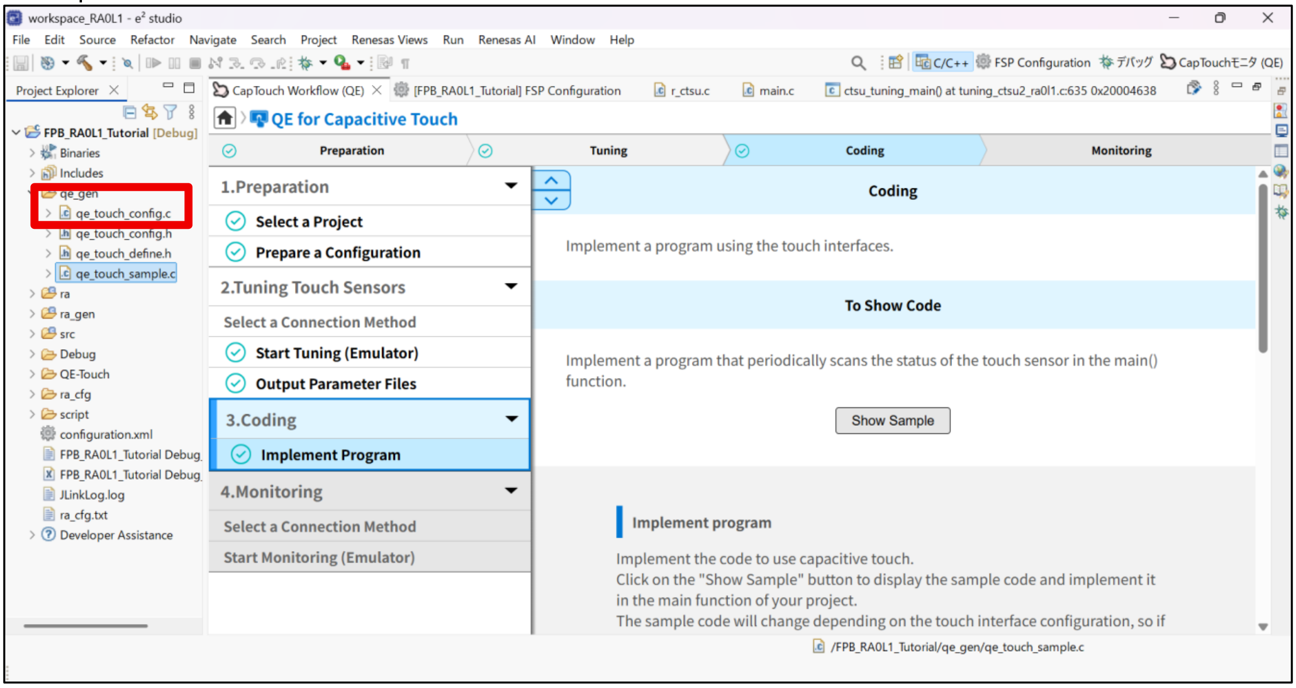


Figure4-30. Sample Program Implementation

(2) Additional implementation of function calls in the sample code

Open "FPB_RA0L1_Tutorial" → "src" → "hal_entry.c" from Project Explorer and add code that calls qe_touch_main function to hal_entry function in "qe_touch_sample.c".

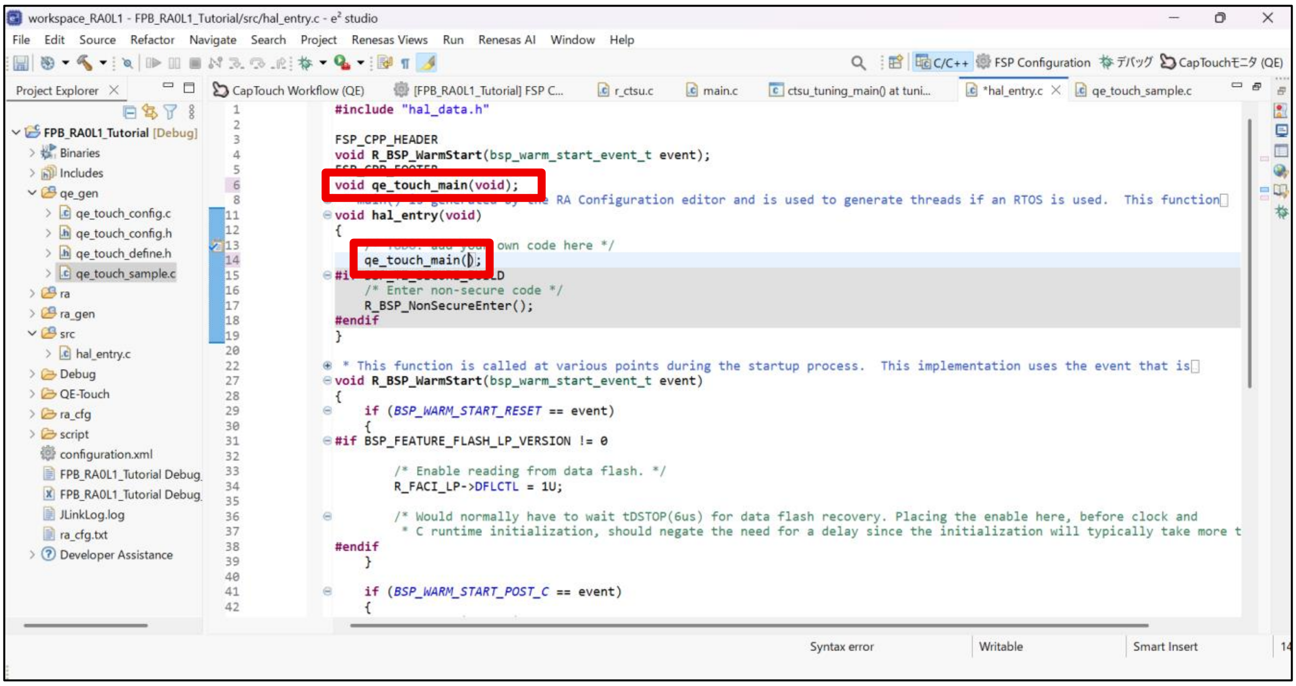


Figure4-31. Sample Program Implementation

Click the launch icon to start the build.

Build log is displayed at “Console” window.

If "Build Finished. 0 errors," is displayed at the end, it means the build was successful.

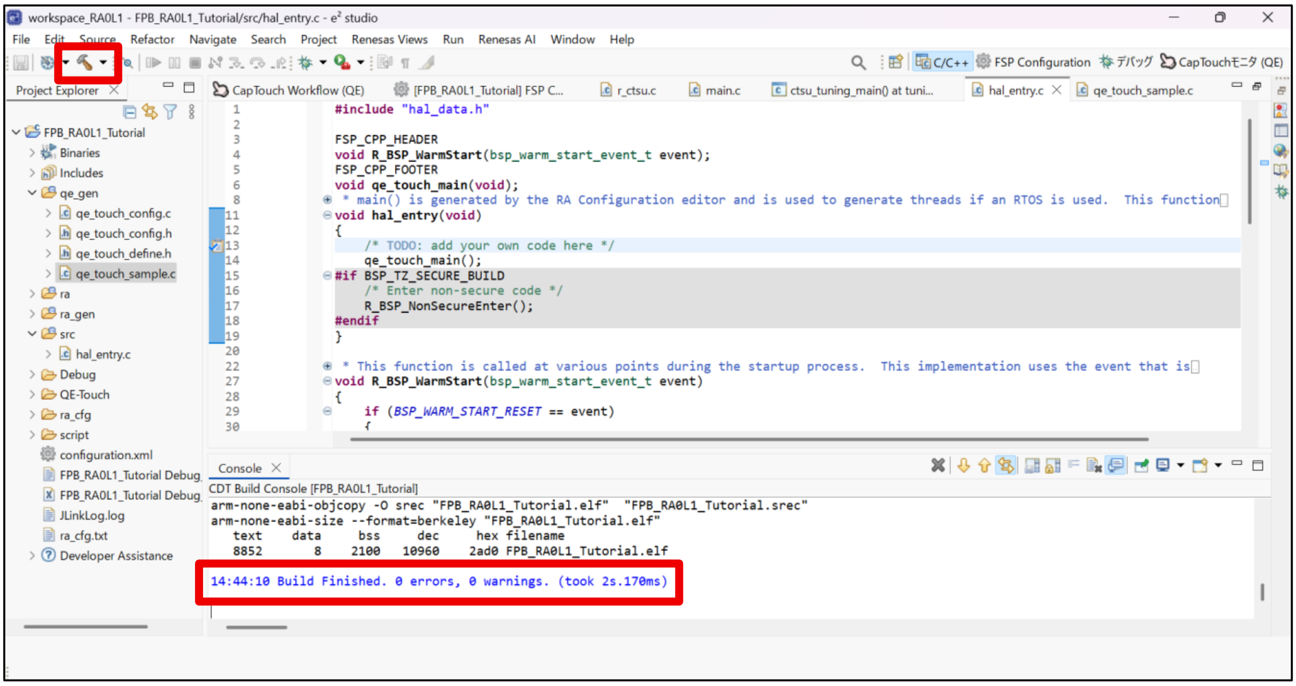


Figure4-32. Implementation of sample program

4.3 Touch sensor monitoring with QE Touch

4.3.1 Debug settings and launch

(1) Debug Configurations

From the e² studio menu bar, click “Run” → “Debug Configurations”.

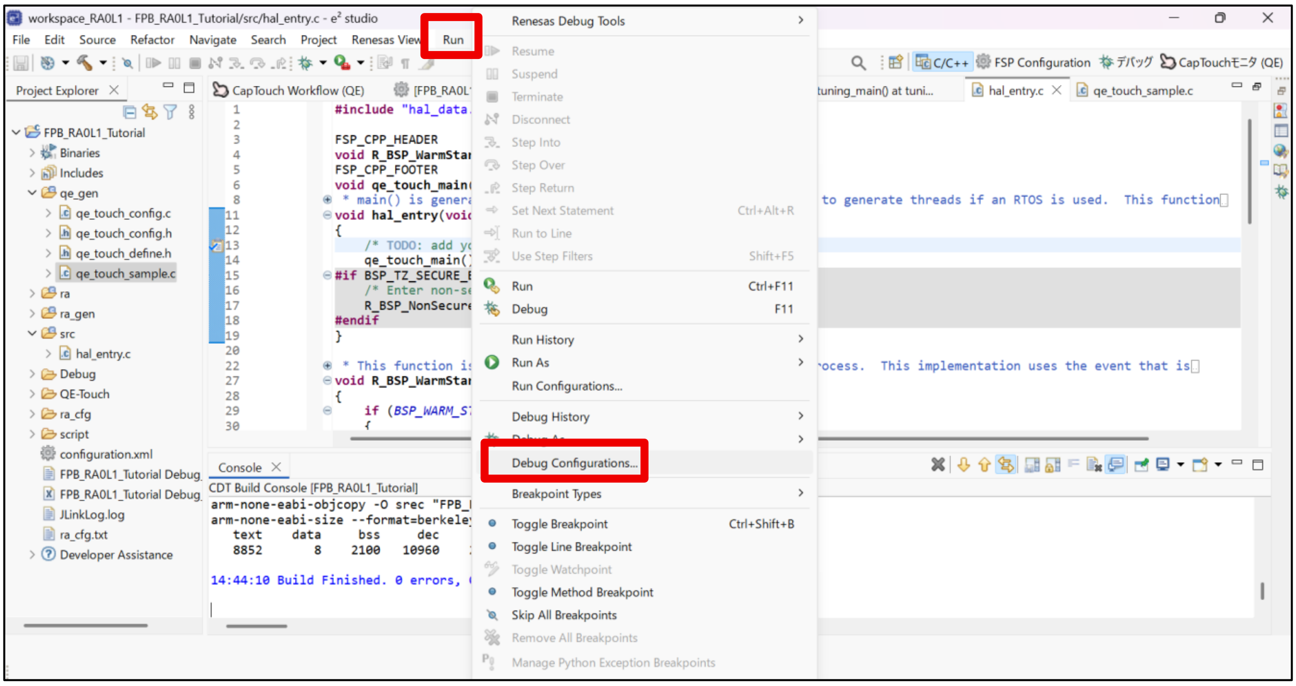


Figure4-33. Debug Configurations

Open the Debugger. Ensure that the Debug hardware is “J-Link ARM” and Target Device is “R7FA0L107”.

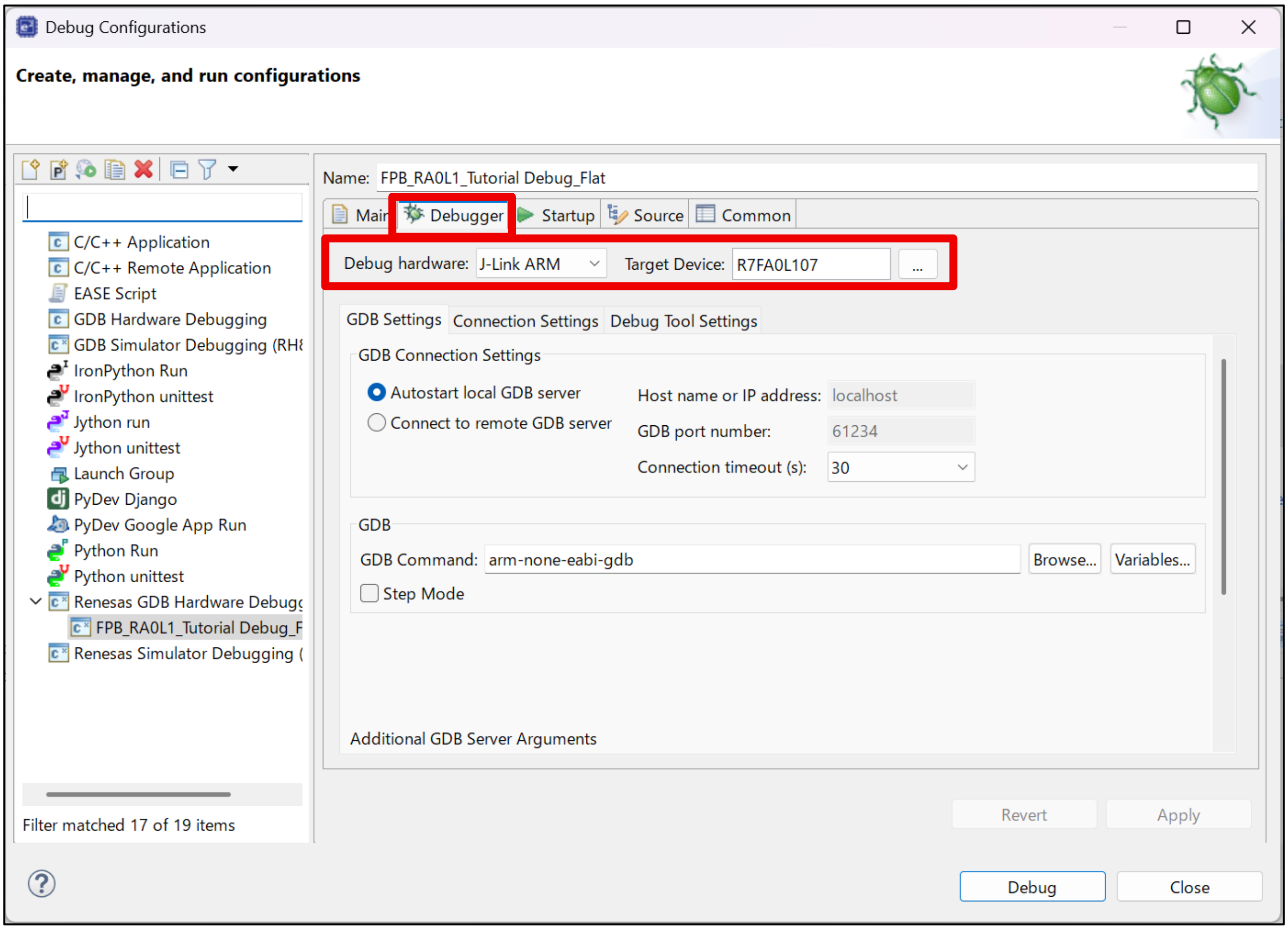


Figure4-34. Debug Configurations

Open the Startup tab and check “Resume” under Runtime Option.

The Debug Configuration is now completed. Click “Apply” then “Debug”.

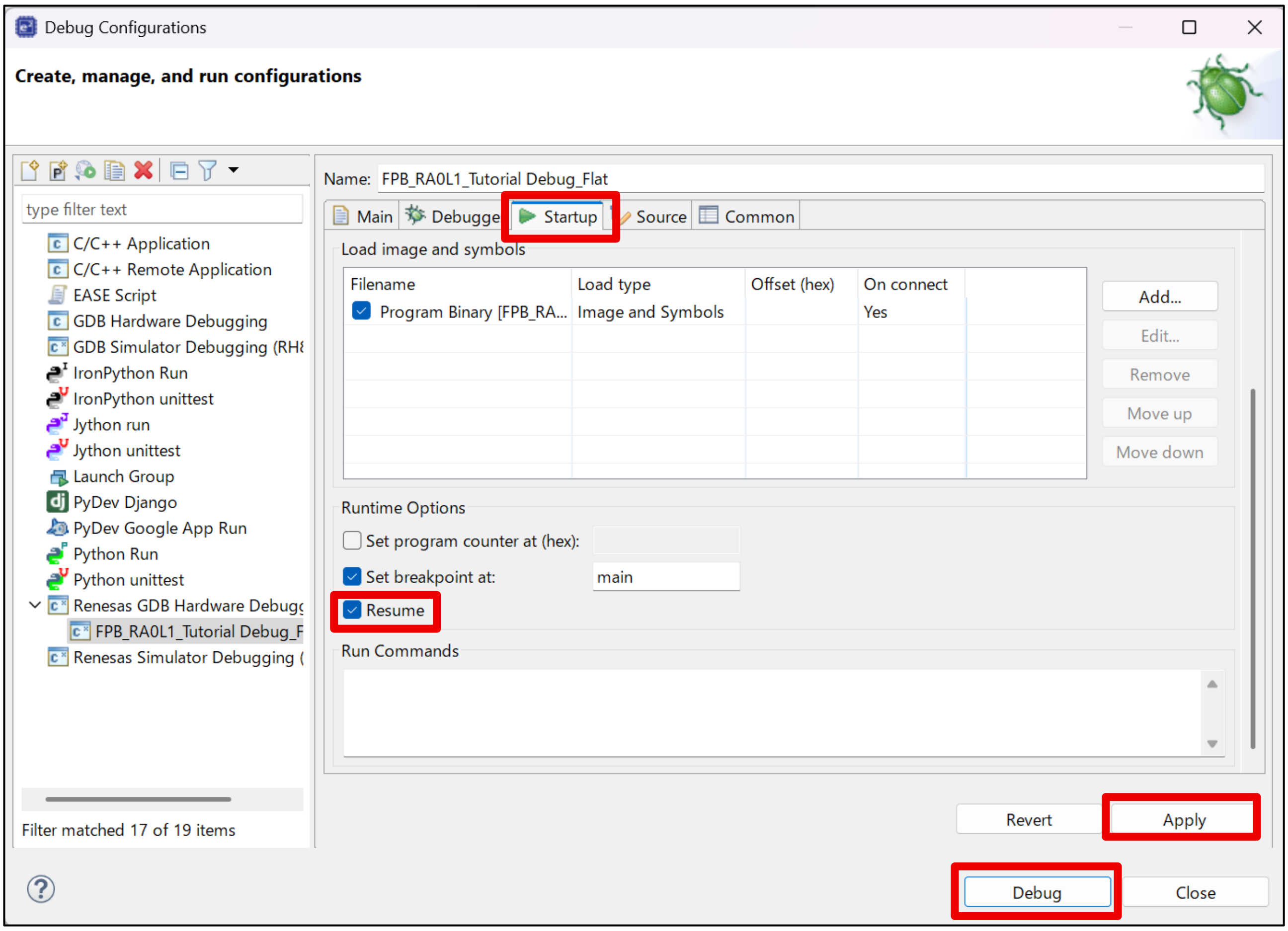


Figure4-35. Debug Configurations

A pop-up window will appear, asking if you would like to switch perspective to the "Debug" perspective. Click the switch to proceed.

*Even if you click "No", you can switch the screen later by pressing the "Debug" button at the top right of e^2 studio.

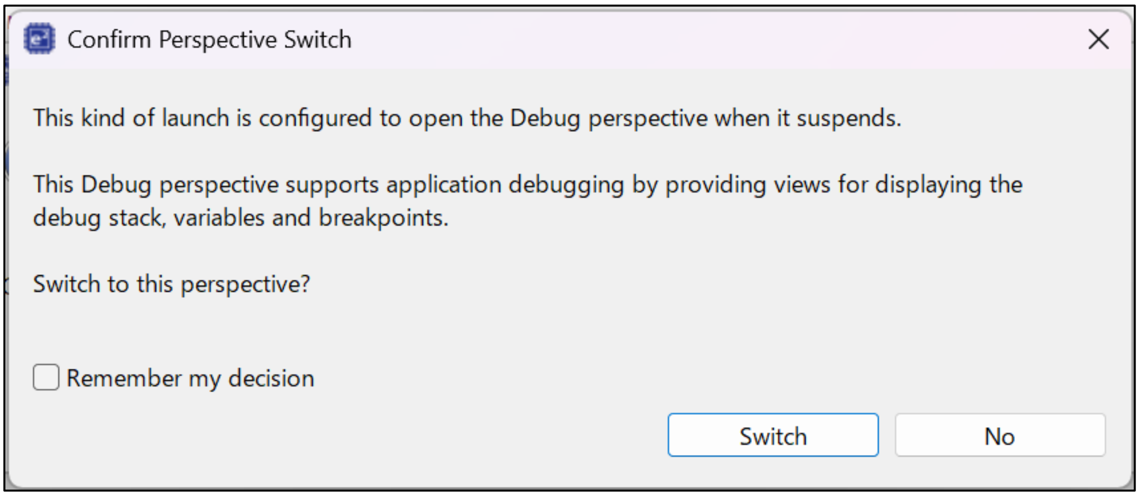


Figure4-36. Debug Configurations

(2) Execution

After debugging, the program will pause at the beginning of main function. (This is because of the default setting for e² studio projects, which is to pause at the main function.)

This indicates that SystemInit() has been completed, and the initial settings have been applied. Run the program by clicking the resume button.

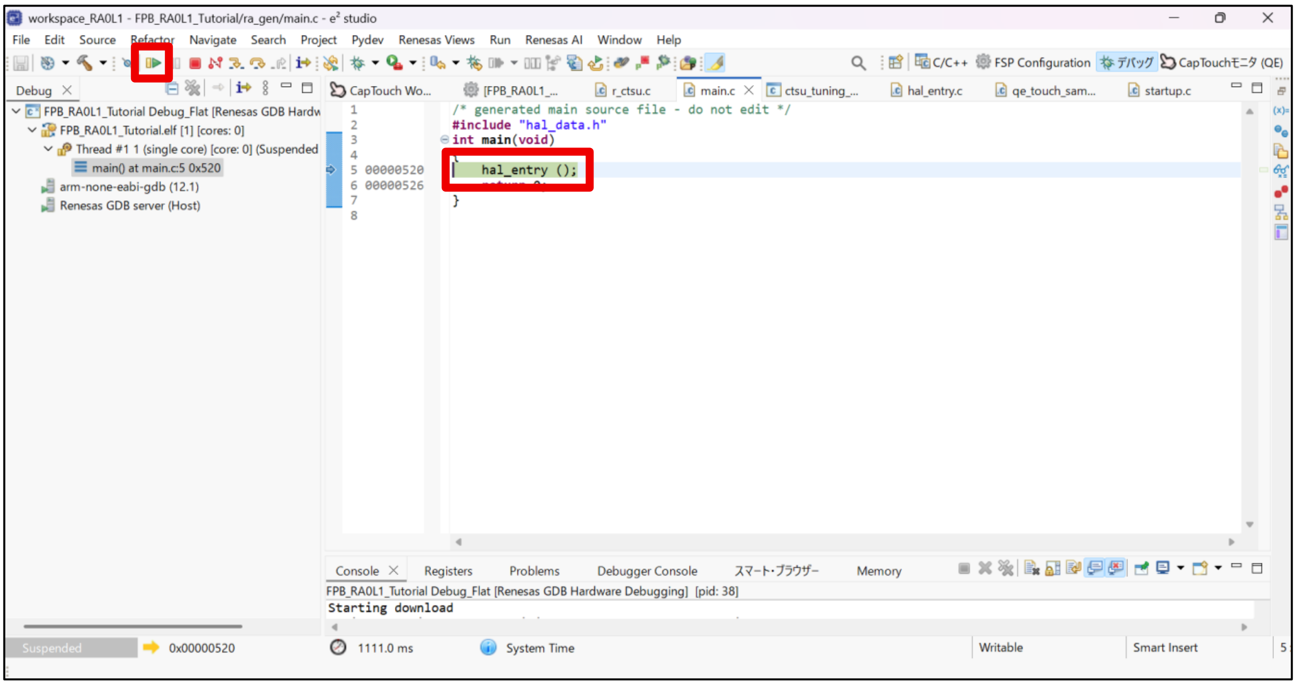


Figure4-37. Execution

4.3.2 Monitoring

Open the CapTouch workflow and set the connection method for monitoring to “Emulator”.

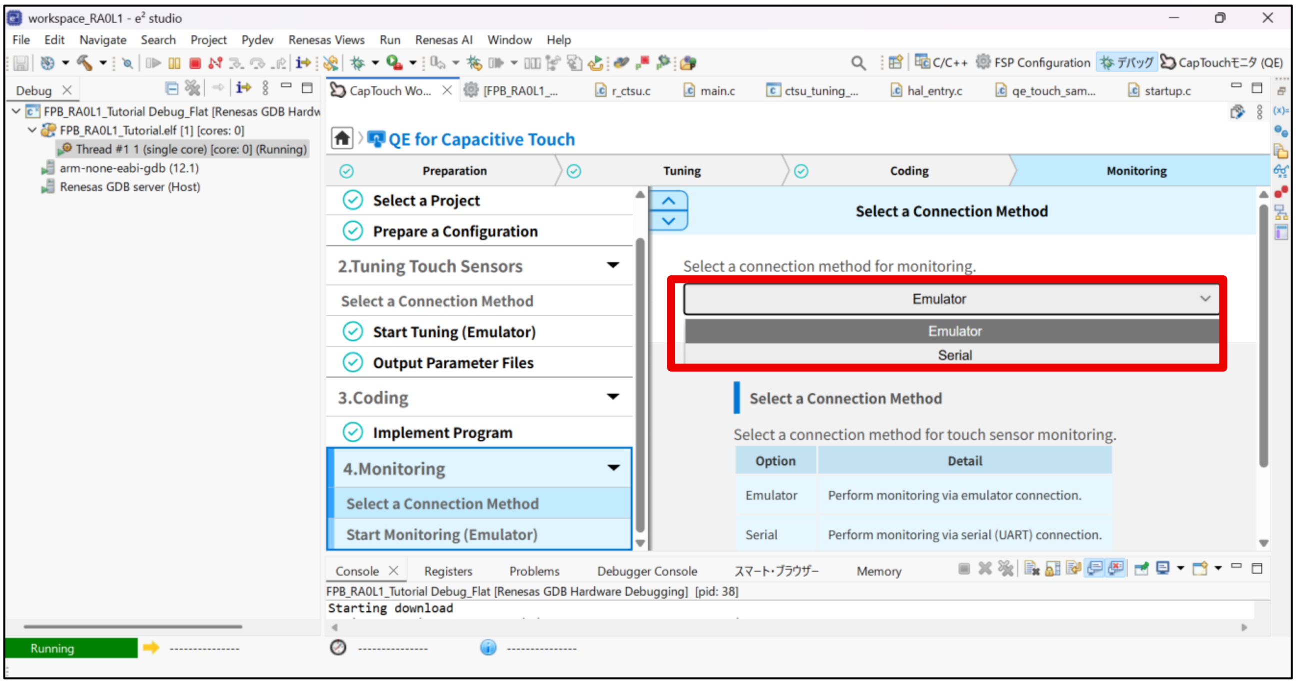


Figure4-38. Selecting a connection method for monitoring

Click “Show Views” from Start Monitoring (Emulator) at CapTouch workflow.

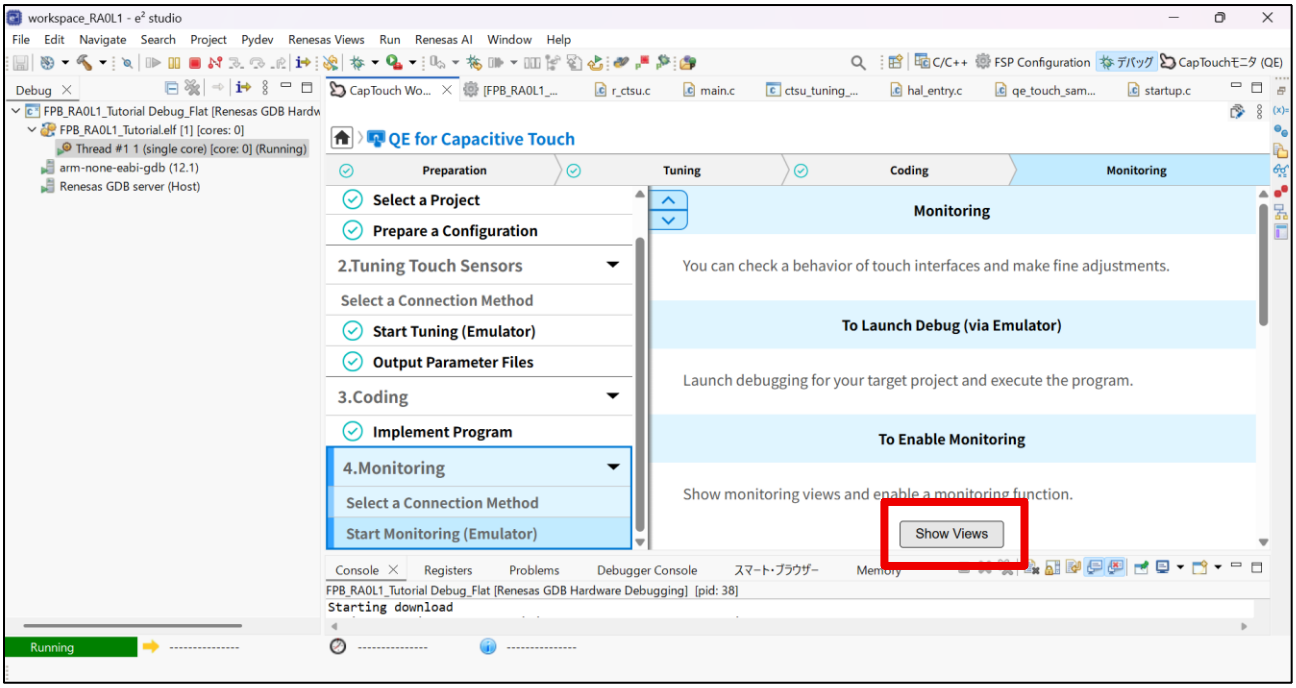


Figure4-39. Starting monitoring

Ensure that “CapTouch Board Monitor”, “Cap Touch Status Chart” and “CapTouch Multi Status Chart” are displayed.

In this state, “Monitoring: Disabled” is indicated.

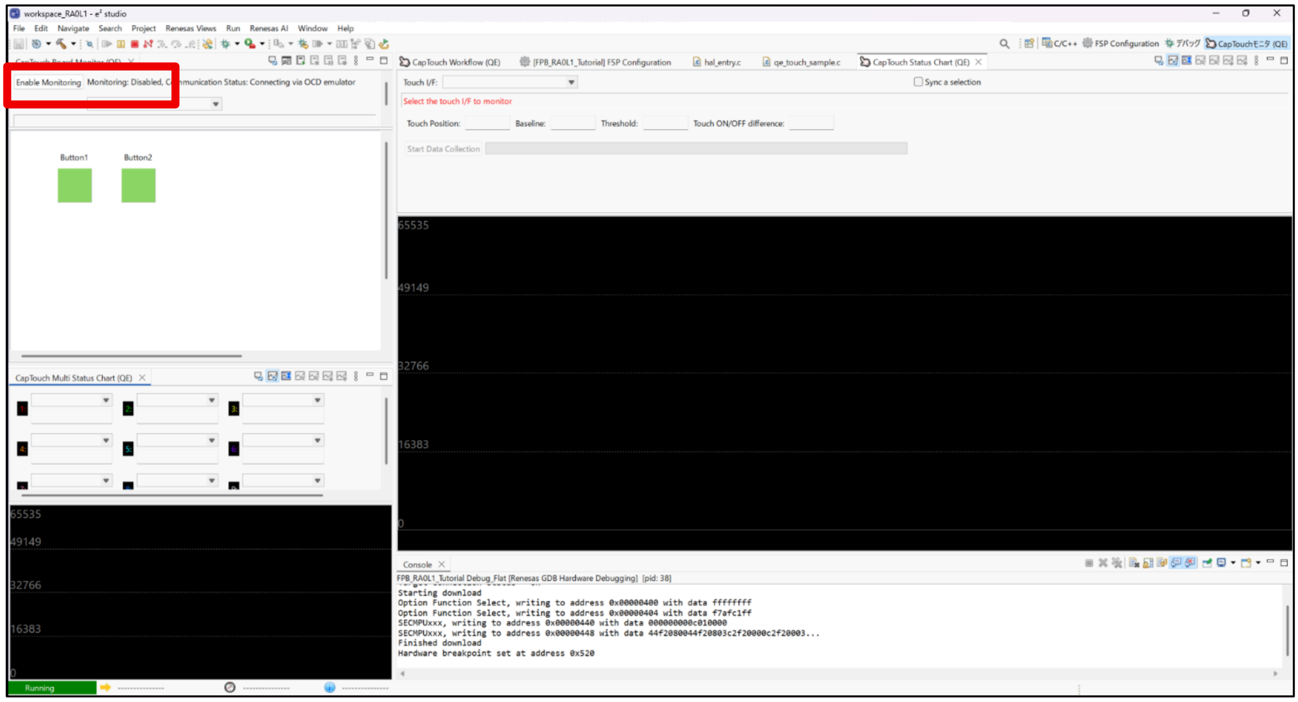


Figure4-40. Monitoring : Disabled

Click “Enable Monitoring” and “Monitoring: Enabled” will be indicated.

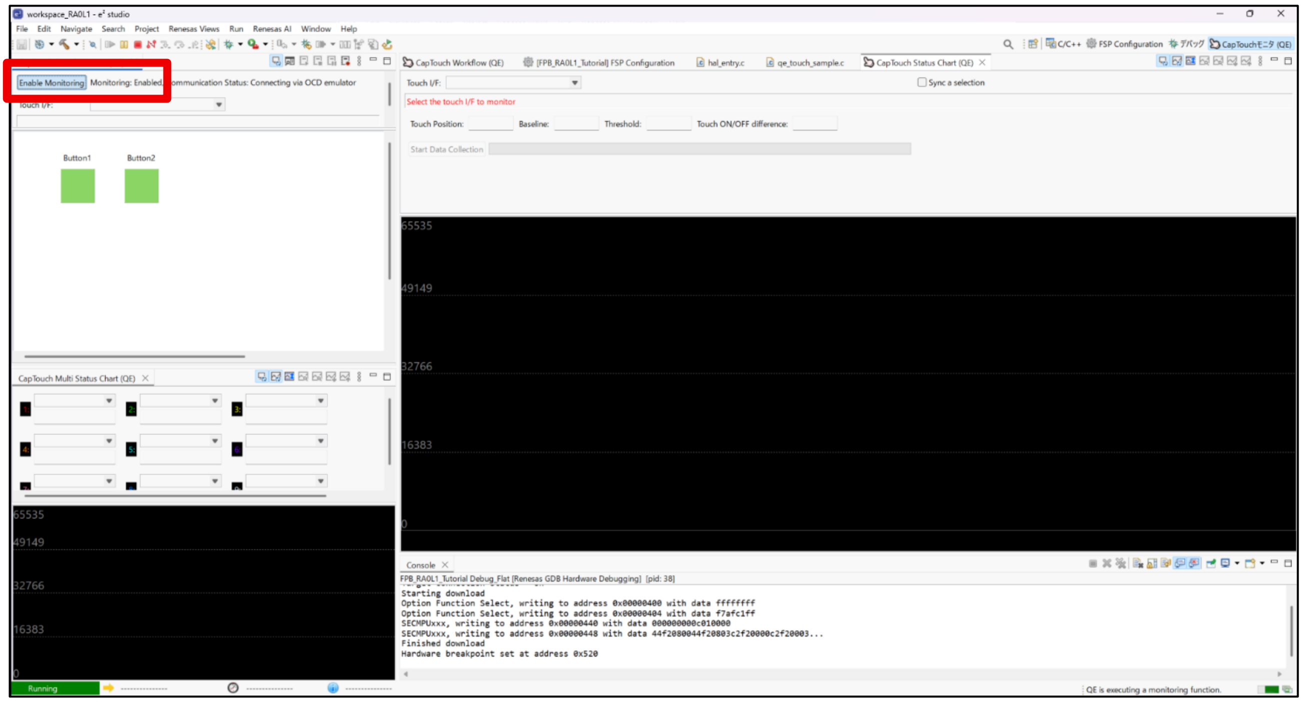


Figure4-41. Monitoring : Enabled

In the CapTouch Board Monitor, you can check the ON/OFF status of the touch sensors.

When a touch is detected at Button1, a finger icon is displayed.

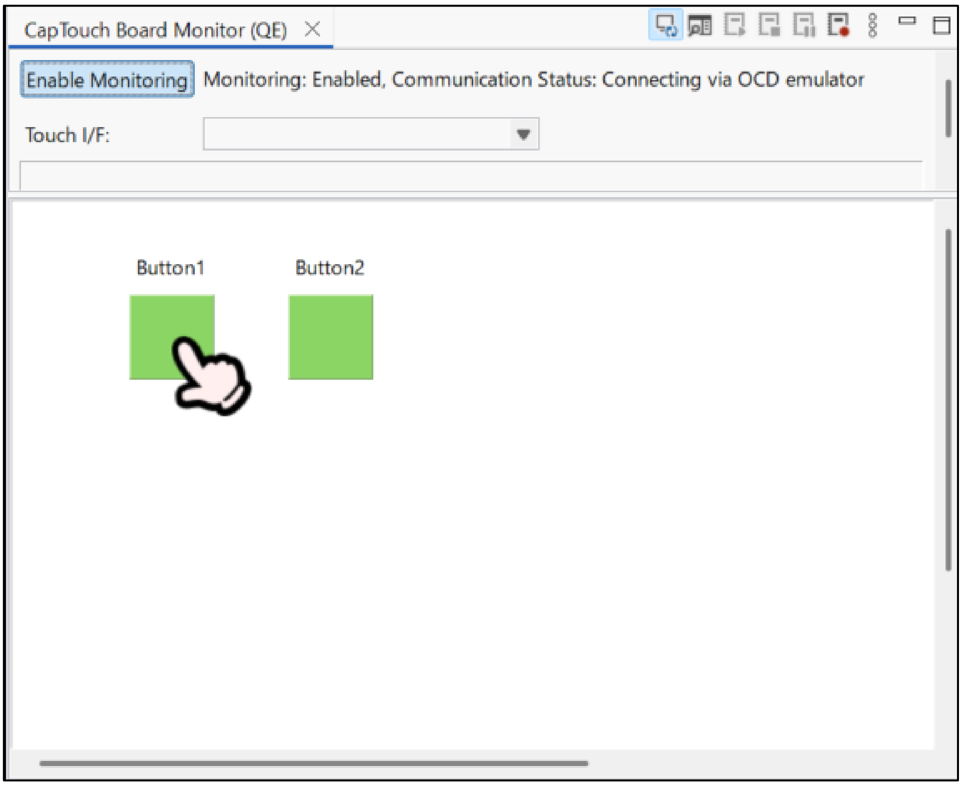


Figure4-42. Monitoring touch detection at Button1

When a touch is detected at Button2, a finger icon is displayed.

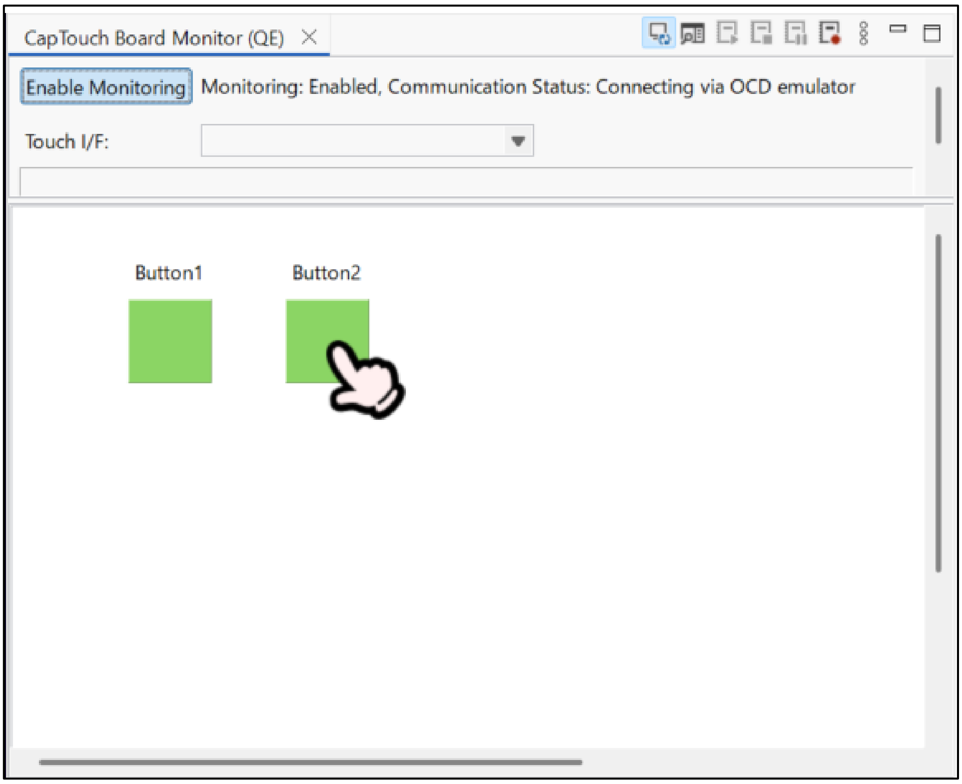


Figure4-43. Monitoring touch detection at Button2

In the CapTouch Status Chart, the values for one selected touch I/F can be monitored.

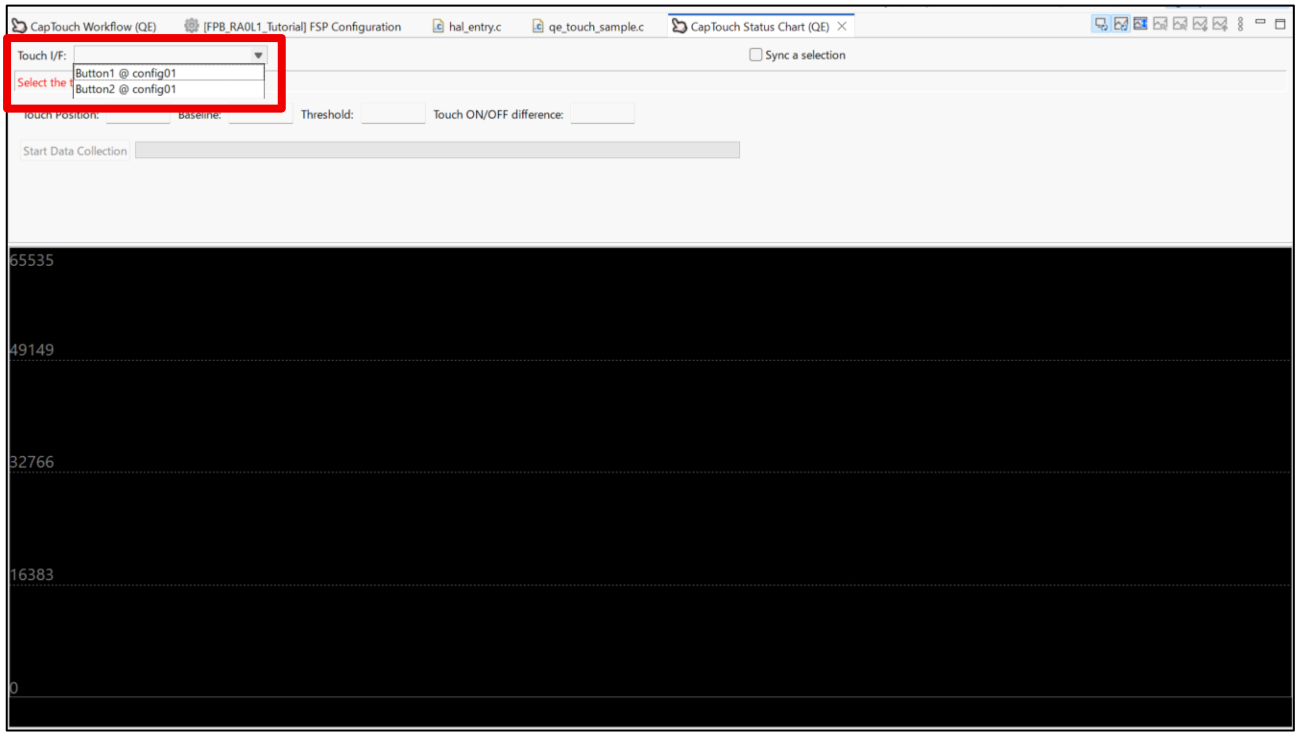


Figure4-44. CapTouch Status Chart: Select touch I/F

This shows the state when Button1 is touched.

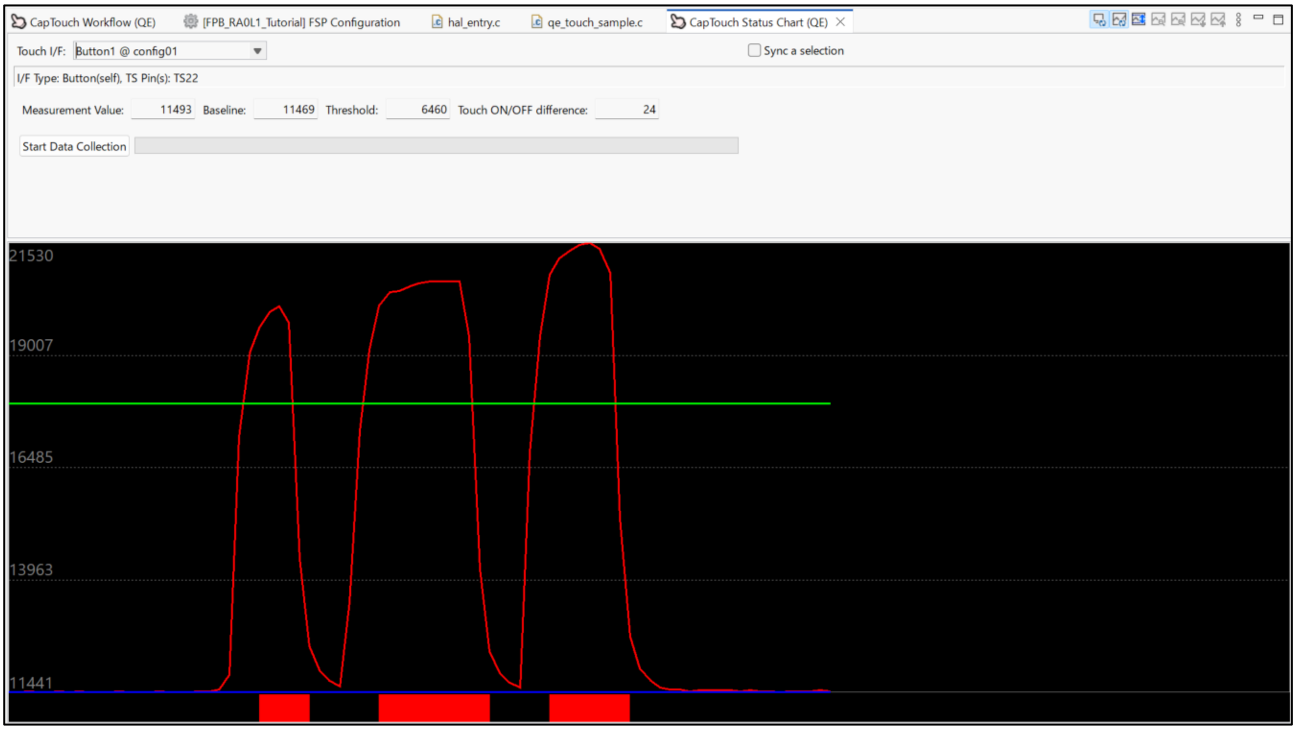


Figure4-45. CapTouch Status Chart: Touch detection at Button1

This shows the state Button2 is touched.

Figure4-46. CapTouch Status Chart: Touch detection at Button2

In the CapTouch Multi Status Chart, maximum of 9 values can be monitored simultaneously, similar to the CapTouch Status Chart.

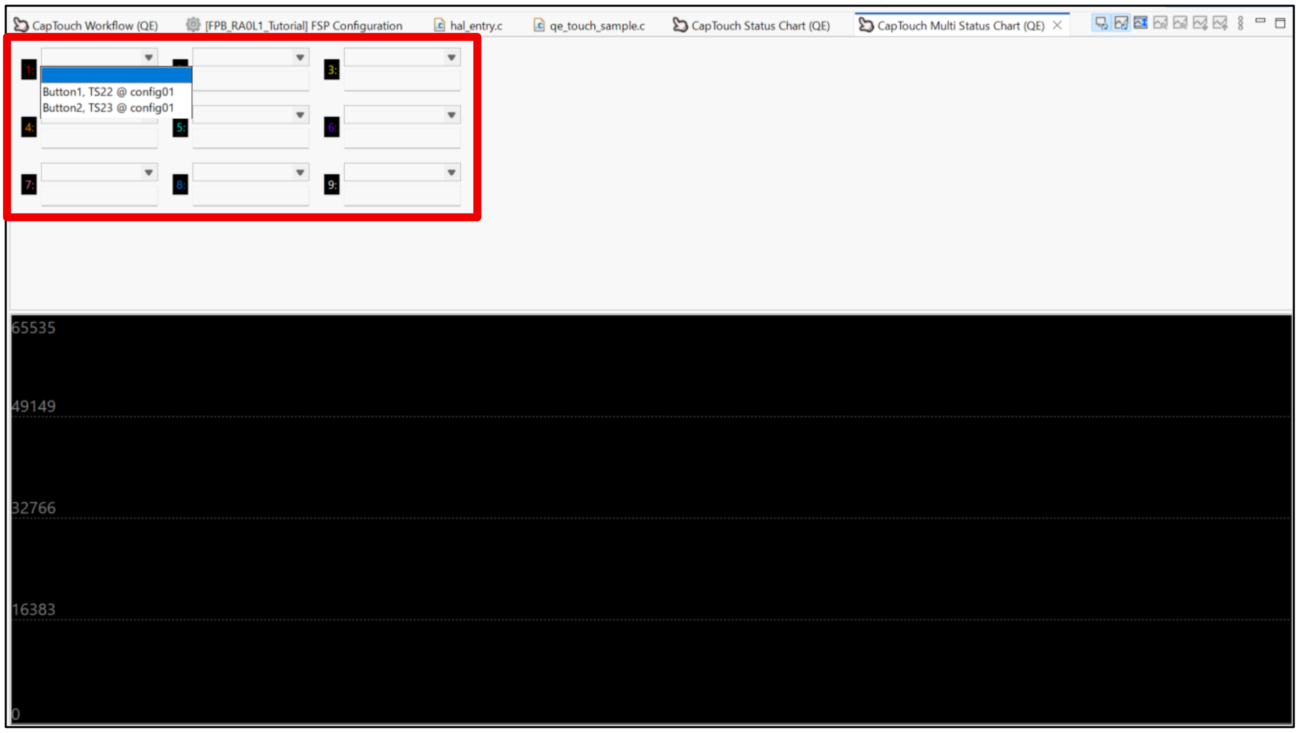


Figure4-47. CapTouch Status Chart: Select touch I/F

This shows the state when Button1 and Button2 are touched.

Button1: Red line

Button2: Green line

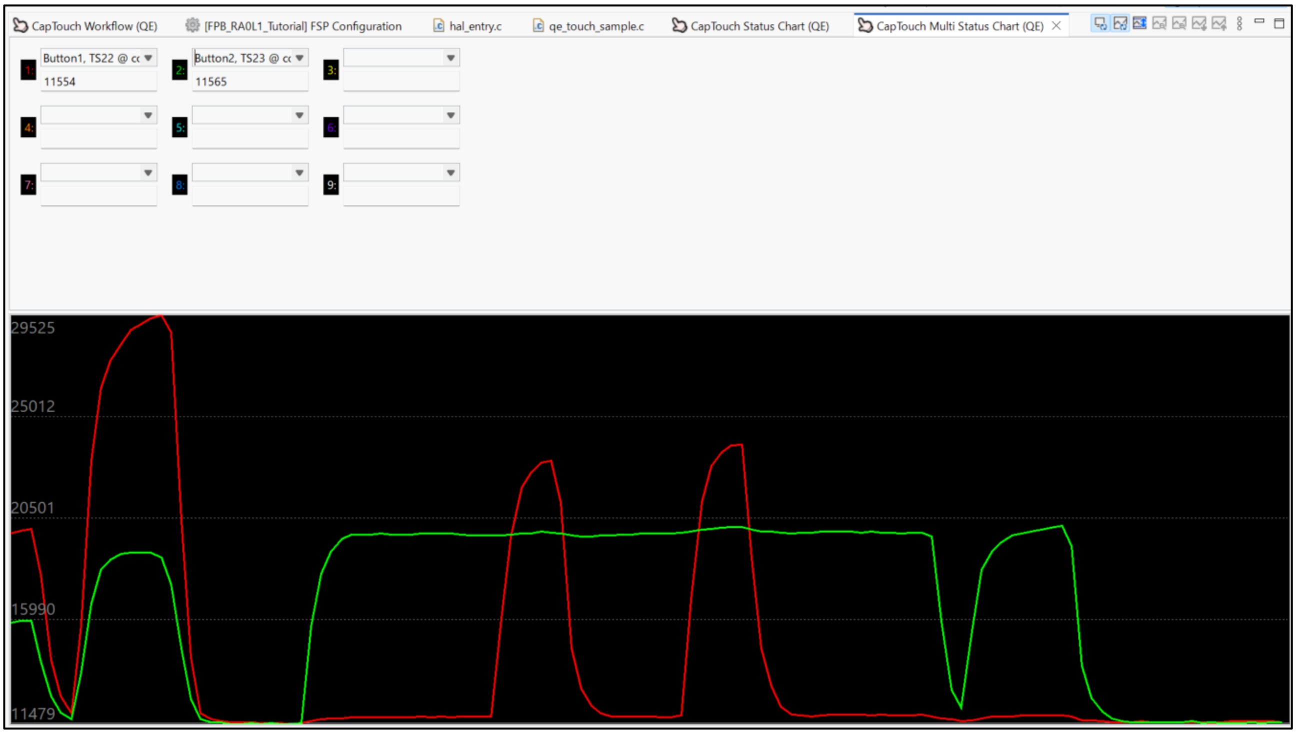


Figure4-48. CapTouch Status Chart: Touch detection

5. Create a sample program using a touch sensor

5.1 FSP Configurator settings for the sample program

5.1.1 Timer settings

Open the "Stacks" tab in the FSP Configuration window and add "Timers".

Add to click "New Stack" → "Timer" → "Timer, Independent Channel, 16-bit and 8-bit Timer Operation (r_tau)".

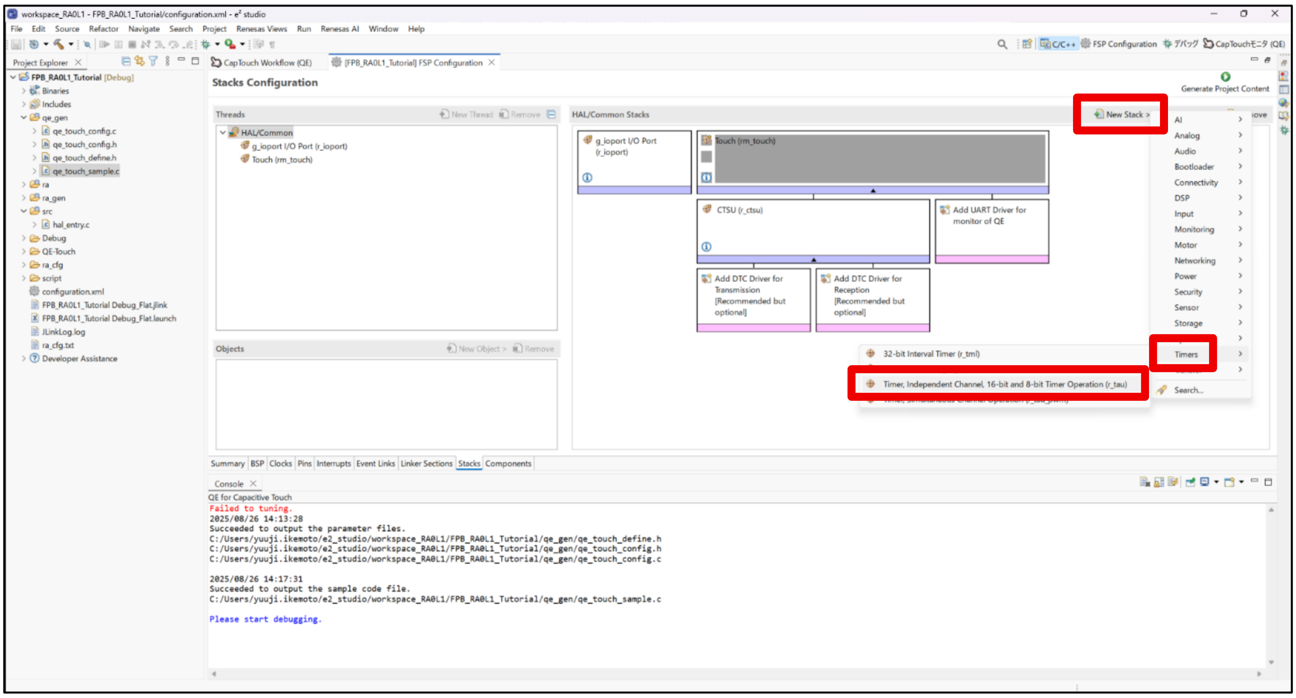


Figure5-1. Timer settings

Ensure that the stack name "g_timer0" has been added.

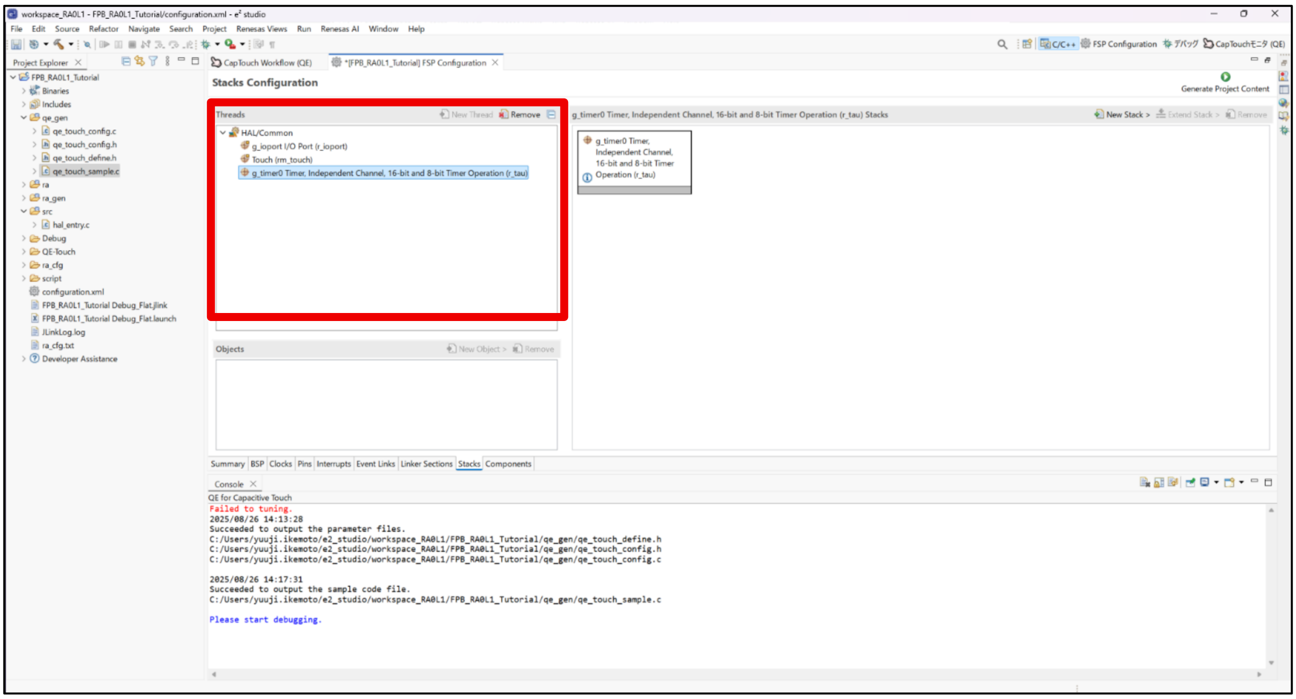


Figure5-2. Timer settings

When the “Properties” window is displayed, clicking the “g_timer0” block will display detailed timer settings.

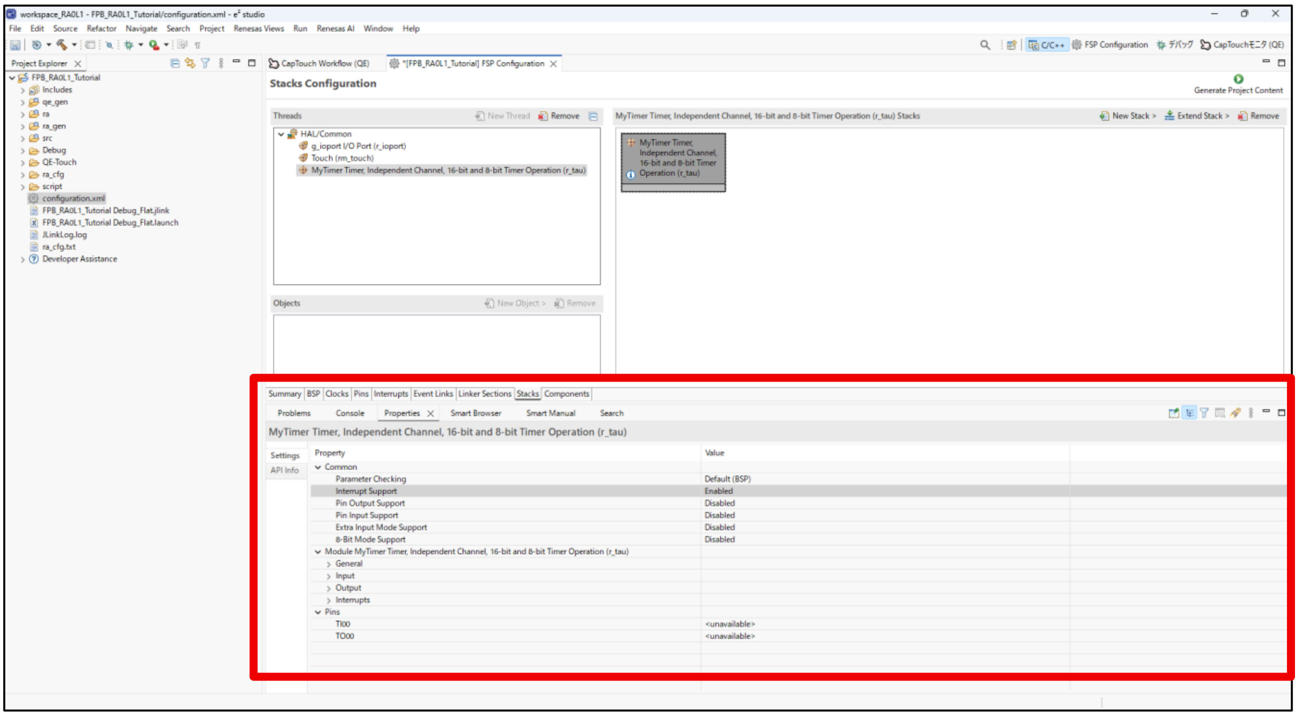


Figure5-3. Timer settings

The “Property” window can also be displayed by selecting "Window" → "Show View" → "Properties" from the e^2 studio menu bar.

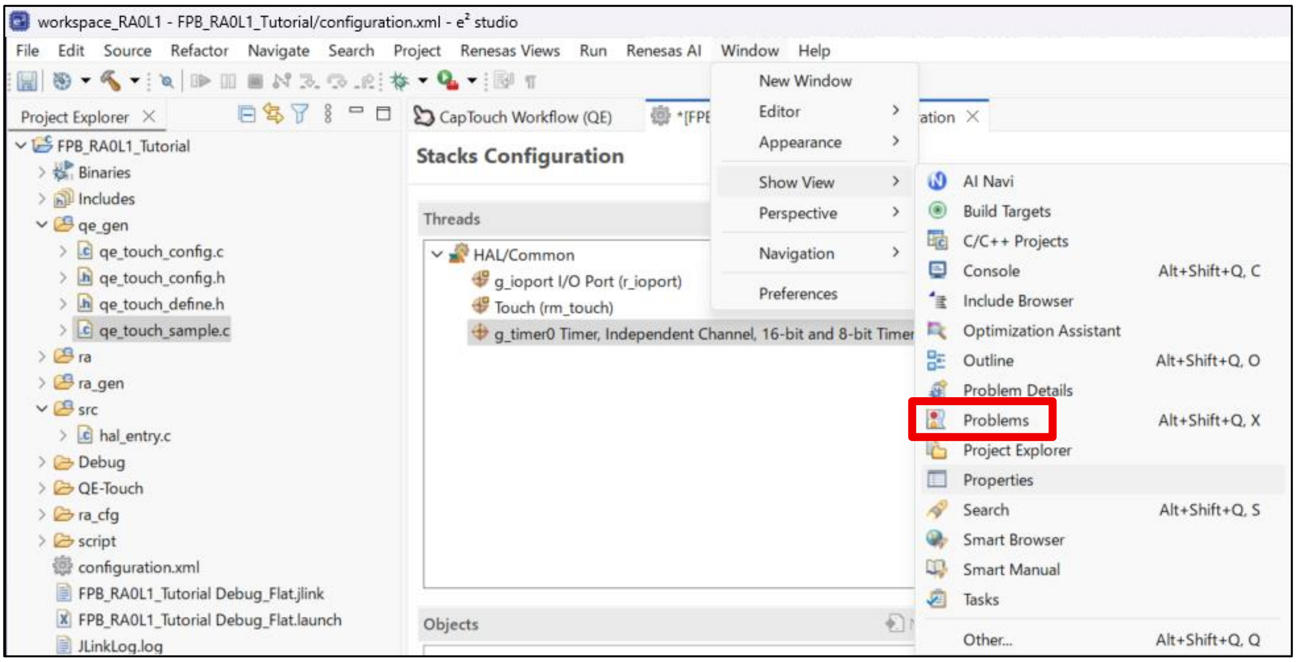


Figure5-4. Timer settings

Displays the "General" list. Configure the timer basic settings in this window.

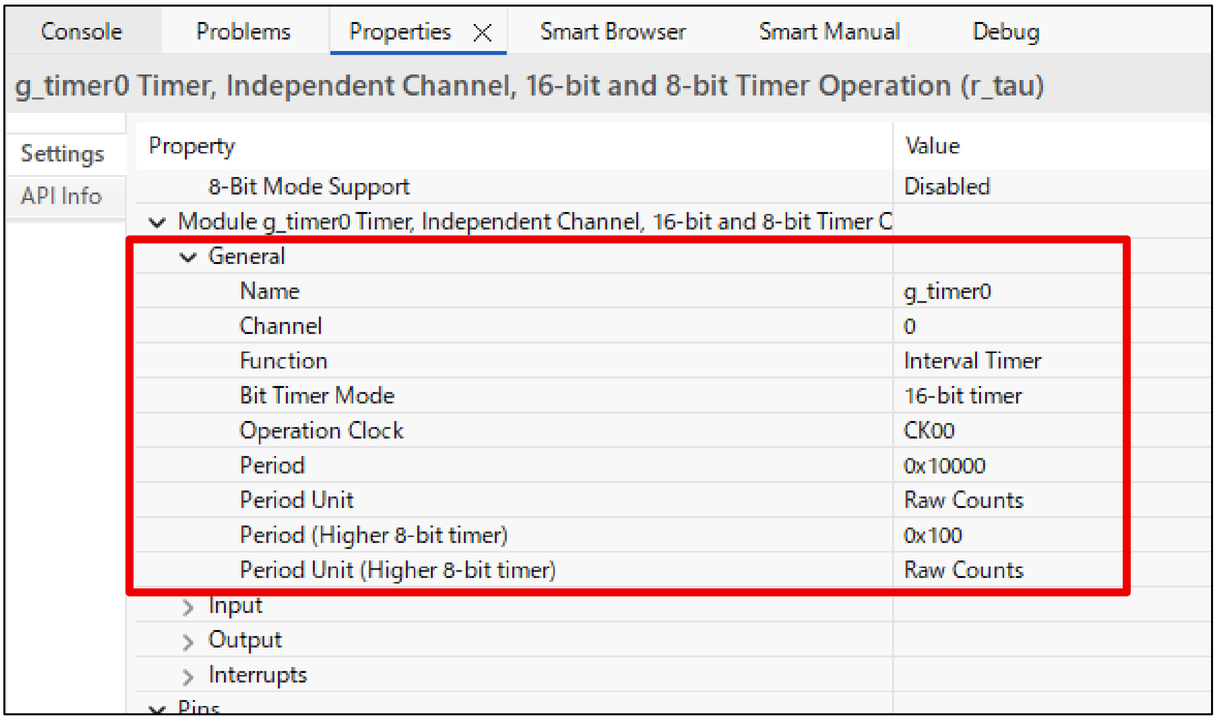


Figure5-5. Timer settings

Set the following items in “General”

- Name : Timer module name
  “MyTimer” in this application note.
- Channel : TAU channel number
  Use channel 0. Set “0”.
- Function : TAU channel function
  Use as Interval Timer. Set “Interval Timer”.
- Bit Timer: TAU channel counter bit size
  Use 16-bit timer. Set “16-bit timer”.
- Operation Clock: TAU operation clock.
  Set “CK00”.
- Period / Period Unit : 16-bit timer period.
  Set “Period =1” and “Period Unit = Milliseconds” to create a 1ms period.

This completes the General settings.

Module g_timer0 Timer, Independent Channel, 16-bit and 8-bit Timer C	
General	
Name	MyTimer
Channel	0
Function	Interval Timer
Bit Timer Mode	16-bit timer
Operation Clock	CK00
Period	0x1
Period Unit	Milliseconds
Period (Higher 8-bit timer)	0x100
Period Unit (Higher 8-bit timer)	Raw Counts

Figure5-6. Timer settings

Displays the “Interrupts” list. Configure the interrupt settings in this window.

Interrupts	
Setting of starting count and interrupt	Timer interrupt is not generated when counting is started/Start trigger is invalid during counting operation.
Callback	NULL
Interrupt Priority	Disabled
Higher 8-bit Interrupt Priority	Disabled

Figure5-7. Timer settings

Set Callback to an interrupt handling function implemented by the user.

The default setting “NULL” indicates that no function is implemented.

In this application note, “MyCallback” is created as callback function.

Interrupts	
Setting of starting count and interrupt	Timer interrupt is not generated when counting is started/Start trigger is invalid during counting operation.
Callback	MyCallback
Interrupt Priority	Disabled
Higher 8-bit Interrupt Priority	Disabled

Figure5-8. Timer settings

Set Interrupt priority to “Interrupt Priority”.

"Disabled" by default means interrupts are disabled.

Set the priority to a value from 0 to 3 to enable interrupts.

In this project, set it to "Priority3".

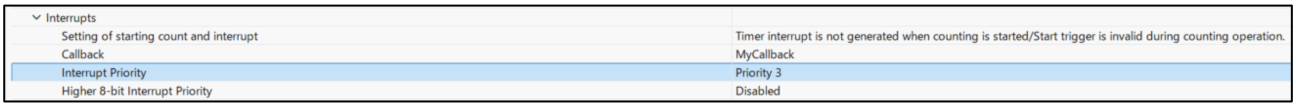


Figure5-9. Timer setting

Timer settings are completed.

5.1.2 Pin settings

Pins to use to control LED are shown below:

Table 5-1 FPB-RA0L1 LED PORT

Name	Color	RA MCU PORT
LED1	Green	P002(High Lightning)
LED2	Green	P104(High Lightning)
LED5	Green	P401(Low Lightning)
LED6	Green	P400(Low Lightning)

Open the “Pins” tab in the FSP Configuration window and confirm that the Mode of the terminal corresponding to the LED is set to Output mode. The pin configuration is complete.

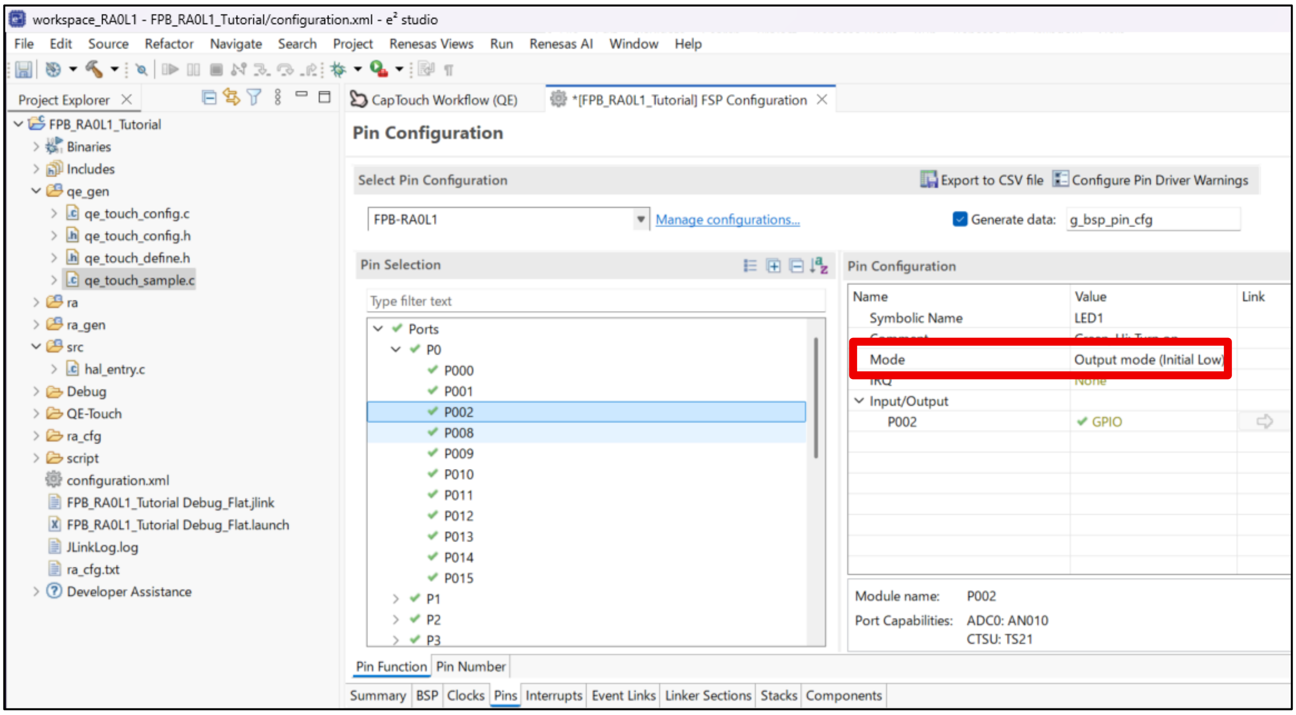


Figure5-10. LED1 Pin setting

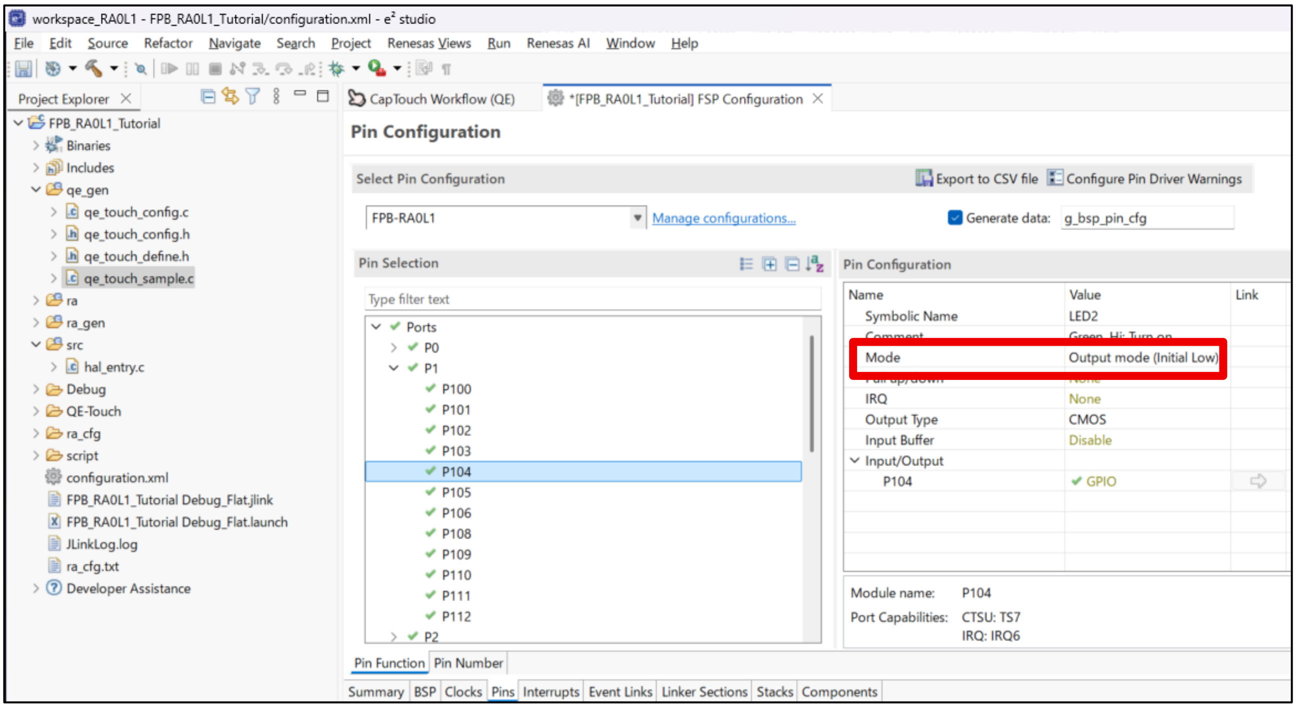


Figure5-11. LED2 Pin setting

The settings required to create this program are complete. Click the “Generate Project Content” button to generate the code.

Figure5-12. Pin settings

A confirmation window will appear, asking whether to save to the Configuration.xml file. Press the "Continue" button to save.

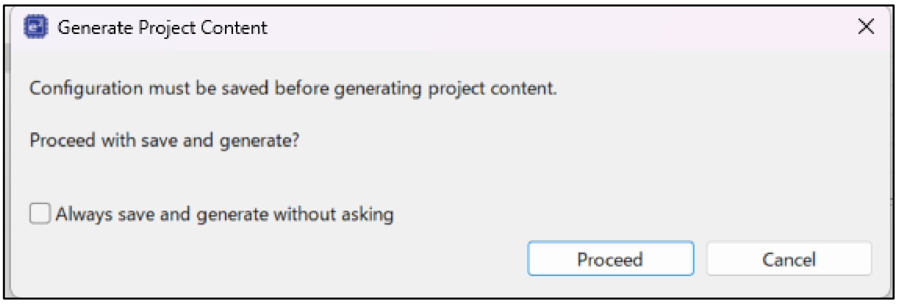


Figure5-13. Pin setting

5.2 Coding the sample program

Implement the code in the main program and interrupt function.

The contents to implement are as follows:

- Main program
  - Start the timer
  - Start / Stop the LED blinking and change the blinking speed when a touch is detected on the Touch Button.
- Interrupt program
  - Count variables for time measument

The file to be modified by user is ¥qe_gen¥qe_touch_sample.c in the Project Folder.

Dubble-click qe_touch_sample.c to open the file.

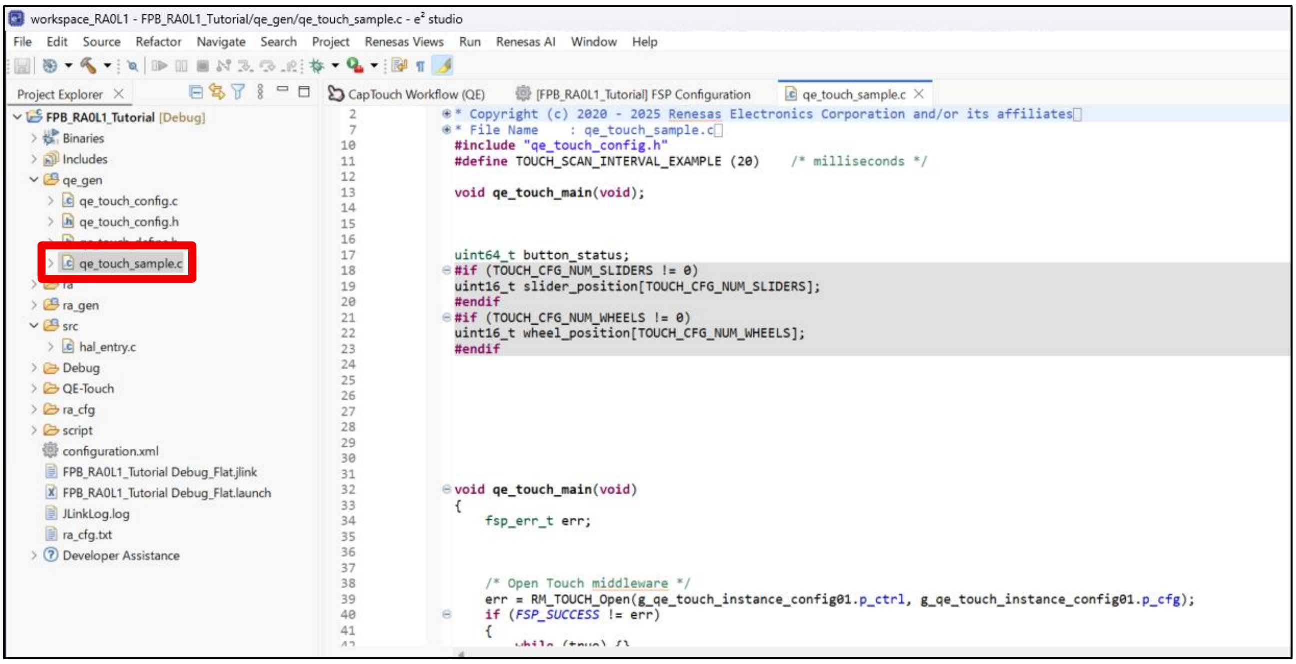


Figure5-14. Implement main program

Write the source code in void qe_touch_main(void) in qe_touch_sample.c.

The initial settings for the I/O ports are executed before the qe_touch_main function is called, so there is no need to write them.

The program creation is explained in the following steps:

(1) Perform touch scans every 20ms using the timer function and acquire touch information
(2) Control the lightning status of LED5 and LED6 based on touch detection status of the Touch Button.
(3) Control the start/stop of blinking of LED1 and LED2 based on touch detection of Touch Button1.
(4) Change the blinking cycle based on touch detection of Touch Button2.

5.2.1 Perform touch scans every 20ms using timer function and acquire touch information

(1) Implement the timer open function

When the “Developer Assistance” list in the “Project Explorer” window is opened, the timer module “MyTimer” configured in “5.1.1 Timer settings” will be displayed.

Expanding the list displays the function list.

The function to open the timer is R_TAU_Open() function. Drag and drop this function onto the source file using the mouse. Call this function before entering the main loop.

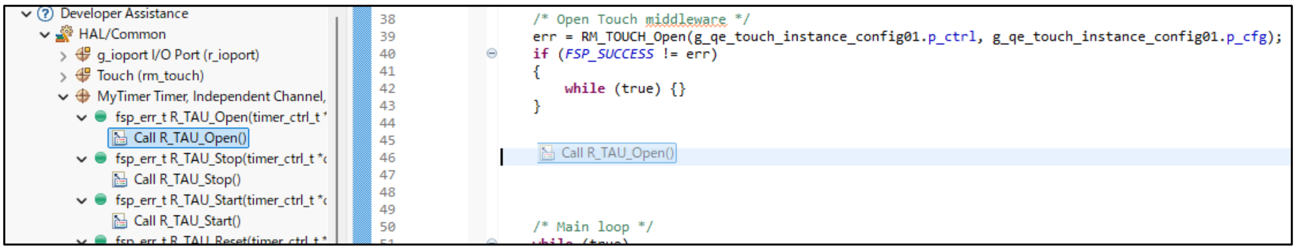


Figure5-15. Implement main program

Open function is added.

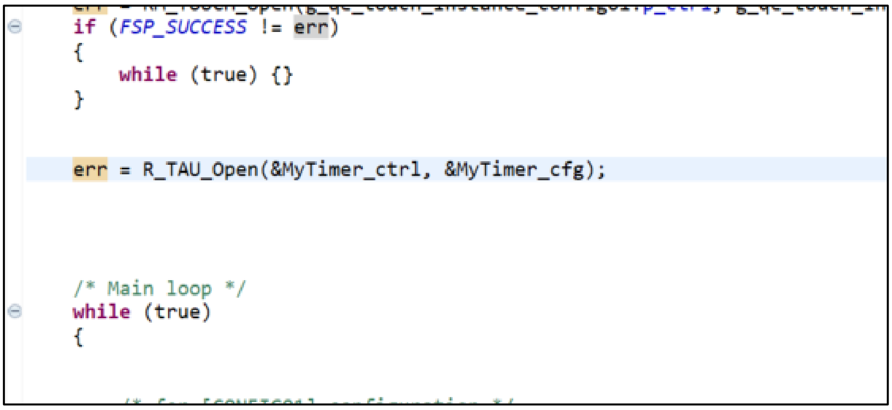


Figure5-16. Implement main program

(2) Implement of timer start function

The timer start function is R_TAU_Start() function. As before, drag and drop the source file onto the R_TAU_Open() function call.

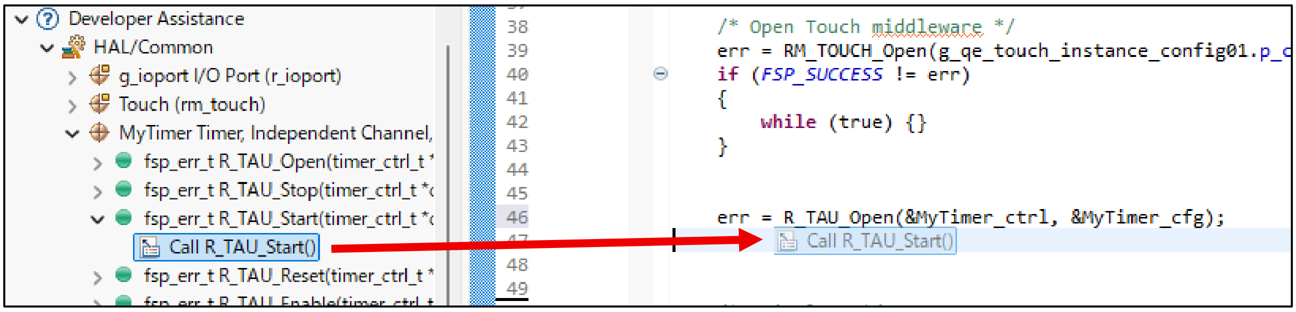


Figure5-17. Implement main program

The timer start function has been added.

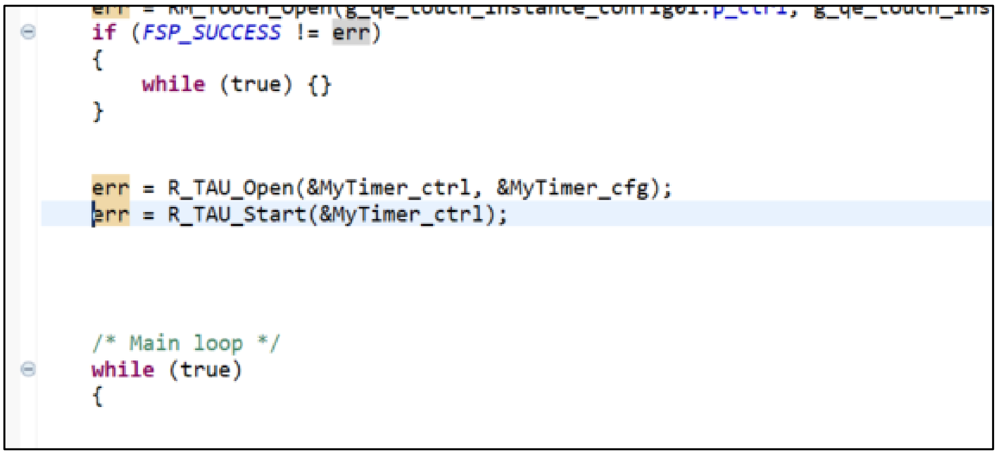


Figure5-18. Implement main program

(3) Verifying the return value

To detect cases where the open or start function returns an abnormal termination, implement a while(err); loop after the function call. If an abnormal termination occurs, this statement will cause an infinite loop.

The function's source code is as follows:

Figure5-19. Implement main program

(4) Implement the timer interrupt callback function

Write the callback function called from the timer interrupt. The file to be modified is ge_touch_sample.c.

Drag and drop the “Callback function definition” from the Timer Module in the Developer Assistance list to the end of the source file.

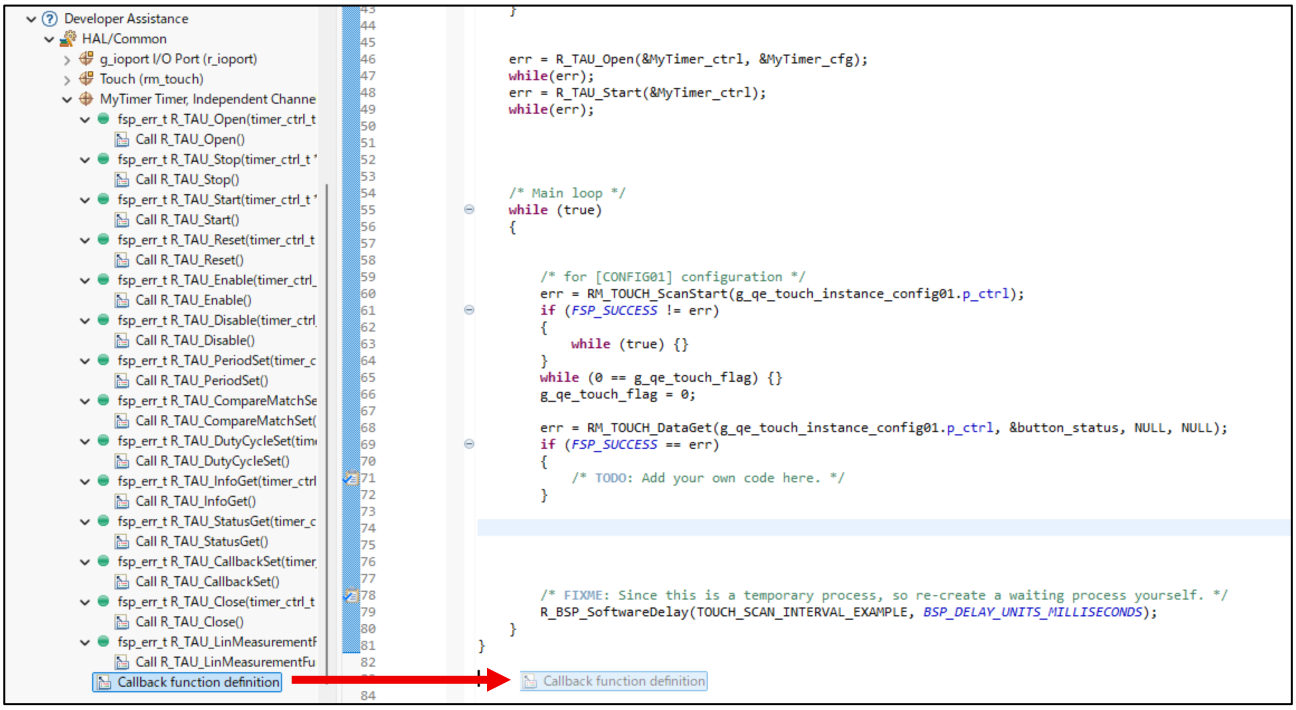


Figure5-20. Implement main program

Callback function has been added.

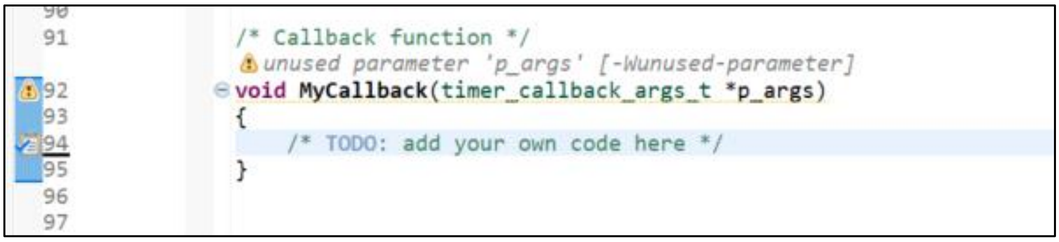


Figure5-21. Implement main program

Declare a variable for incrementing.

Declare volatile uint16_t g_touch_info_get_interval; outside the function.

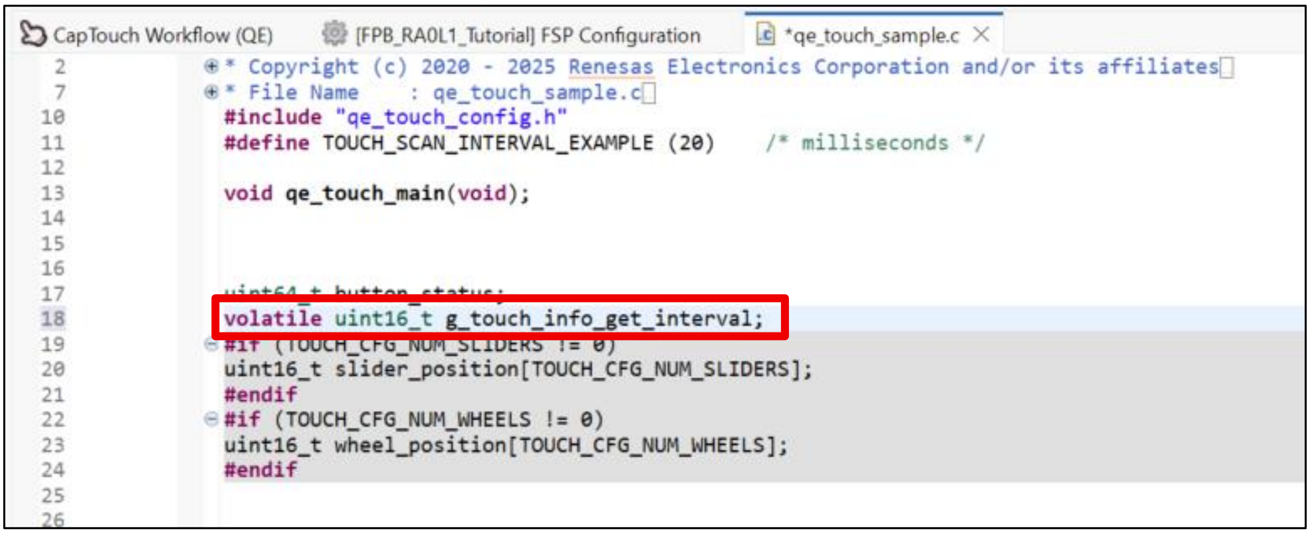


Figure5-22. Implement main program

Within the callback function, add code to increment the g_touch_info_get_interval variable. This will increment the value of this variable every 1 ms.

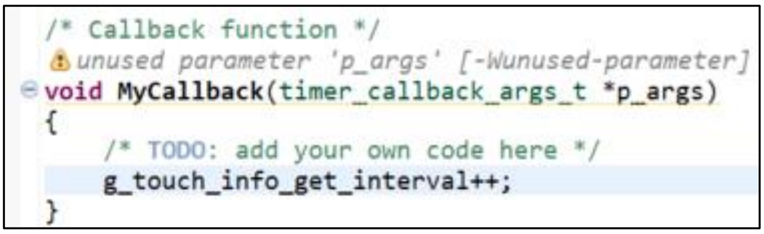


Figure5-23. Implement main program

(5) Adjusting the execution timing of touch functions

Add a conditional statement to perform a touch scan and read touch information every 20ms using the variables prepared in step (d).

Function to perform the touch scan is RM_TOUCH_ScanStart() function, and function to read touch information is RM_TOUCH_DataGet() function.

Add a conditional statement before the above two functions are called.

CapTouch Workflow (QE) [FPB_RA0L1_Tutorial] FSP Configuration qe_touch_sample.c

```c
    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }


    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);



    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS);
            g_touch_info_get_interval = 0;
        }

    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
}
```

Figure5-24. Implement main program

(6) Comment out the software delay function

The R_BSP_SoftwareDelay() function is no longer needed due to the implementation of step (e), and therefore it should be commented out.

Figure5-25. Implement main program

The program to acquire touch information by performing a touch scan and reading touch data every 20ms is now complete.

The source code up to this point is provided below in text format. The parts that have been added since qe_touch_sample.c was generated are shown in blue.

```
void qe_touch_main(void)
{
    fsp_err_t err;



    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
            }

            /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
}
```

5.2.2 Control the lightning status of LED5 and LED6 based on touch detection status of the Touch Button

When the RM_TOUCH_DataGet() function, which reads touch information, is executed, the information for TouchButton1 and 2 is stored in the button_status variable specified as the second argument.

For example, if you want to reference only the information for TouchButton1, you can do so by performing logical AND operation using the macro definition (highlighted in red) described in the project folder¥qe_gen¥qe_touch_define.h.

The names “BUTTON1” and “BUTTON2,” which are part of the macro definition names shown in the red box, correspond to the names set in “4.2 Tuning touch sensor by using QE Touch”.

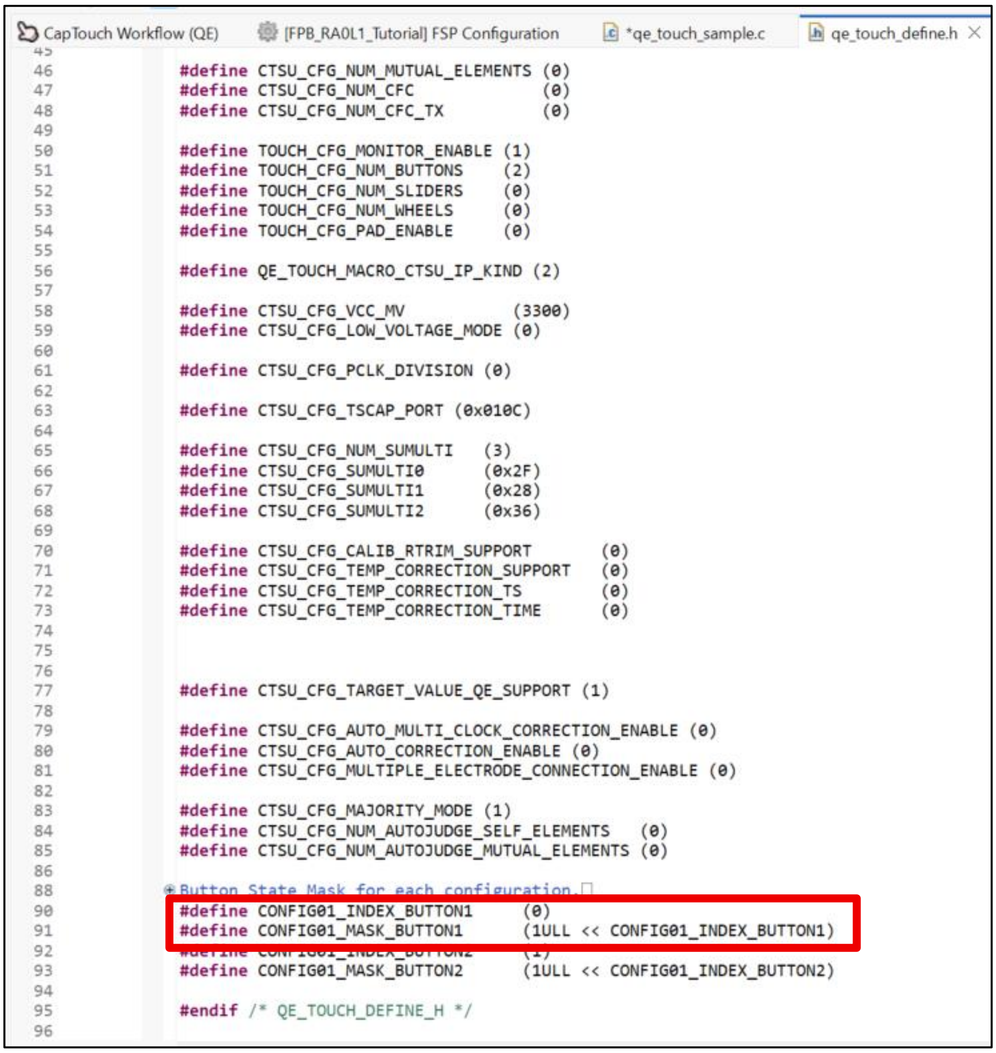


Figure5-26. Implement main program

(1) Program to light LED5 while detecting a touch on Touch Button 1

Add the conditional statement as shown below.

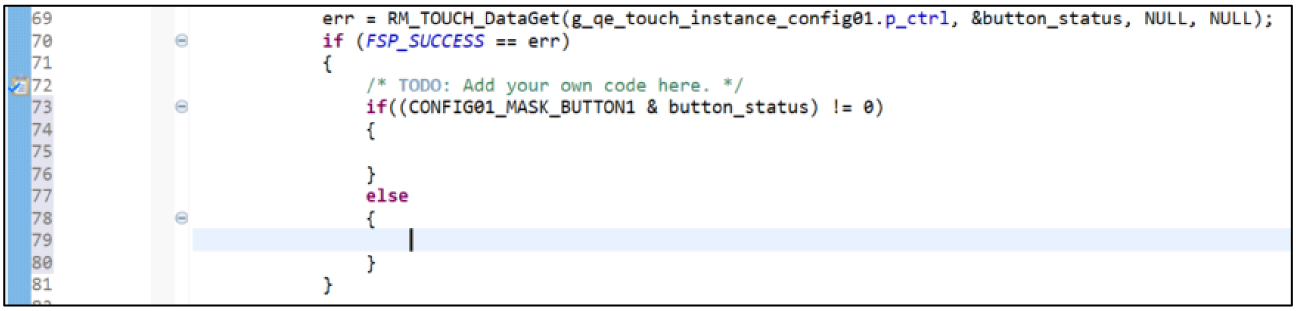


Figure5-27. Implement main program

Code a program to set the ON / OFF state of LED5 within a conditional statement.

Drag and drop the R_IOPORT_PinWrite() function from the Developer Assistance list into the conditional section of the source file.

Figure5-28. Implement main program

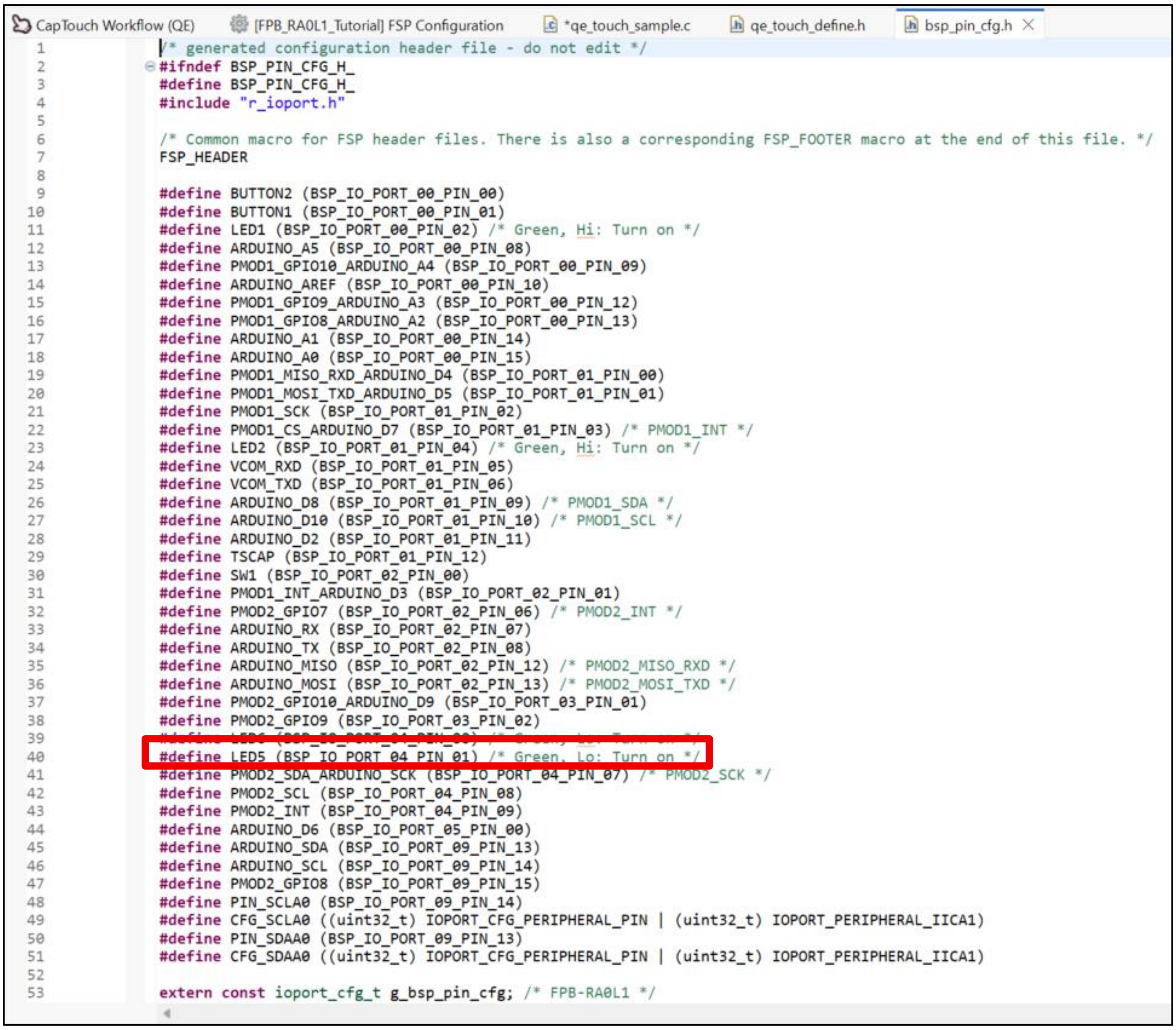
The R_IOPORT_PinWrite() function has been added.

Figure5-29. Implement main program

Modify the arguments of the R_IOPORT_PinWrite() function to set the LED ON/OFF.

To configure LED5, the macro definition is specified in the red box below within ra_cfg¥fsp_cfg¥bsp_pin_cfg.h in the Project Explorer.

Use this macro definition as an argument for the R_IOPORT_PinWrite() function to set LED5 ON/OFF.




```
/* generated configuration header file - do not edit */
#ifndef BSP_PIN_CFG_H_
#define BSP_PIN_CFG_H_
#include "r_ioport.h"

/* Common macro for FSP header files. There is also a corresponding FSP_FOOTER macro at the end of this file. */
FSP_HEADER

#define BUTTON2 (BSP_IO_PORT_00_PIN_00)
#define BUTTON1 (BSP_IO_PORT_00_PIN_01)
#define LED1 (BSP_IO_PORT_00_PIN_02) /* Green, Hi: Turn on */
#define ARDUINO_A5 (BSP_IO_PORT_00_PIN_08)
#define PMOD1_GPIO10_ARDUINO_A4 (BSP_IO_PORT_00_PIN_09)
#define ARDUINO_AREF (BSP_IO_PORT_00_PIN_10)
#define PMOD1_GPIO9_ARDUINO_A3 (BSP_IO_PORT_00_PIN_12)
#define PMOD1_GPIO8_ARDUINO_A2 (BSP_IO_PORT_00_PIN_13)
#define ARDUINO_A1 (BSP_IO_PORT_00_PIN_14)
#define ARDUINO_A0 (BSP_IO_PORT_00_PIN_15)
#define PMOD1_MISO_RXD_ARDUINO_D4 (BSP_IO_PORT_01_PIN_00)
#define PMOD1_MOSI_TXD_ARDUINO_D5 (BSP_IO_PORT_01_PIN_01)
#define PMOD1_SCK (BSP_IO_PORT_01_PIN_02)
#define PMOD1_CS_ARDUINO_D7 (BSP_IO_PORT_01_PIN_03) /* PMOD1_INT */
#define LED2 (BSP_IO_PORT_01_PIN_04) /* Green, Hi: Turn on */
#define VCOM_RXD (BSP_IO_PORT_01_PIN_05)
#define VCOM_TXD (BSP_IO_PORT_01_PIN_06)
#define ARDUINO_D8 (BSP_IO_PORT_01_PIN_09) /* PMOD1_SDA */
#define ARDUINO_D10 (BSP_IO_PORT_01_PIN_10) /* PMOD1_SCL */
#define ARDUINO_D2 (BSP_IO_PORT_01_PIN_11)
#define TSCAP (BSP_IO_PORT_01_PIN_12)
#define SW1 (BSP_IO_PORT_02_PIN_00)
#define PMOD1_INT_ARDUINO_D3 (BSP_IO_PORT_02_PIN_01)
#define PMOD2_GPIO7 (BSP_IO_PORT_02_PIN_06) /* PMOD2_INT */
#define ARDUINO_RX (BSP_IO_PORT_02_PIN_07)
#define ARDUINO_TX (BSP_IO_PORT_02_PIN_08)
#define ARDUINO_MISO (BSP_IO_PORT_02_PIN_12) /* PMOD2_MISO_RXD */
#define ARDUINO_MOSI (BSP_IO_PORT_02_PIN_13) /* PMOD2_MOSI_TXD */
#define PMOD2_GPIO10_ARDUINO_D9 (BSP_IO_PORT_03_PIN_01)
#define PMOD2_GPIO9 (BSP_IO_PORT_03_PIN_02)
#define LED6 (BSP_IO_PORT_04_PIN_00) /* Green, Lo: Turn on */
#define LED5 (BSP_IO_PORT_04_PIN_01) /* Green, Lo: Turn on */
#define PMOD2_SDA_ARDUINO_SCK (BSP_IO_PORT_04_PIN_07) /* PMOD2_SCK */
#define PMOD2_SCL (BSP_IO_PORT_04_PIN_08)
#define PMOD2_INT (BSP_IO_PORT_04_PIN_09)
#define ARDUINO_D6 (BSP_IO_PORT_05_PIN_00)
#define ARDUINO_SDA (BSP_IO_PORT_09_PIN_13)
#define ARDUINO_SCL (BSP_IO_PORT_09_PIN_14)
#define PMOD2_GPIO8 (BSP_IO_PORT_09_PIN_15)
#define PIN_SCLA0 (BSP_IO_PORT_09_PIN_14)
#define CFG_SCLA0 ((uint32_t) IOPORT_CFG_PERIPHERAL_PIN | (uint32_t) IOPORT_PERIPHERAL_IICA1)
#define PIN_SDAA0 (BSP_IO_PORT_09_PIN_13)
#define CFG_SDAA0 ((uint32_t) IOPORT_CFG_PERIPHERAL_PIN | (uint32_t) IOPORT_PERIPHERAL_IICA1)

extern const ioport_cfg_t g_bsp_pin_cfg; /* FPB-RA0L1 */
```

Figure5-30. Implement main program

The second argument of the R_IOPORT_PinWrite() function specifies the pin number to configure = “LED5”, and the third argument specifies the output content.

To configure LED5 to light up while Touch Button 1 is detected as touched, edit the source code as follows:

CapTouch Workflow (QE) | [FPB_RA0L1_Tutorial] FSP Configuration | *qe_touch_sample.c | qe_touch_define.h | bsp_pin_cfg.h

```c
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);



    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);

                }
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }

    }
}
```

Figure5-31. Implement main program

(2) Program to light LED6 while a touch is detected on Touch Button 2

Implement control of LED6 using the procedure described above.

Add the conditional statement as shown below.

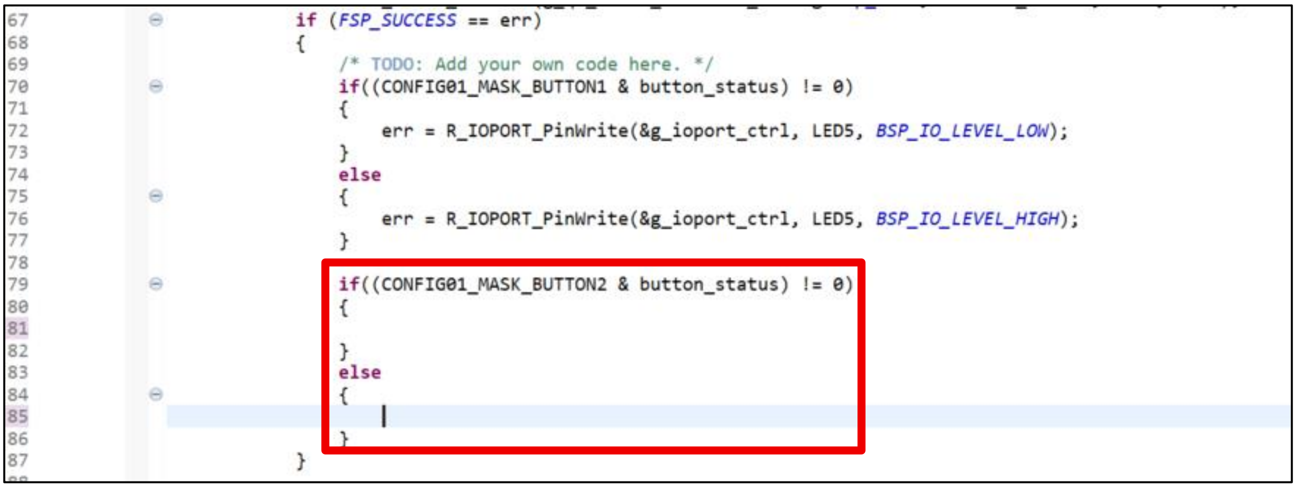


Figure5-32. Implement main program

Drag and drop the R_IOPORT_PinWrite() function from the Developer Assistance list into the conditional section of the source file.

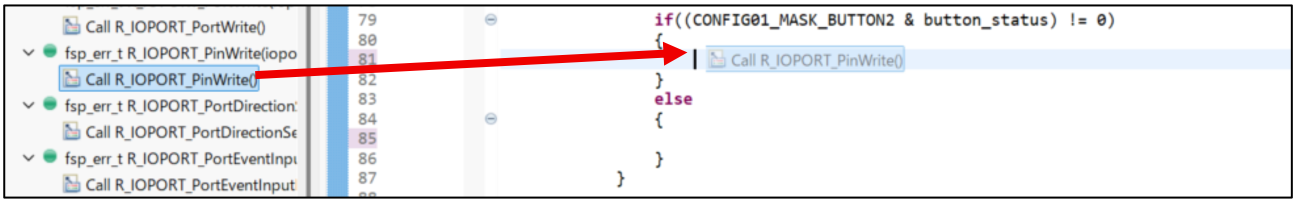


Figure5-33. Implement main program

R_IOPORT_PinWrite() function has been added.

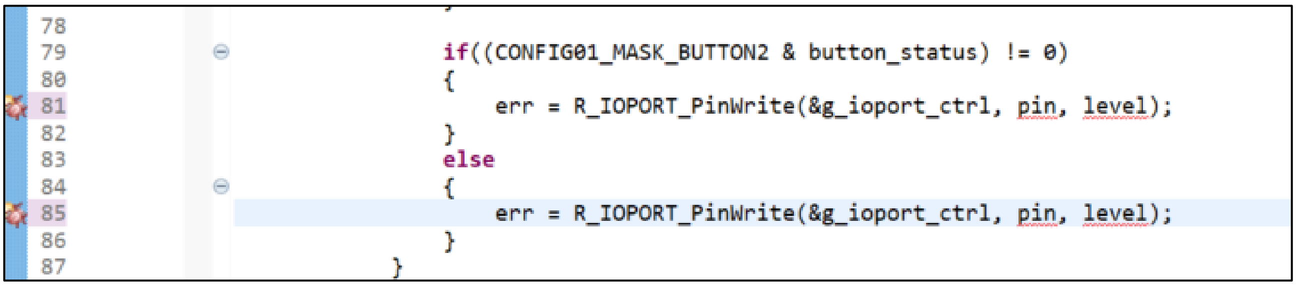


Figure5-34. Implement main program

As with LED5, the following macro definition written in bsp_pin_cfg.h can be used as an argument to the R_IOPORT_PinWrite() function to set LED6 ON/OFF.

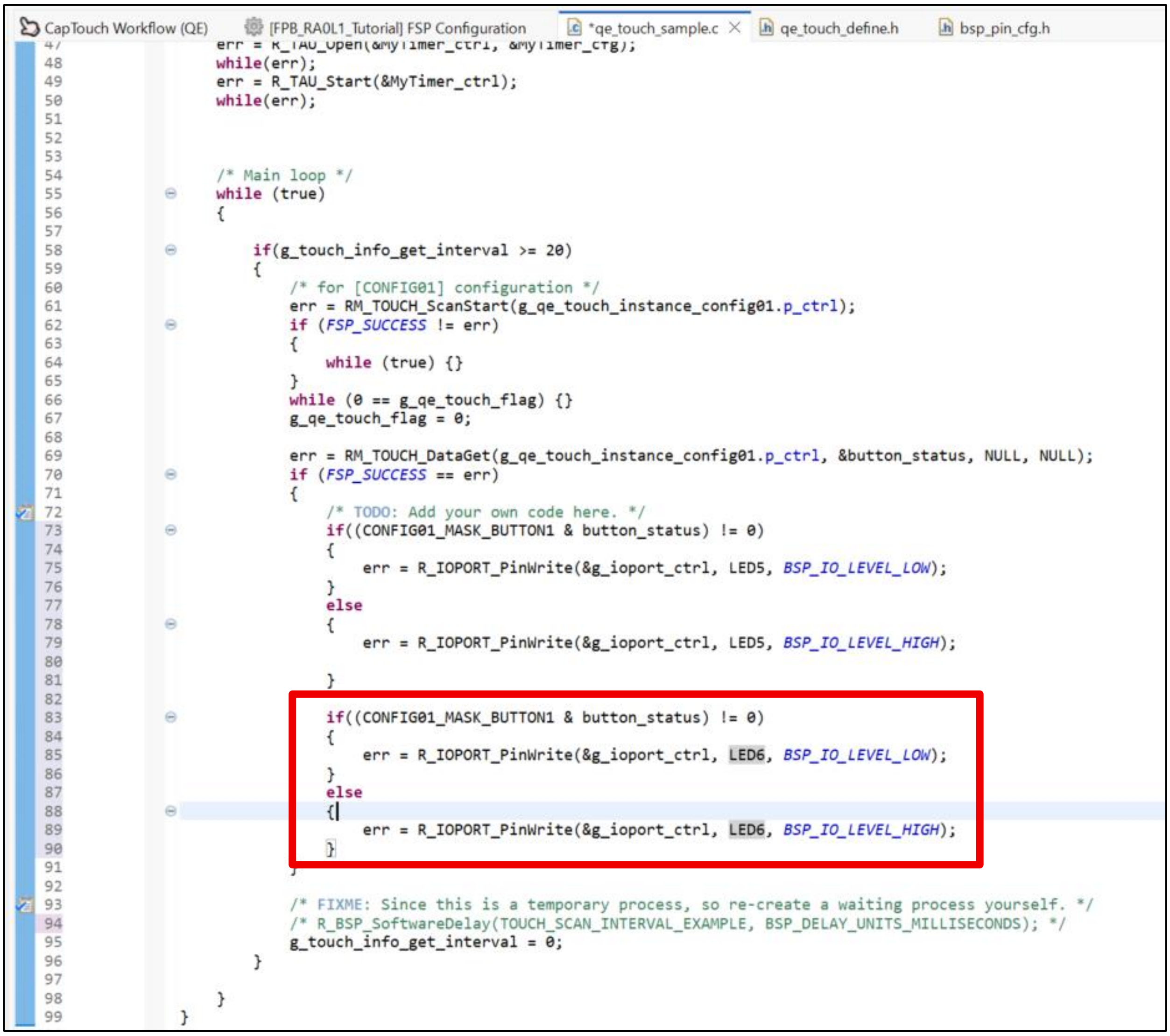
CapTouchワークフロー (QE) | [FPB_RA0L1_Tutorial] FSP Configuration | r_ctsu.c | main.c | *qe_touch_sample.c | qe_touch_define.h | r_ioport_cfg.h | bsp_pin_cfg.h

```c
/* generated configuration header file - do not edit */
#ifndef BSP_PIN_CFG_H_
#define BSP_PIN_CFG_H_
#include "r_ioport.h"

/* Common macro for FSP header files. There is also a corresponding FSP_FOOTER macro at the end of this file. */
FSP_HEADER

#define BUTTON2 (BSP_IO_PORT_00_PIN_00)
#define BUTTON1 (BSP_IO_PORT_00_PIN_01)
#define LED1 (BSP_IO_PORT_00_PIN_02) /* Green, Hi: Turn on */
#define ARDUINO_A5 (BSP_IO_PORT_00_PIN_08)
#define PMOD1_GPIO10_ARDUINO_A4 (BSP_IO_PORT_00_PIN_09)
#define ARDUINO_AREF (BSP_IO_PORT_00_PIN_10)
#define PMOD1_GPIO9_ARDUINO_A3 (BSP_IO_PORT_00_PIN_12)
#define PMOD1_GPIO8_ARDUINO_A2 (BSP_IO_PORT_00_PIN_13)
#define ARDUINO_A1 (BSP_IO_PORT_00_PIN_14)
#define ARDUINO_A0 (BSP_IO_PORT_00_PIN_15)
#define PMOD1_MISO_RXD_ARDUINO_D4 (BSP_IO_PORT_01_PIN_00)
#define PMOD1_MOSI_TXD_ARDUINO_D5 (BSP_IO_PORT_01_PIN_01)
#define PMOD1_SCK (BSP_IO_PORT_01_PIN_02)
#define PMOD1_CS_ARDUINO_D7 (BSP_IO_PORT_01_PIN_03) /* PMOD1_INT */
#define LED2 (BSP_IO_PORT_01_PIN_04) /* Green, Hi: Turn on */
#define VCOM_RXD (BSP_IO_PORT_01_PIN_05)
#define VCOM_TXD (BSP_IO_PORT_01_PIN_06)
#define ARDUINO_D8 (BSP_IO_PORT_01_PIN_09) /* PMOD1_SDA */
#define ARDUINO_D10 (BSP_IO_PORT_01_PIN_10) /* PMOD1_SCL */
#define ARDUINO_D2 (BSP_IO_PORT_01_PIN_11)
#define TSCAP (BSP_IO_PORT_01_PIN_12)
#define SW1 (BSP_IO_PORT_02_PIN_00)
#define PMOD1_INT_ARDUINO_D3 (BSP_IO_PORT_02_PIN_01)
#define PMOD2_GPIO7 (BSP_IO_PORT_02_PIN_06) /* PMOD2_INT */
#define ARDUINO_RX (BSP_IO_PORT_02_PIN_07)
#define ARDUINO_TX (BSP_IO_PORT_02_PIN_08)
#define ARDUINO_MISO (BSP_IO_PORT_02_PIN_12) /* PMOD2_MISO_RXD */
#define ARDUINO_MOSI (BSP_IO_PORT_02_PIN_13) /* PMOD2_MOSI_TXD */
#define PMOD2_GPIO10_ARDUINO_D9 (BSP_IO_PORT_03_PIN_01)
[illegible]
#define LED6 (BSP_IO_PORT_04_PIN_00) /* Green, Lo: Turn on */
[illegible]
#define PMOD2_SDA_ARDUINO_SCK (BSP_IO_PORT_04_PIN_07) /* PMOD2_SCK */
#define PMOD2_SCL (BSP_IO_PORT_04_PIN_08)
#define PMOD2_INT (BSP_IO_PORT_04_PIN_09)
#define ARDUINO_D6 (BSP_IO_PORT_05_PIN_00)
#define ARDUINO_SDA (BSP_IO_PORT_09_PIN_13)
#define ARDUINO_SCL (BSP_IO_PORT_09_PIN_14)
#define PMOD2_GPIO8 (BSP_IO_PORT_09_PIN_15)
#define PIN_SCLA0 (BSP_IO_PORT_09_PIN_14)
#define CFG_SCLA0 ((uint32_t) IOPORT_CFG_PERIPHERAL_PIN | (uint32_t) IOPORT_PERIPHERAL_IICA1)
#define PIN_SDAA0 (BSP_IO_PORT_09_PIN_13)
#define CFG_SDAA0 ((uint32_t) IOPORT_CFG_PERIPHERAL_PIN | (uint32_t) IOPORT_PERIPHERAL_IICA1)
```

Figure5-35. Implement main program

The second argument of the R_IOPORT_PinWrite() function specifies the pin number to configure = “LED5”, and the third argument specifies the output content.

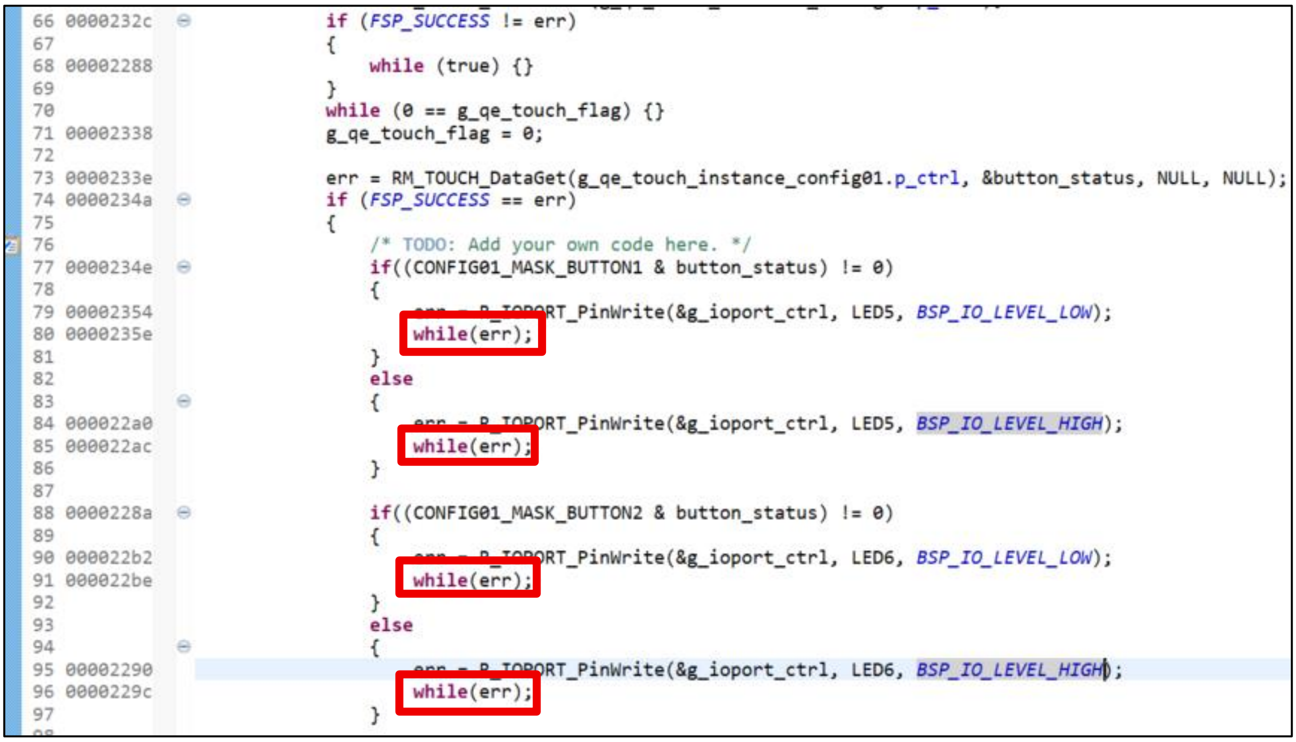
To configure LED6 to light up while Touch Button 2 is detected as touched, edit the source code as follows.




```c
    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);



    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);

                }

                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
                }
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }

    }
}
```

Figure5-36. Implement main program

To detect cases where the R_IOPORT_PInWrite() function returns an abnormal termination, implement a while(err); loop after the function call.



```
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
                    while(err);
                }
```

Figure5-37. Implement main program

The source code up to this point is provided below as text. The parts that have been added since qe_touch_sample.c was generated are shown in blue.

```
void qe_touch_main(void)
{
    fsp_err_t err;


    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                }
```

```
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                                 BSP_IO_LEVEL_HIGH);
                    while(err);
                }
            }

            /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
}
```

5.2.3 Control the start/stop of blinking of LED1 and LED2 based on touch detection of Touch Button1

Detect the touch detection timing of Touch Button1 and control the start/stop of the blinking operation of LED1 and LED2. At this stage, the blinking cycle is fixed at 2 seconds.

(1) Set Initial values for LED1 and LED2

Set the initial values for LED1 and LED2 so that LED1 is ON and LED2 is OFF.

Since these value are set before the main loop starts, add the code before the "while (true)" loop.

Drag and drop the R_IOPORT_PinWrite() function from the Developer Assistance list into the source file.

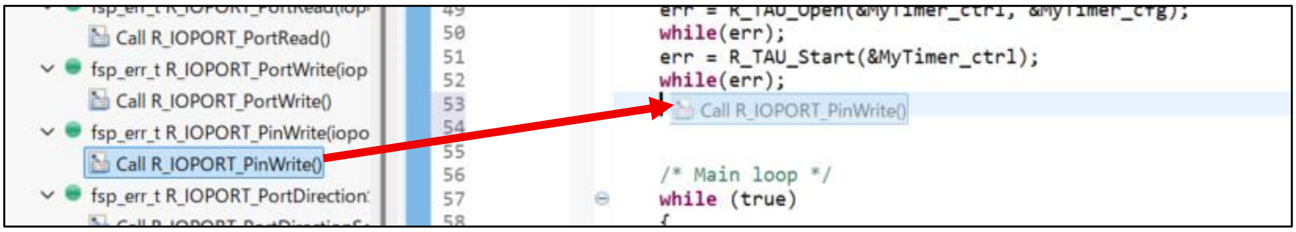


Figure5-38. Implement main program

R_IOPORT_PinWrite() function has been added.

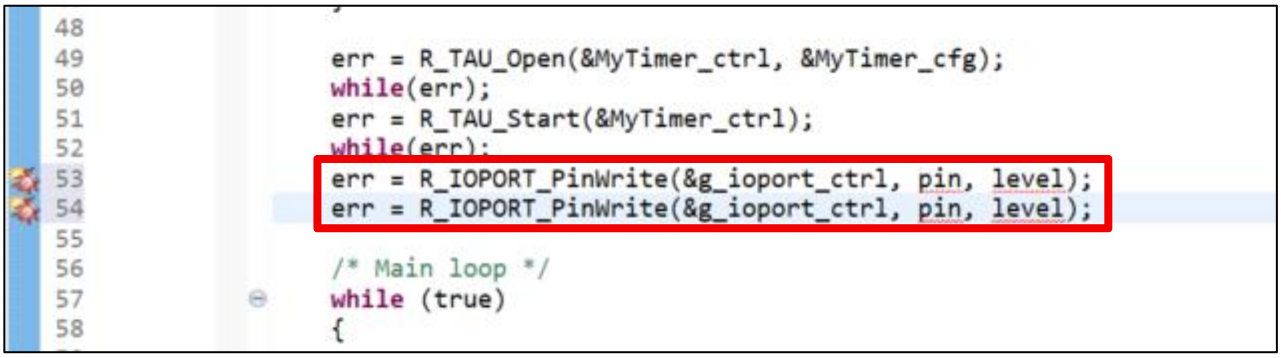


Figure5-39. Implement main program

The following macro definitions written in bsp_pin_cfg.h can be used as arguments to the R_IOPORT_PinWrite() function to set LED1 and LED2 to ON/OFF.

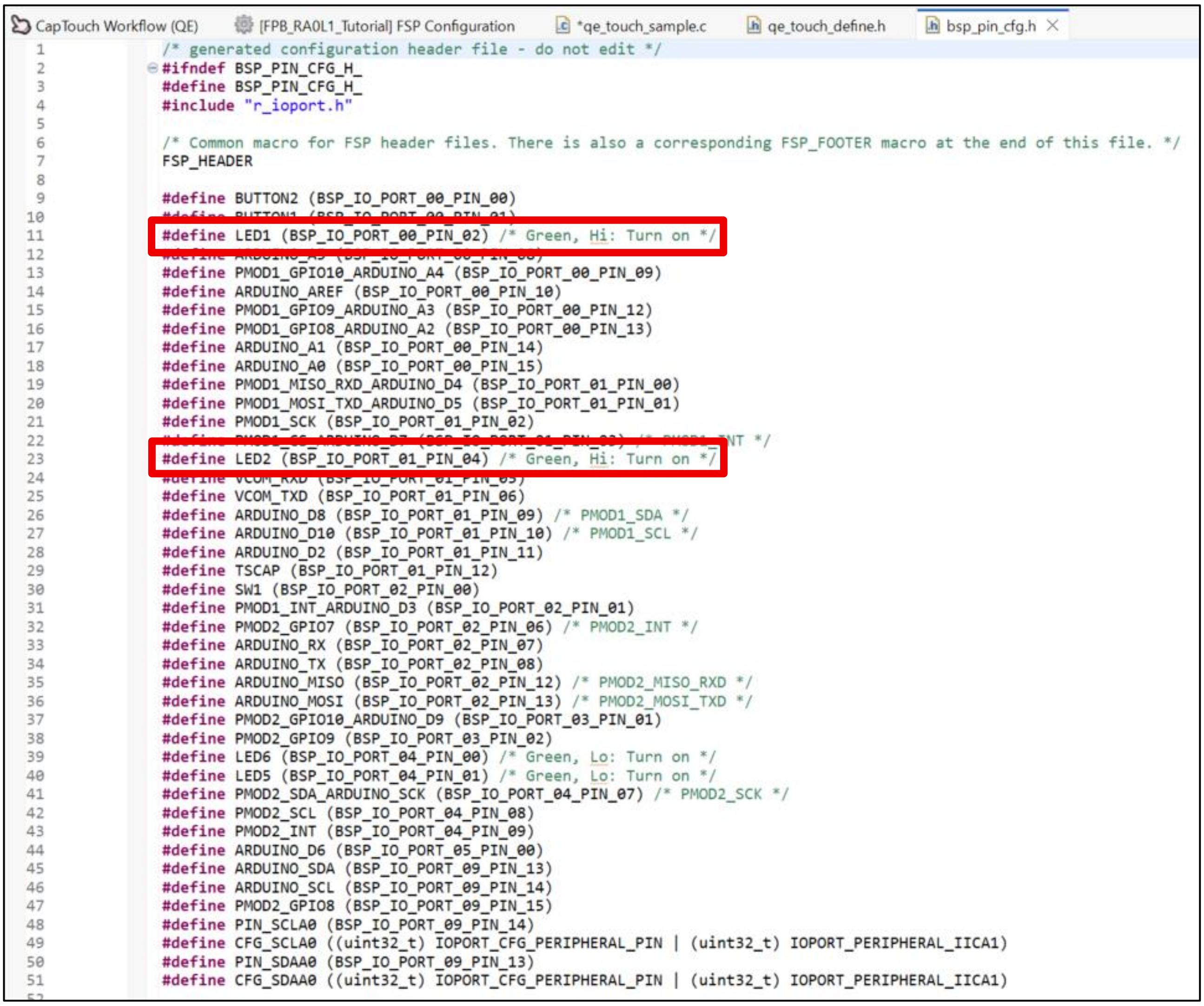


Figure5-40. Implement main program

The second argument of the R_IOPORT_PinWrite() function specifies the pin number to configure = "LED1" and "LED2". The third argument specifies the output content.

Edit the initial setting process for LED1 and LED2 as follows.

```
void qe_touch_main(void)
{
    fsp_err_t err;


    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }


    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);


    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
```

Figure5-41. Implement main program

To detect cases where an abnormal termination is returned, implement a "while(err);" loop after the function call.

```
void qe_touch_main(void)
{
    fsp_err_t err;


    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }


    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
    while(err);


    /* Main loop */
    while (true)
    {
```

Figure5-42. Implement main program

(2) Store the previous Touch Button information

Touch Button information is obtained every 20ms and stored in button_status, but in order to detect the timing of touch detection, a new variable is created to store the previous Touch Button information.

uint64_t g_button_status_previous;

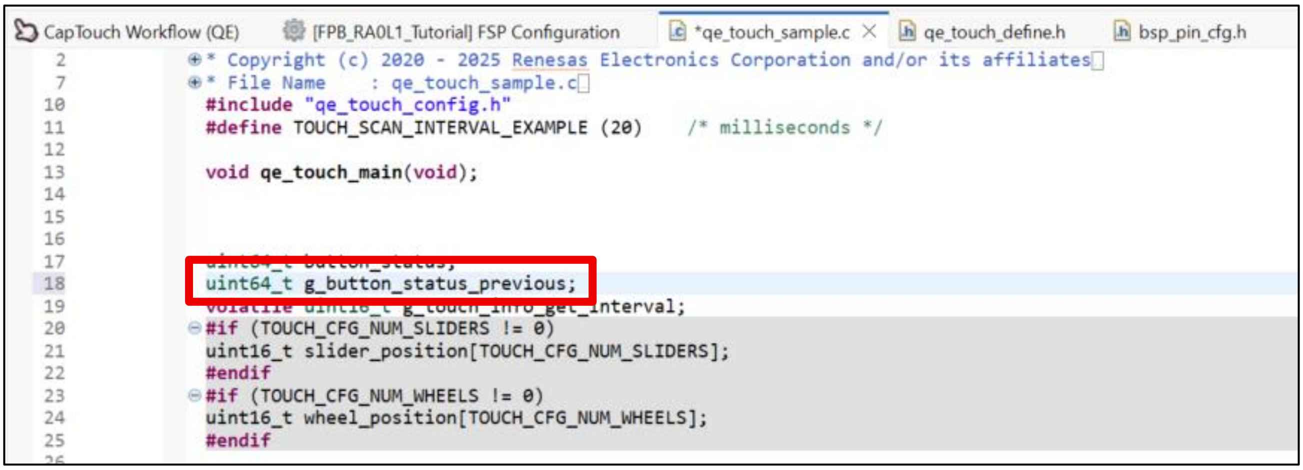


Figure5-43. Implement main program

Placing “g_button_status_previous = button_status;” at the end of the “if (FSP_SUCCESS == err)” condition allows the previous value to be continuously updated every 20ms.

CapTouch Workflow (QE) [FPB_RA0L1_Tutorial] FSP Configuration *qe_touch_sample.c qe_touch_define.h bsp_pin_cfg.h

```
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
                    while(err);
                }


                g_button_status_previous = button_status;
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }

    }
}
```

Figure5-44. Implement main program

(3) Implement the Touch Detection Program for Touch Button1

Create a conditional statement to detect when Touch Button1 is touched.

The touch detection condition is the logical AND of the following two conditions.

- Was Touch Button1 touched this time?
  ((CONFIG01_MASK_BUTTON1 & button_status) != 0)
- Was Touch Button1 touched last time?
  ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0)

```
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
                {

                }
                g_button_status_previous = button_status;
            }
            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }
    }
}
```

Figure5-45. Implement main program

(4) Implement increment processing when Touch Button1 is touched

Create a variable that increments when Touch Button1 is pressed and incorporate it into the conditional statement.

Declare “uint8_t g_button1_push_count;”.

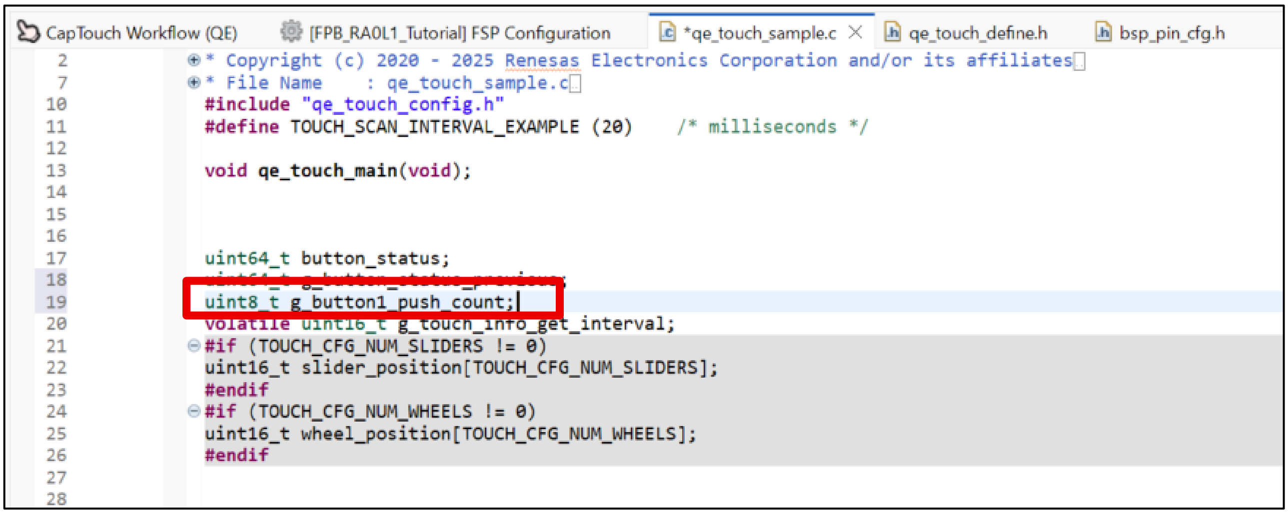


Figure5-46. Implement main program

Increment the value when Touch Button1 is touched.

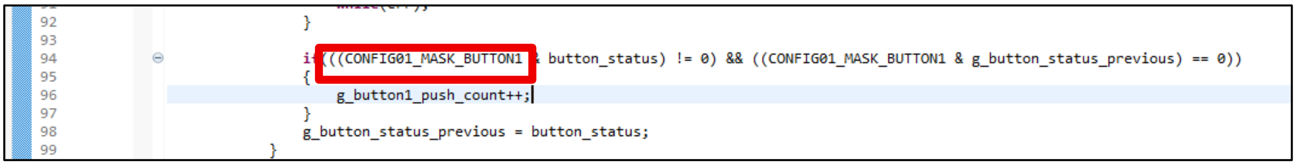


Figure5-47. Implement main program

(5) Clear the count variable when Touch Button1 is touched again

Create a conditional statement to initialize the g_button1_push_count variable when touch is detected at Touch Button 1 again.

This is the program that starts/stops the blinking action. A variable value of 1 indicates the blinking start state.

```
        if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
        {
            g_button1_push_count++;
            if(g_button1_push_count > 1)
            {
                g_button1_push_count = 0;
            }
        }
        g_button_status_previous = button_status;
    }
```

Figure5-48. Implement main program

(6) Add conditional statement for the LED blinking start state

Using the variable implemented in step (d), add a conditional statement to control the blinking start state.

```
        if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
        {
            g_button1_push_count++;
            if(g_button1_push_count > 1)
            {
                g_button1_push_count = 0;
            }
        }

        if(g_button1_push_count == 1)
        {

        }
```

Figure5-49. Implement main program

(7) Add and implement variables to control LED blinking cycle

Create a new variable to control the LED blinking cycle of 2 seconds and incorporate it into a conditional statement.

volatile uint16_t g_led_blink_interval;

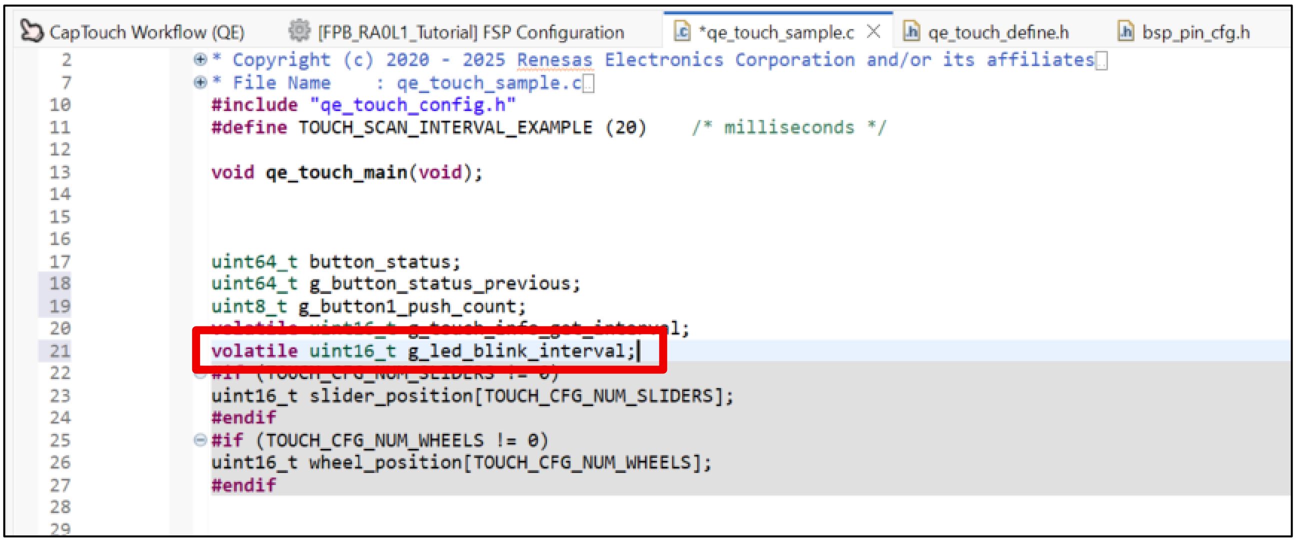


Figure5-50. Implement main program

Increment the g_led_blink_interval variable within MyCallback() function.

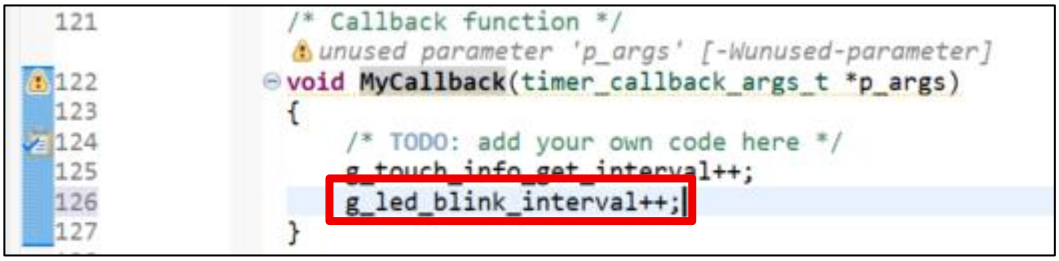


Figure5-51. Implement main program

Create a condition in step (e) that determines whether 2 seconds have passed using the g_led_blink_interval variable.

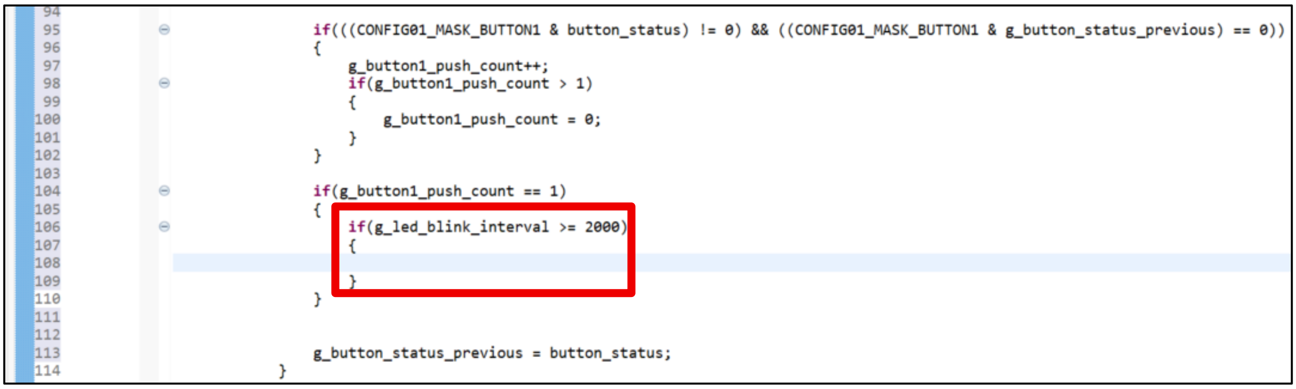


Figure5-52. Implement main program

When the condition is met after 2 seconds has elapsed, the g_led_blink_interval variable is initialized.

```
if(g_button1_push_count == 1)
{
    if(g_led_blink_interval >= 2000)
    {
        g_led_blink_interval = 0;
    }
}
```

Figure5-53. Implement main program

Additionally, the g_led_blink_interval variable is initialized when Touch Button1 is touched.

```
if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
{
    g_button1_push_count++;
    g_led_blink_interval = 0;
    if(g_button1_push_count > 1)
    {
        g_button1_push_count = 0;
    }
}
```

Figure5-54. Implement main program

(8) Create a program to switch the lighting state of LED1 and LED2

Drag and drop the R_IOPORT_PinRead() function from the Developer Assistance list onto the source file.

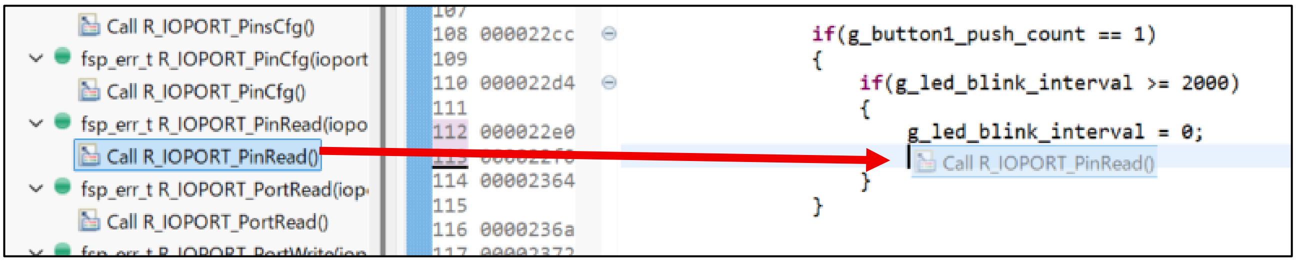


Figure5-55. Implement main program

R_IOPORT_PinRead() function has been added.

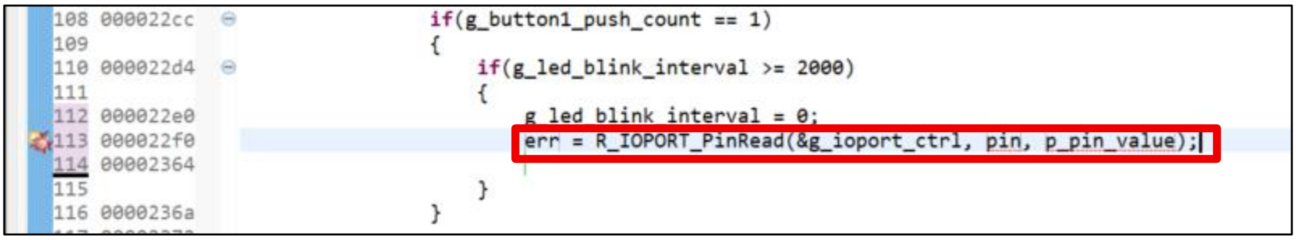


Figure5-56. Implement main program

The R_IOPORT_PinRead() function stores the information for the pin specified by the second argument into the third argument. Create a variable to assign to the third argument.

bsp_io_level_t value;

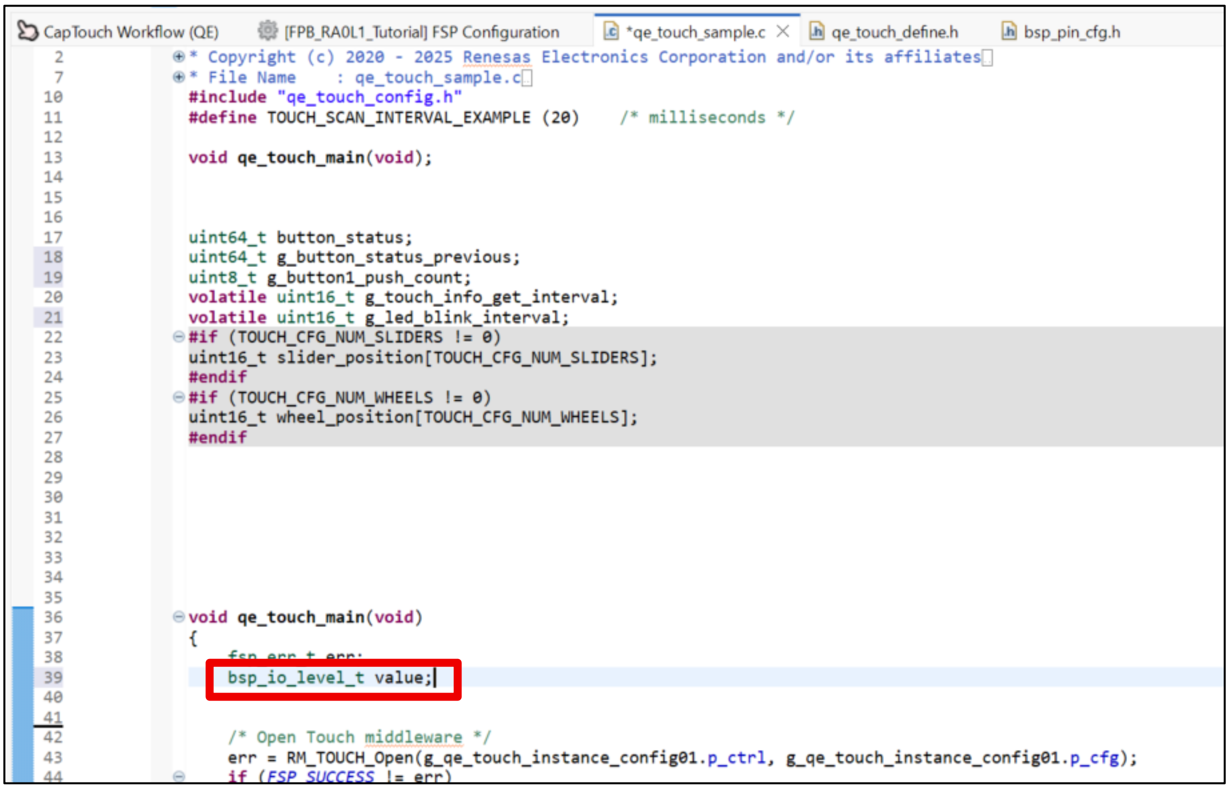
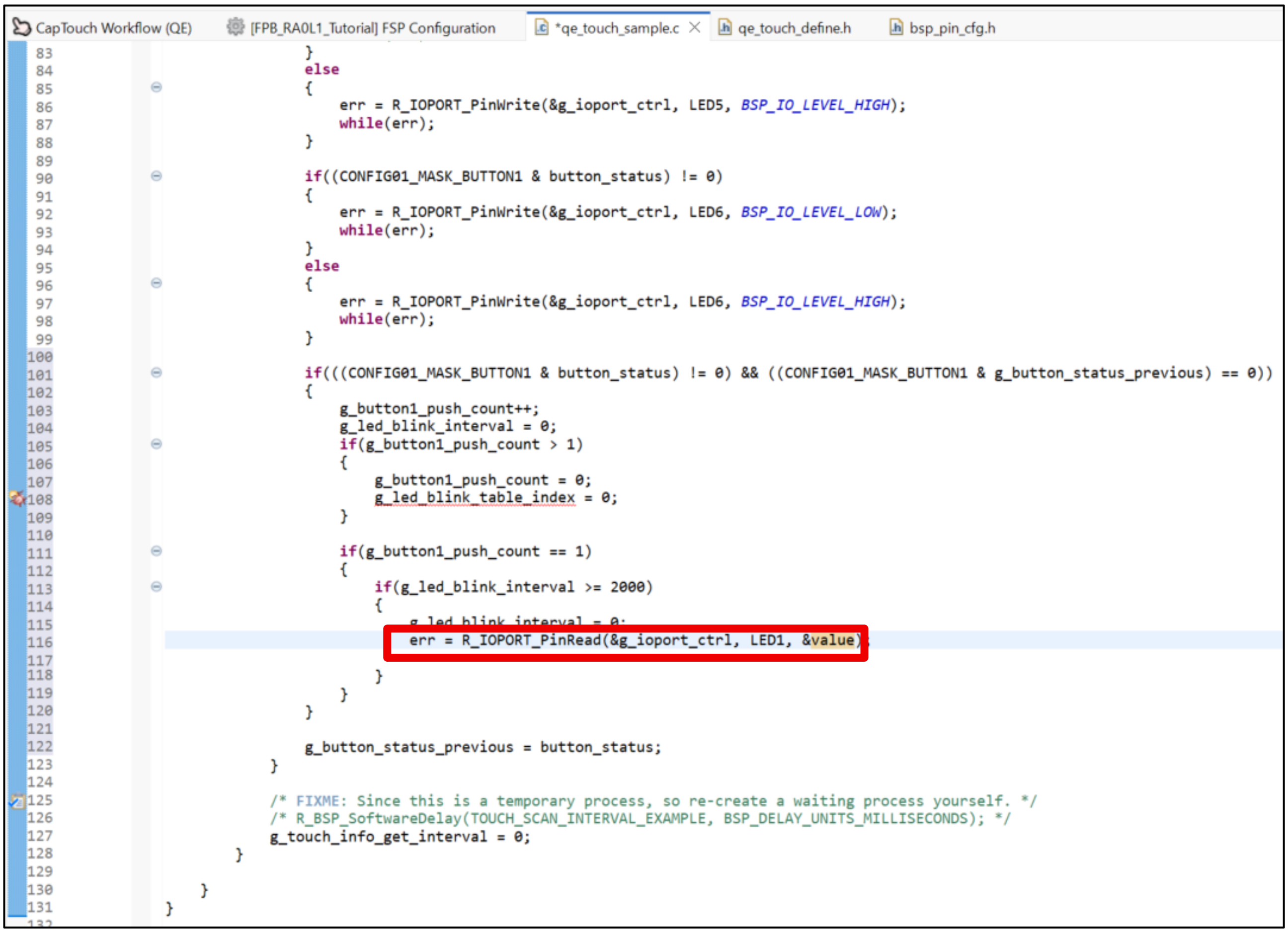


Figure5-57. Implement main program

To obtain the status of LED1, edit the code as follows.




```
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
                {
                    g_button1_push_count++;
                    g_led_blink_interval = 0;
                    if(g_button1_push_count > 1)
                    {
                        g_button1_push_count = 0;
                        g_led_blink_table_index = 0;
                    }

                    if(g_button1_push_count == 1)
                    {
                        if(g_led_blink_interval >= 2000)
                        {
                            g_led_blink_interval = 0;
                            err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);

                        }
                    }
                }

                g_button_status_previous = button_status;
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }

    }
}
```

Figure5-58. Implement main program

To detect cases where an abnormal termination is returned, implement a “while(err);” loop after the function call.

```
if(g_button1_push_count == 1)
{
    if(g_led_blink_interval >= 2000)
    {
        g_led_blink_interval = 0;
        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
        while(err);
    }
}
```

Figure5-59. Implement main program

Create a program that uses the third argument of the R_IOPORT_PinRead() function to switch the blinking of LED1 and LED2.

The conditional statement is as follows.

```
if(g_button1_push_count == 1)
{
    if(g_led_blink_interval >= 2000)
    {
        g_led_blink_interval = 0;
        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
        while(err);
        if(value == BSP_IO_LEVEL_HIGH)
        {

        }
        else
        {

        }
    }
}
```

Figure5-60. Implement main program

Determine the output state of LED1 and set the outputs of LED1 and LED2 as follows.

- If LED1 is HIGH, sets LED1 to LOW and LED2 to HIGH.
- If LED1 is LOW, set LED1 to HIGH and LED2 to LOW.

```
            if(g_button1_push_count == 1)
            {
                if(g_led_blink_interval >= 2000)
                {
                    g_led_blink_interval = 0;
                    err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                    while(err);
                    if(value == BSP_IO_LEVEL_HIGH)
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
                        while(err);
                    }
                    else
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
                        while(err);
                    }
                }
            }
```

Figure5-61. Implement main program

The source code up to this point is provided as text. The parts that have been added since qe_touch_sample.c was generated are shown in blue.

```
void qe_touch_main(void);

uint64_t button_status;
uint64_t g_button_status_previous;
uint8_t g_button1_push_count;
volatile uint16_t g_touch_info_get_interval;
volatile uint16_t g_led_blink_interval;
#if (TOUCH_CFG_NUM_SLIDERS != 0)
uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
#endif
#if (TOUCH_CFG_NUM_WHEELS != 0)
uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
#endif

void qe_touch_main(void)
{
    fsp_err_t err;
    bsp_io_level_t value;

    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;
            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
```

```
BSP_IO_LEVEL_LOW);
                while(err);
            }
            else
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
BSP_IO_LEVEL_HIGH);
                while(err);
            }

            if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
BSP_IO_LEVEL_LOW);
                while(err);
            }
            else
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
BSP_IO_LEVEL_HIGH);
                while(err);
            }

            if(((CONFIG01_MASK_BUTTON1 & button_status) != 0)
&& ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
            {
                g_button1_push_count++;
                g_led_blink_interval = 0;
                if(g_button1_push_count > 1)
                {
                    g_button1_push_count = 0;
                }
            }

            if(g_button1_push_count == 1)
            {
                if(g_led_blink_interval >= 2000)
                {
                    g_led_blink_interval = 0;
                    err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                    while(err);
                    if(value == BSP_IO_LEVEL_HIGH)
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
BSP_IO_LEVEL_LOW);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
BSP_IO_LEVEL_HIGH);
                        while(err);
                    }
                    else
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
BSP_IO_LEVEL_HIGH);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
BSP_IO_LEVEL_LOW);
                        while(err);
```

```
                    }
                }
            }
            g_button_status_previous = button_status;
        }
        /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
        /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
        g_touch_info_get_interval = 0;
    }
  }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
    g_led_blink_interval++;
}
```

5.2.4 Change the blinking cycle based on touch detection of Touch Button2

Create a program that changes the blinking cycle in a rotation of 2 seconds → 1 second → 0.5 seconds each time Touch Button 2 is detected during LED blinking operation.

(1) Implement the touch detection program for Touch Button2

Create a conditional statement to detect when Touch Button2 is touched.

To restrict the operation to when the LED is blinking, write the code within the if(g_button1_push_count == 1) condition.

Create a conditional statement for Touch Button2 in the same way as step (3)-(c).

The conditional statement is as follows.

```
            if(g_button1_push_count == 1)
            {
                if(g_led_blink_interval >= 2000)
                {
                    g_led_blink_interval = 0;
                    err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                    while(err);
                    if(value == BSP_IO_LEVEL_HIGH)
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
                        while(err);
                    }
                    else
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
                        while(err);
                    }
                }

                if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
                {

                }
            }
```

Figure5-62. Implement main program

(2) Implement increment processing when Touch Button2 is touched

Create a variable that increments when Touch Button2 is touched.

uint8_t g_led_blink_table_index;

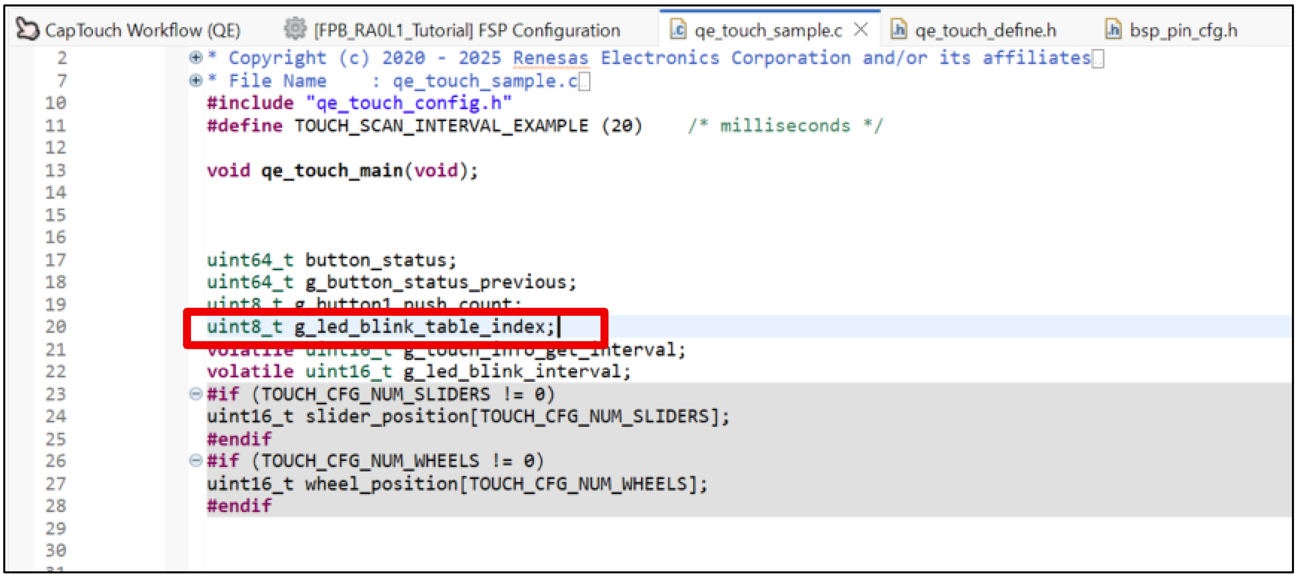


Figure5-63. Implement main program

Add increment processing to the conditional statement created in step (a).

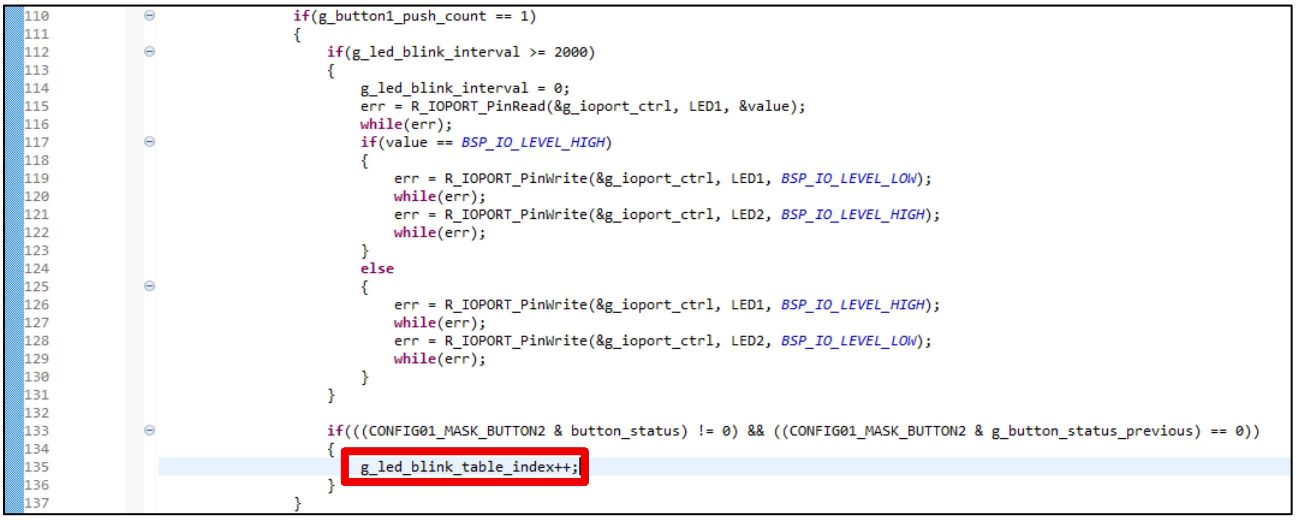
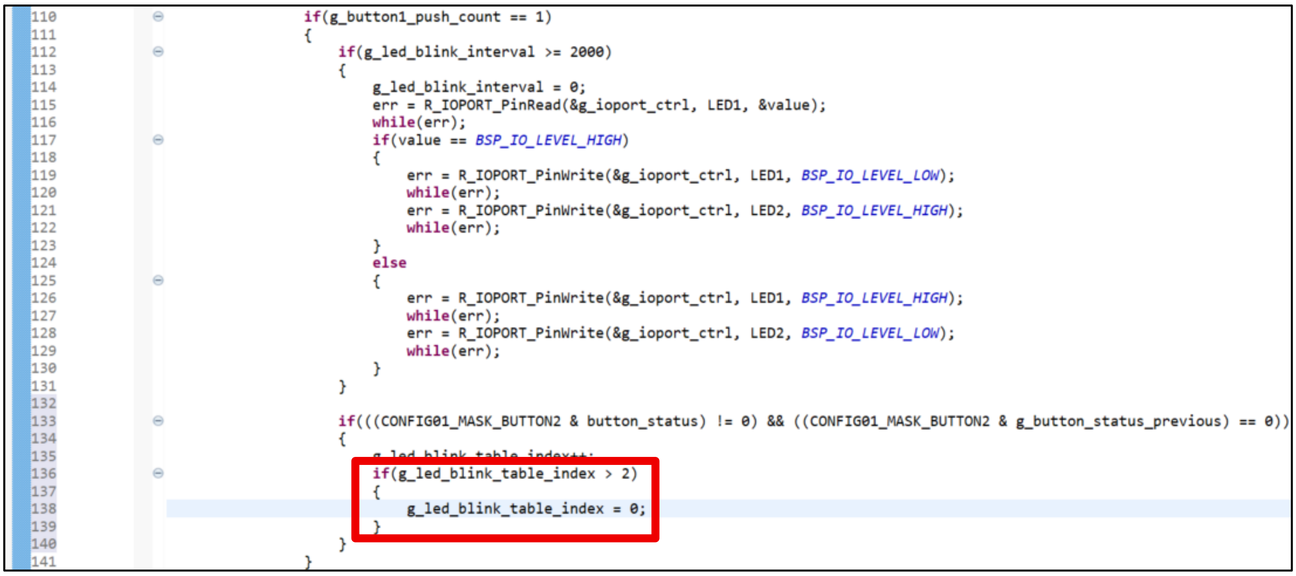


Figure5-64. Implement main program

(3) Implement blinking cycle rotation control when Touch Button2 touch detection

Create a conditional statement to initialize the incremented variable when the Touch Button2 is touched three times, causing the blinking cycle to rotate through 2 seconds → 1 second → 0.5 seconds each time a touch is detected.



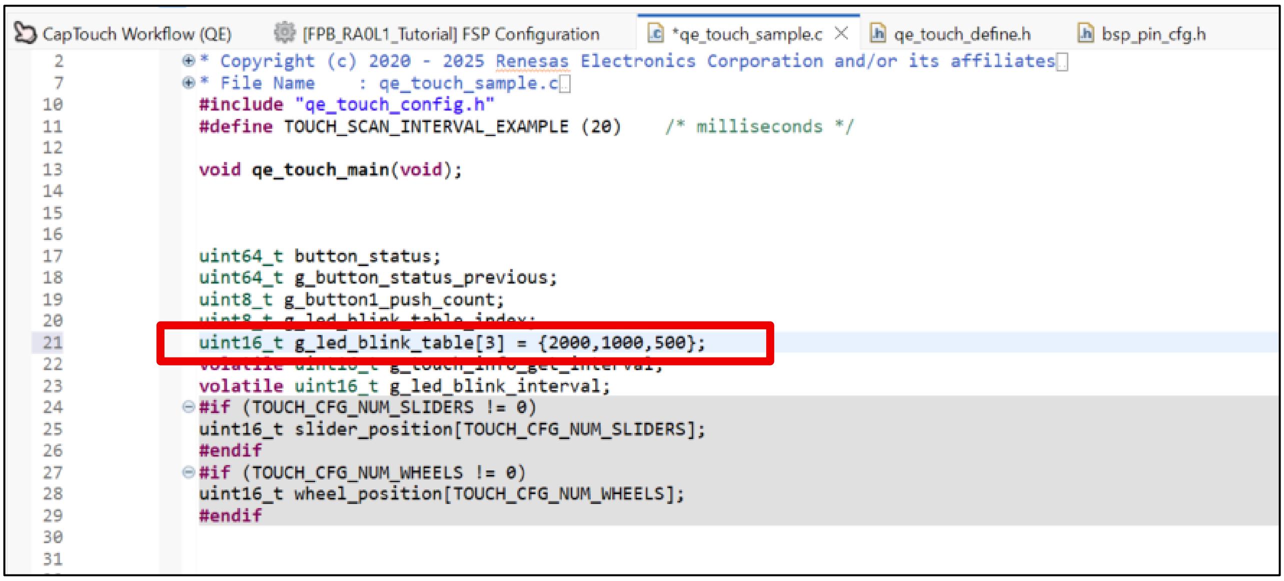
```
if(g_button1_push_count == 1)
{
    if(g_led_blink_interval >= 2000)
    {
        g_led_blink_interval = 0;
        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
        while(err);
        if(value == BSP_IO_LEVEL_HIGH)
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
            while(err);
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
            while(err);
        }
        else
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
            while(err);
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
            while(err);
        }
    }

    if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
    {
        g_led_blink_table_index++;
        if(g_led_blink_table_index > 2)
        {
            g_led_blink_table_index = 0;
        }
    }
}
```

Figure5-65. Implement main program

(4) Add a table variable to define the blinking period

In step (3)-(g), the blinking cycle is fixed at 2 seconds, but to change the blinking cycle to a rotation of 2 seconds → 1 second → 0.5 seconds, create an array variable that defines the cycle.

uint16_t g_led_blink_table[3] = {2000,1000,500};




```
* Copyright (c) 2020 - 2025 Renesas Electronics Corporation and/or its affiliates
* File Name    : qe_touch_sample.c
#include "qe_touch_config.h"
#define TOUCH_SCAN_INTERVAL_EXAMPLE (20)    /* milliseconds */

void qe_touch_main(void);



uint64_t button_status;
uint64_t g_button_status_previous;
uint8_t g_button1_push_count;
uint8_t g_led_blink_table_index;
uint16_t g_led_blink_table[3] = {2000,1000,500};
volatile uint16_t g_touch_info_get_interval;
volatile uint16_t g_led_blink_interval;
#if (TOUCH_CFG_NUM_SLIDERS != 0)
uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
#endif
#if (TOUCH_CFG_NUM_WHEELS != 0)
uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
#endif
```

Figure5-66. Implement main program

(5) Modify the blinking period determination statement

Replace the 2-second elapsed condition created in step (3)-(f) with the array created in step (4)-(d).

```c
                if(g_button1_push_count == 1)
                {
                    if(g_led_blink_interval >= g_led_blink_table[g_led_blink_table_index])
                    {
                        g_led_blink_interval = 0;
                        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                        while(err);
                        if(value == BSP_IO_LEVEL_HIGH)
                        {
                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
                            while(err);
                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
                            while(err);
                        }
                        else
                        {
                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
                            while(err);
                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
                            while(err);
                        }
                    }

                    if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
                    {
                        g_led_blink_table_index++;
                        if(g_led_blink_table_index > 2)
                        {
                            g_led_blink_table_index = 0;
                        }
                    }
                }
```

Figure5-67. Implement main program

(6) Fixed blinking cycle at the start of blinking

Initialize the g_led_blink_table_index variable at the start of the blinking operation to ensure the blinking period is always 2 seconds.

Therefore, the initialization process is added within the if(g_button1_push_count > 1) condition.

```c
                if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
                {
                    g_button1_push_count++;
                    if(g_button1_push_count > 1)
                    {
                        g_button1_push_count = 0;
                        g_led_blink_table_index = 0;
                    }
                }
```

Figure5-68. Implement main program

The coding for the sample program is now complete.

The source code up to this point is provided as text. The parts that have been added since qe_touch_sample.c was generated are shown in blue.

```
void qe_touch_main(void);

uint64_t button_status;
uint64_t g_button_status_previous;
uint8_t g_button1_push_count;
uint8_t g_led_blink_table_index;
uint16_t g_led_blink_table[3] = {2000,1000,500};
volatile uint16_t g_touch_info_get_interval;
volatile uint16_t g_led_blink_interval;
#if (TOUCH_CFG_NUM_SLIDERS != 0)
uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
#endif
#if (TOUCH_CFG_NUM_WHEELS != 0)
uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
#endif

void qe_touch_main(void)
{
    fsp_err_t err;
    bsp_io_level_t value;

    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
                        g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;
            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
                                   &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
```

```
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                    BSP_IO_LEVEL_LOW);
            while(err);
        }
        else
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                    BSP_IO_LEVEL_HIGH);
            while(err);
        }

        if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                    BSP_IO_LEVEL_LOW);
            while(err);
        }
        else
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                    BSP_IO_LEVEL_HIGH);
            while(err);
        }

        if(((CONFIG01_MASK_BUTTON1 & button_status) != 0)
          && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
        {
            g_button1_push_count++;
            if(g_button1_push_count > 1)
            {
                g_button1_push_count = 0;
                g_led_blink_table_index = 0;
            }
        }

        if(g_button1_push_count == 1)
        {
            if(g_led_blink_interval
                >= g_led_blink_table[g_led_blink_table_index])
            {
                g_led_blink_interval = 0;
                err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                while(err);
                if(value == BSP_IO_LEVEL_HIGH)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
```

```
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
                                                BSP_IO_LEVEL_LOW);
                        while(err);
                    }
                }

                if(((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                 && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
                {
                    g_led_blink_table_index++;
                    if(g_led_blink_table_index > 2)
                    {
                        g_led_blink_table_index = 0;
                    }
                }
            }
            g_button_status_previous = button_status;
        }
        /* FIXME: Since this is a temporary process,
                  so re-create a waiting process yourself. */
        /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
                                BSP_DELAY_UNITS_MILLISECONDS); */
        g_touch_info_get_interval = 0;
    }
  }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
    g_led_blink_interval++;
}
```

5.3 Rebuild

After completing the coding, build the project.

Click the launch icon to start the build.

The build log is output to the “Console” window.

If “Build Finished. 0 errors,” is displayed at the end, the build has completed successfully.

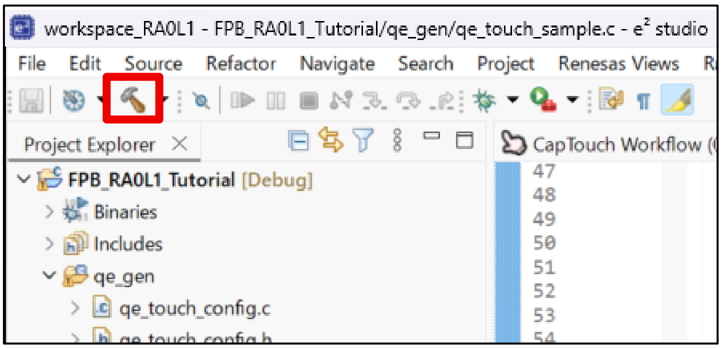


Figure5-69. Rebuild

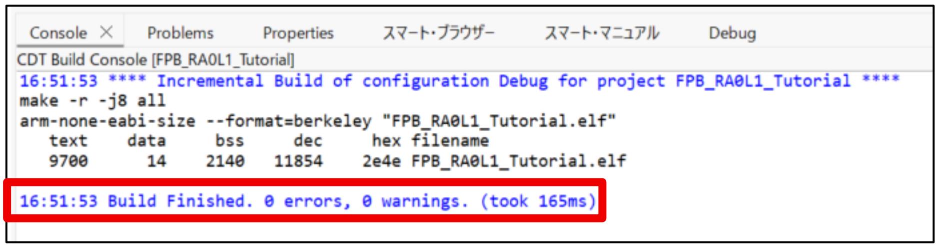


Figure5-70. Rebuild

5.4 Execute

After the build is complete, the program is written to the MCU on the board.

If the write settings have been configured in “4.3.1 Debug settings and launch,” click the bug icon in the red frame to start debugging.

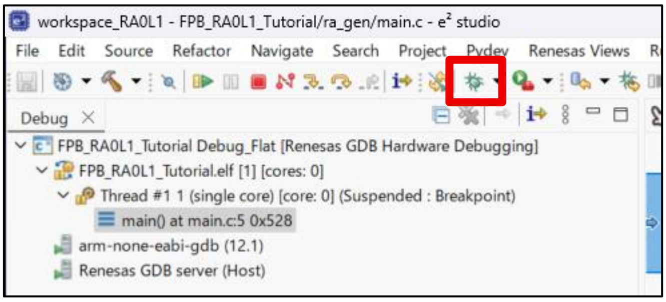


Figure5-71. Execute

Once the program has been downloaded, verify that it is paused at SystemInit();.

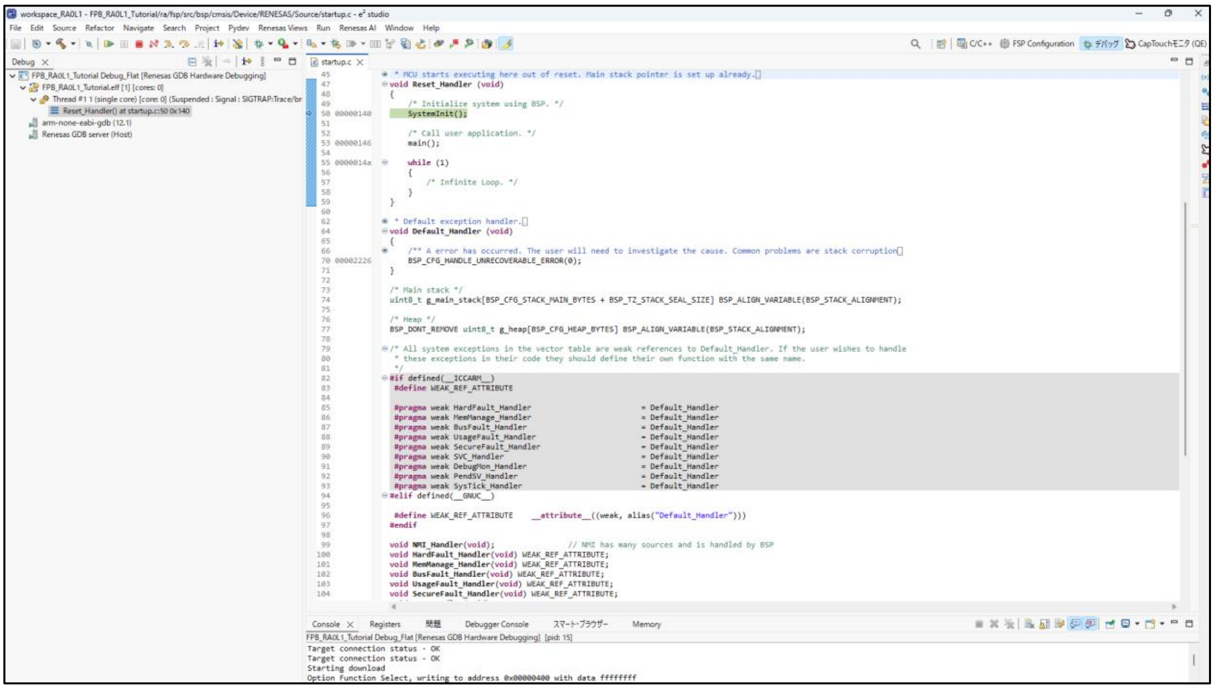


Figure5-72. Execute

Click the resume icon within the red frame to continue execution.

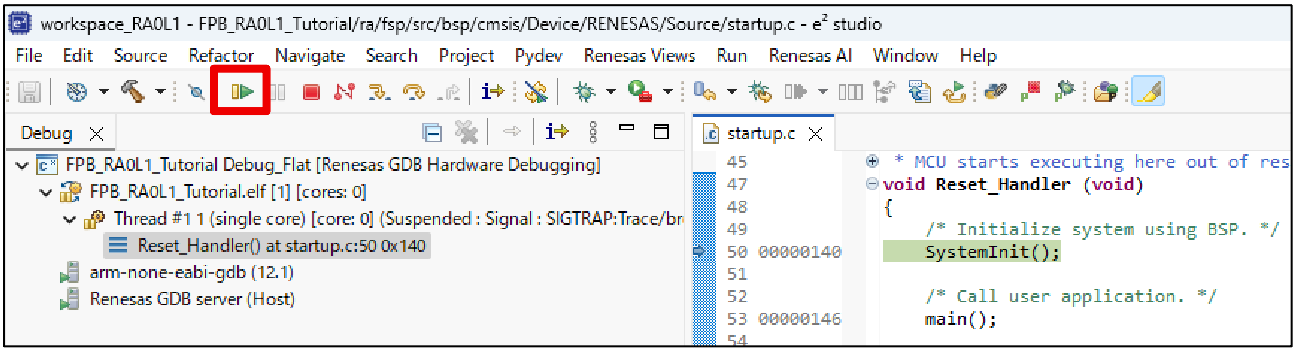


Figure5-73. Execute

The program pauses at the beginning of the main function. (This is the default setting for e² studio projects)

This indicates that SystemInit() has completed, and initialization is finished.

Click the restart icon again to execute.

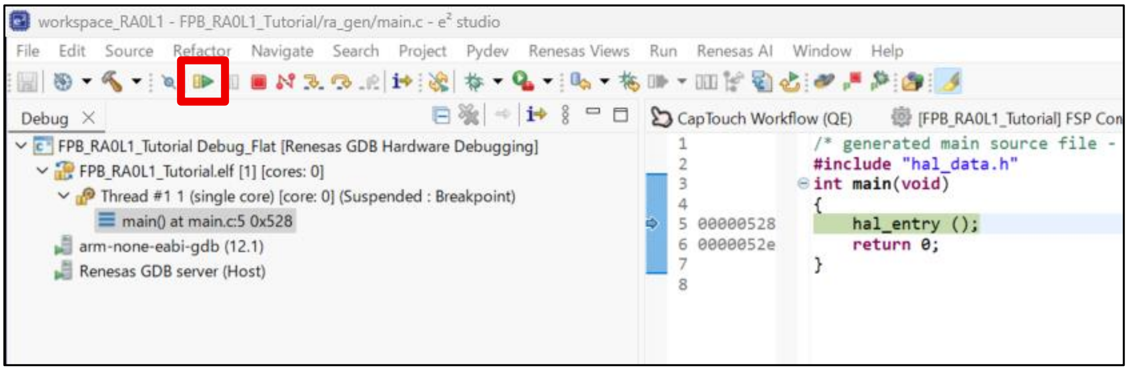


Figure5-74. Execute

The status "Running" appears in the lower-left corner of the screen. In this state, the program is executing. When Button1 and Button2 on the board are touched, LED5 and LED6 light up, and the blinking operation of LED1 and LED2 can be confirmed.

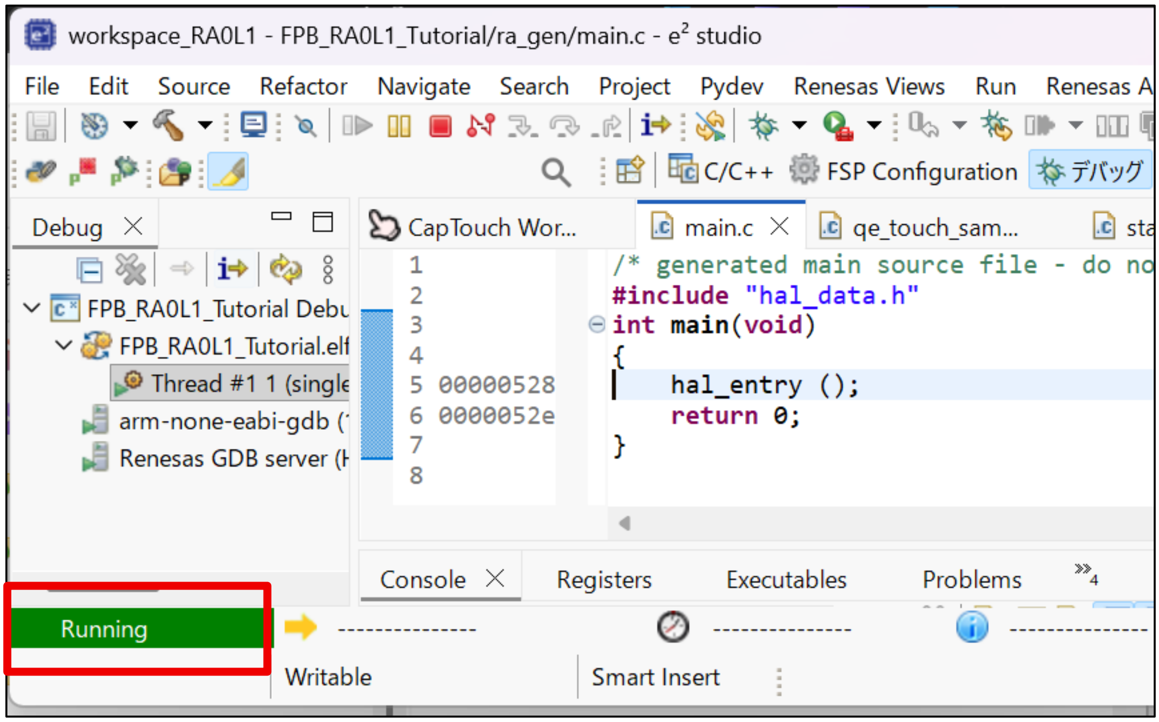


Figure5-75. Execute

Revision History

Rev.	Date	Description	
		page	Summary
1.00	2025.09.03	-	First edition issued

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.