RA8E1 Group

FPB-RA8E1 Tutorial

Introduction

This application note explains how to create a new project and debug a program using the e^2 studio integrated development environment for the FPB-RA8E1 board with Renesas' RA8E1 group MCU.

Target Device

RA8E1 group

Related Documents

[1] Renesas RA Family FPB-RA8E1 v1 User's Manual (R20UT5378)

[2] Integrated development environment e^2 studio 2022-07 or higher User's manual Quick start guide Renesas microcontroller RA family (R20UT5210)

Contents

1. Development environment

This application note explains using the following development environment.

1.1 Hardware environment

Use the following hardware:

- Board: FPB-RA8E1 (RTK7FPA0E1S00001BJ)
  Connect the board and PC using the USB Type A to Type C cable.

1.2 Software environment

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

- Integrated development environment
  — e^2 studio 2025- 10 or later
- Compiler (either can be used)
  — GNU ARM Embedded: 13.2.1.arm-13.7 or later
  — LLVM for ARM： 18.1.3 or later
- Flexible Software Package
  — Version: 6.2.0 or later
  Download: https://github.com/renesas/fsp/releases
  User's Manual: RA Flexible Software Package Documentation

2. Software overview

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

2.1 Program to create

A program which alternately toggles two on-board LEDs using a 500ms periodic timer interrupt.

Indicates the board used and the location of the LEDs.

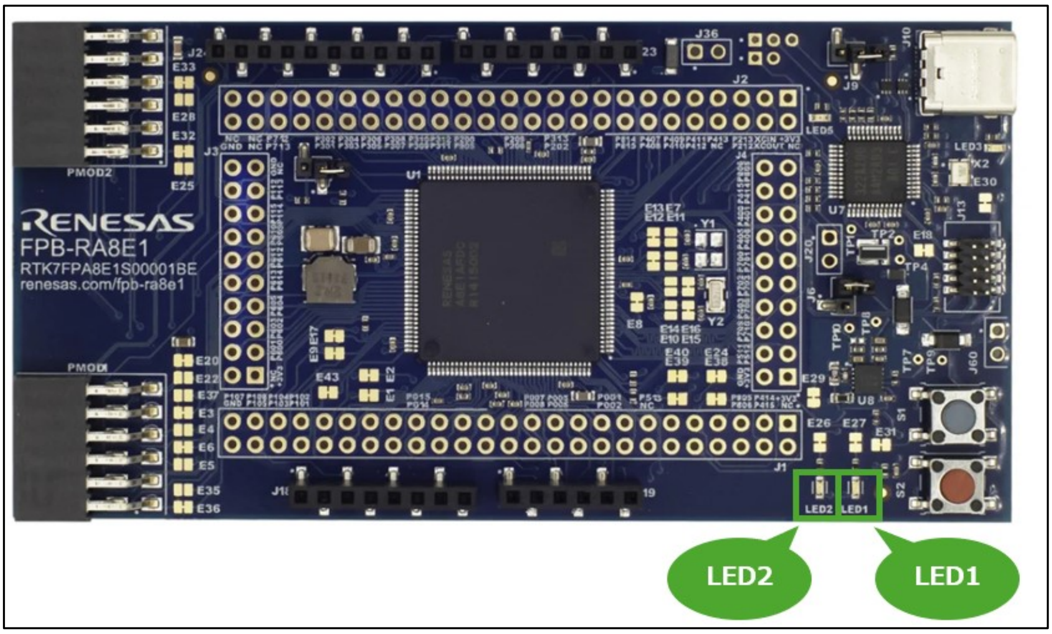

Figure 2-1 FPB-RA8E1 LED position

2.2 Resources

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

2.2.1 Clock

HOCO Clock Frequency	: 32 MHz
HOCO Clock Division	: ÷1
CPU Clock (CPUCLK)	: 32 MHz
System Clock (ICLK)	: 32 MHz
Flash IF Clock (FCLK)	: 32 MHz
Peripheral Module Clock D (PCLKD)	: 32 MHz

2.2.2 Timer

Timer Function Used	: General PWM Timer Channel 0 (GPT320)
General PWM Timer (GPT) Count Clock	: 32 MHz
Timer Period	: 500 ms
FSP Software Stack Used	: r_gpt

2.2.3 Port

Ports used	:P404 (LED1) / P408 (LED2)
FSP software stack used	:r_ioport

2.3 Flowchart

2.3.1 Main Process

The flowchart of the main process is shown below.

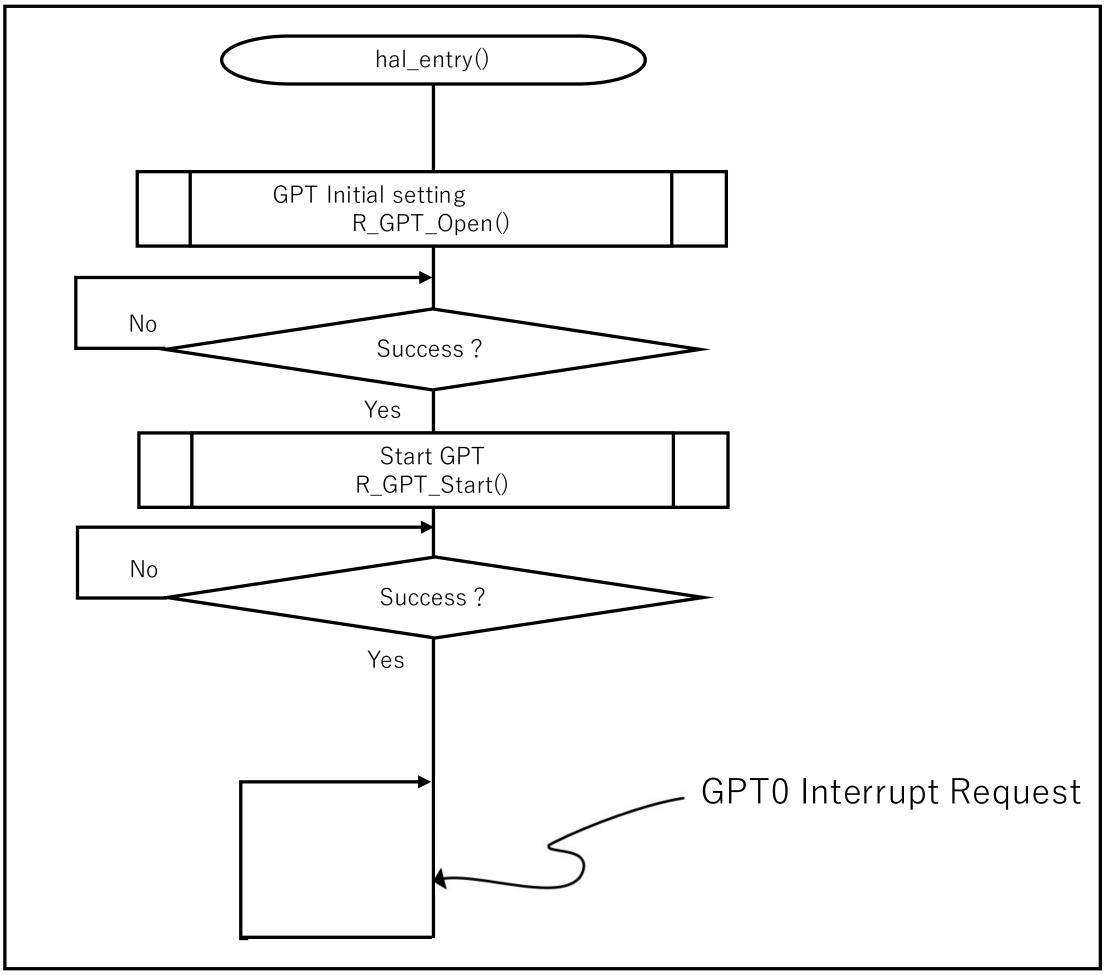

Figure 2-2 Main Process

2.3.2 Interrupts

The flowchart for interrupts is shown below.

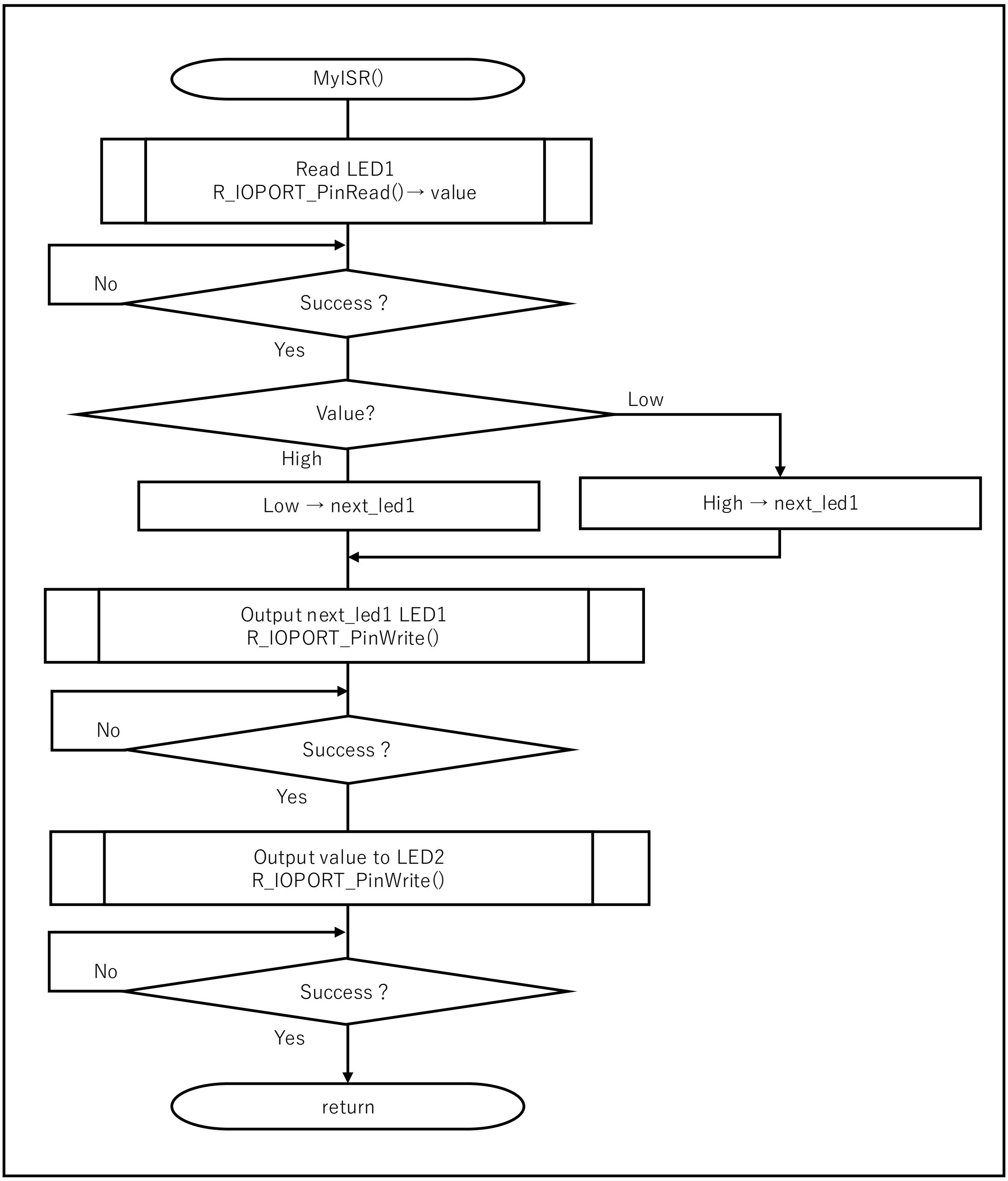

Figure 2-3 Interrupts

3. How to create a program

This chapter explains how to create a project, set up peripheral functions using FSP, write application code, and build.

3.1 Create new project

3.1.1 Project launch

1. From the e^2 studio menu bar
   Select "**New**" → "**Renesas C/C++ Project** " → " **Renesas RA** ".

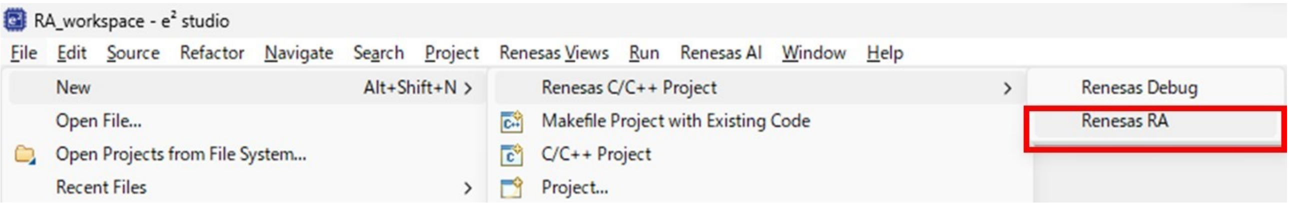

Figure 3-1 Project launch

2. "**Renesas RA C/C++ Project**" and click "**Next**".

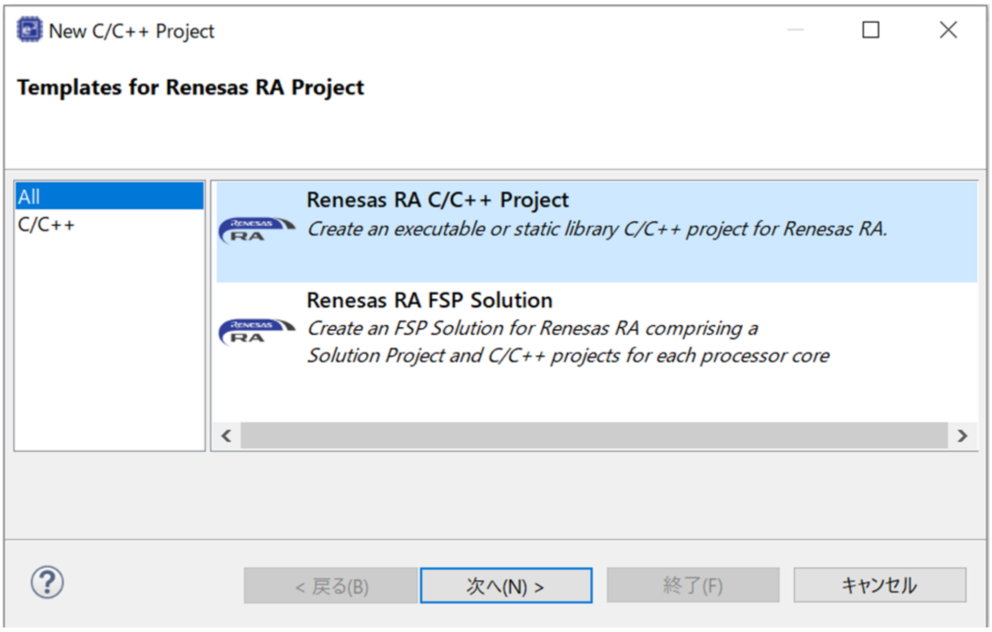

Figure 3-2 Project launch

3. Project name Write any name and click "**Next**".
" Use default location ", the project will be created in the path shown below. If you want to create it in another location, uncheck it and set the path.

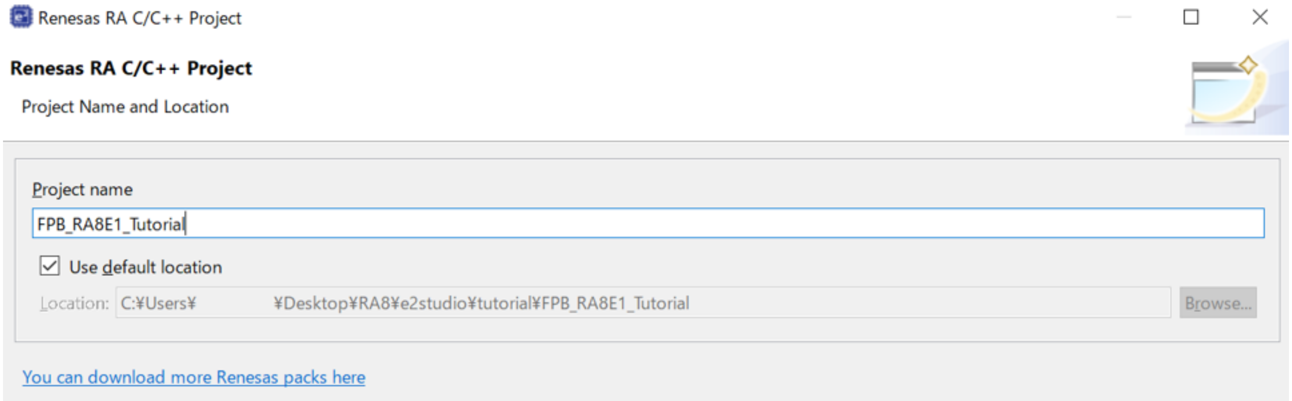

Figure 3-3 Project launch

3.1.2 Device/Tool Configuration

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

1. FSP Version : Select the latest version
   Language : Select C.

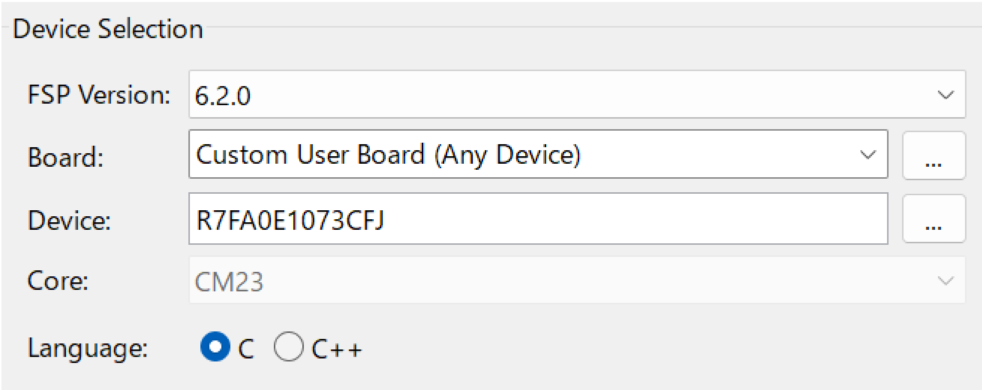

Figure 3-4 Device/Tool Configuration

2. Board: Select **FPB-RA8E1** from the list**.**
   (**Device** and **core** is set automatically)

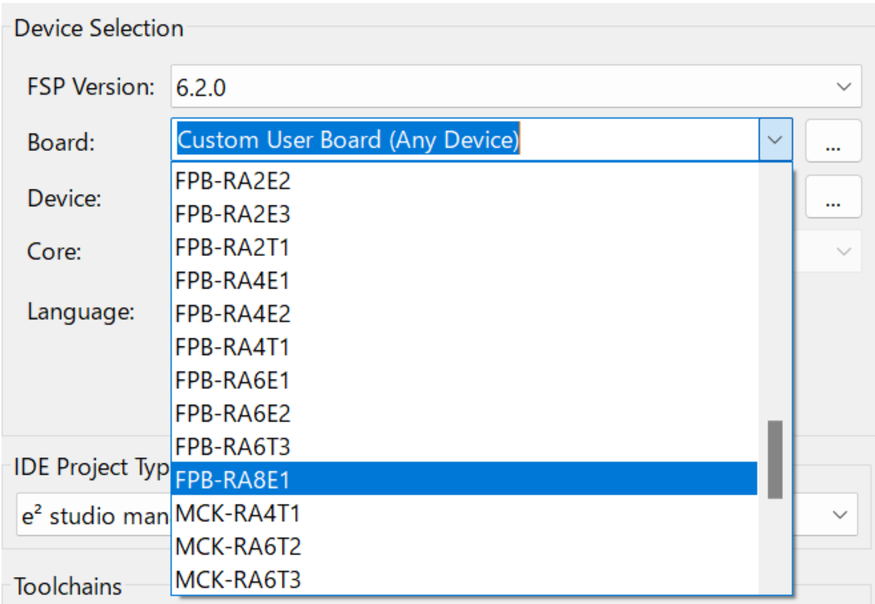

Figure 3-5 Device/Tool Configuration

3. Toolchains settings : Select either the **“LLVM Embedded Toolchain for Arm”** or **“GNU ARM Embedded”**. From the list of available versions, choose the **latest version**.

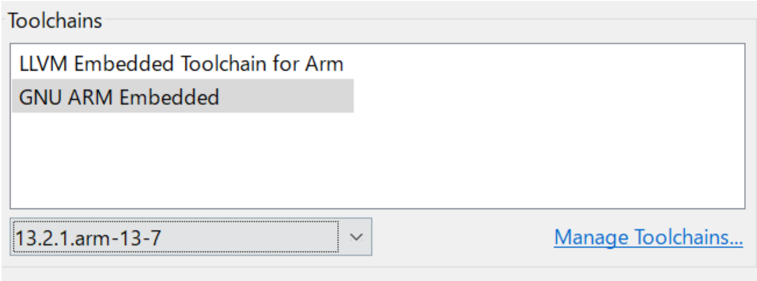

Figure 3-6 Device/Tool Configuration

4. Debugger settings: Select **J -Link ARM**.

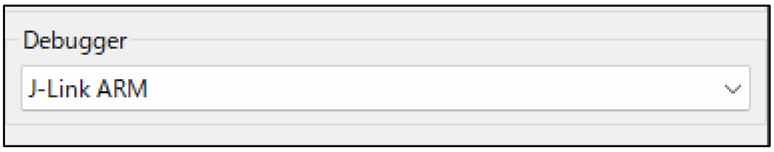

Figure 3-7 Device/Tool Configuration

The settings are complete.

5. Confirm the settings in the red framed areas and click "**Next**”.

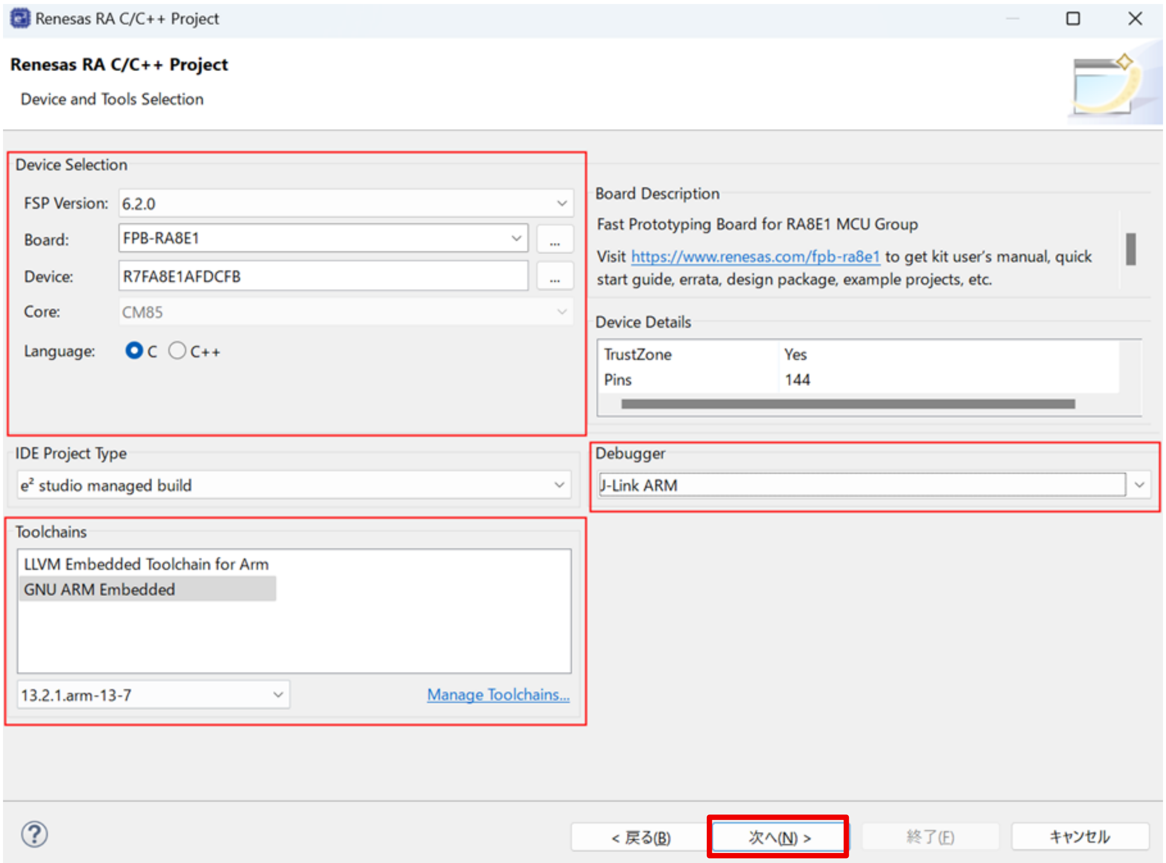

Figure 3-8 Device/Tool Configuration

3.1.3 Project Settings

Set up the TrustZone configuration for the project. As TrustZone is not utilized in this application note, please select a **Flat (Non-TrustZone) project.**

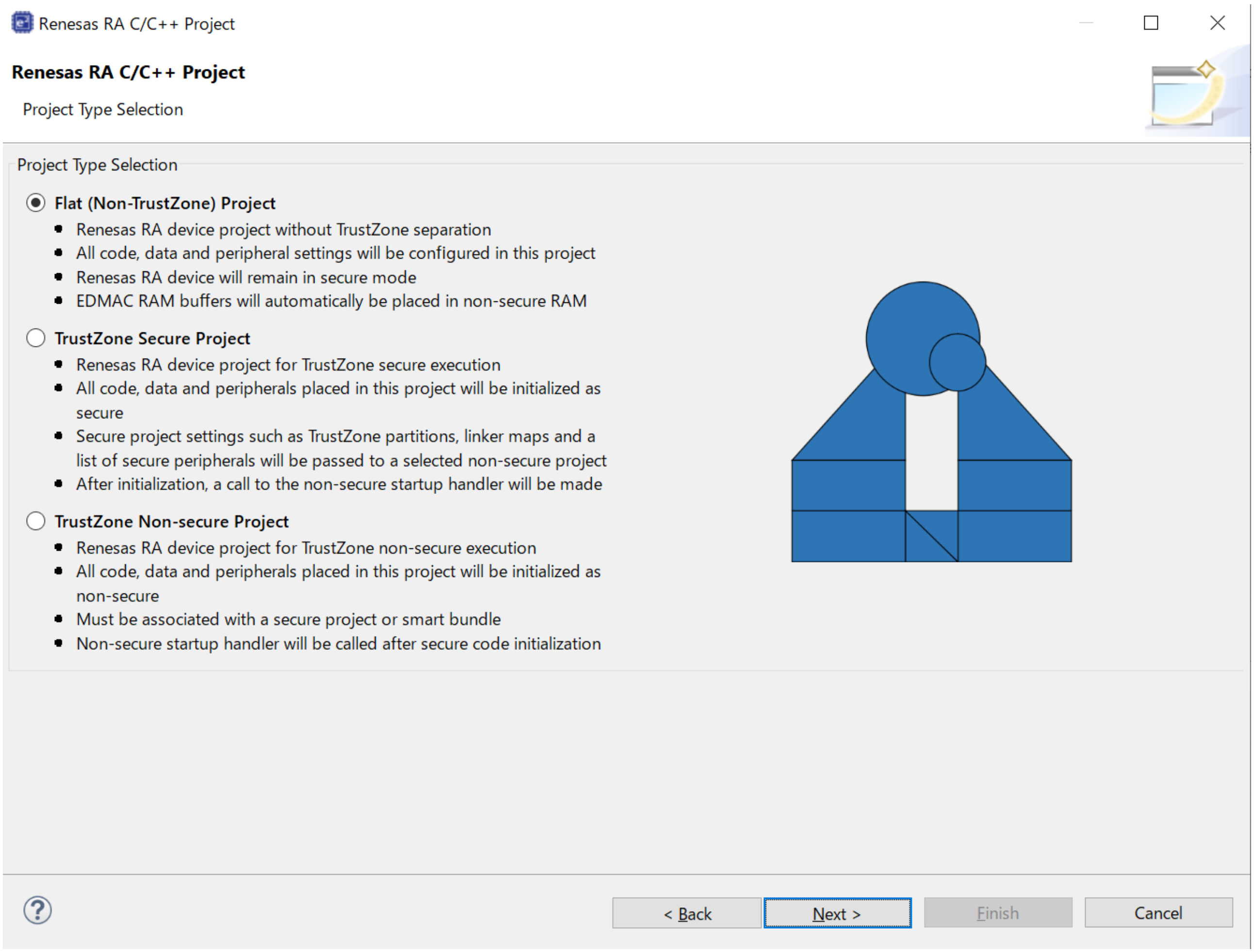

Figure 3-9 Project Settings

3.1.4 Smart Bundle Selection settings

Smart Bundle Selection is not utilized in this application note; therefore, please select "**None**".

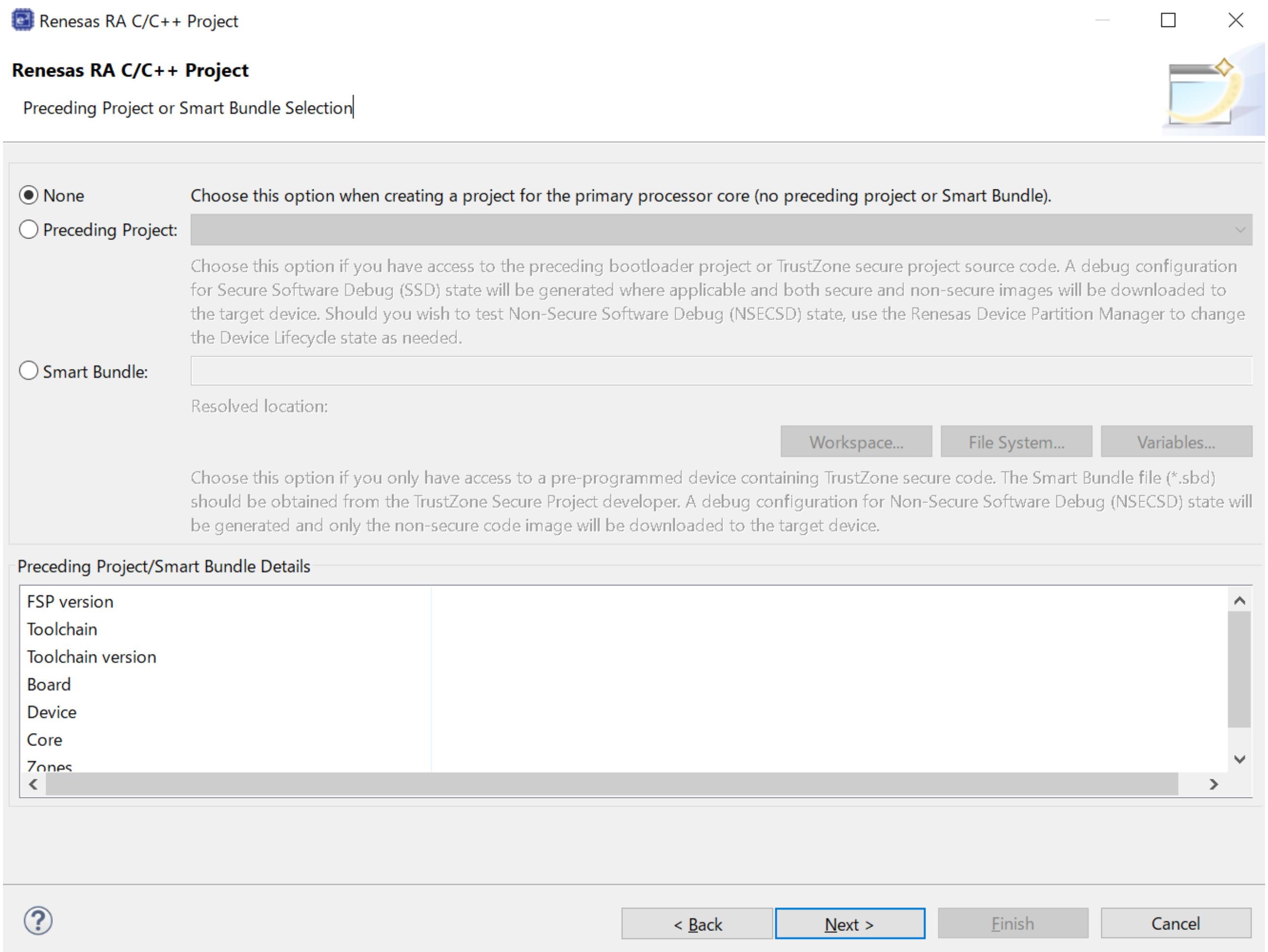

Figure 3-10 Smart Bundle Selection settings

3.1.5 Build artifact settings

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

Since we do not use RTOS, select **No RTOS**.

Click **Next** to proceed.

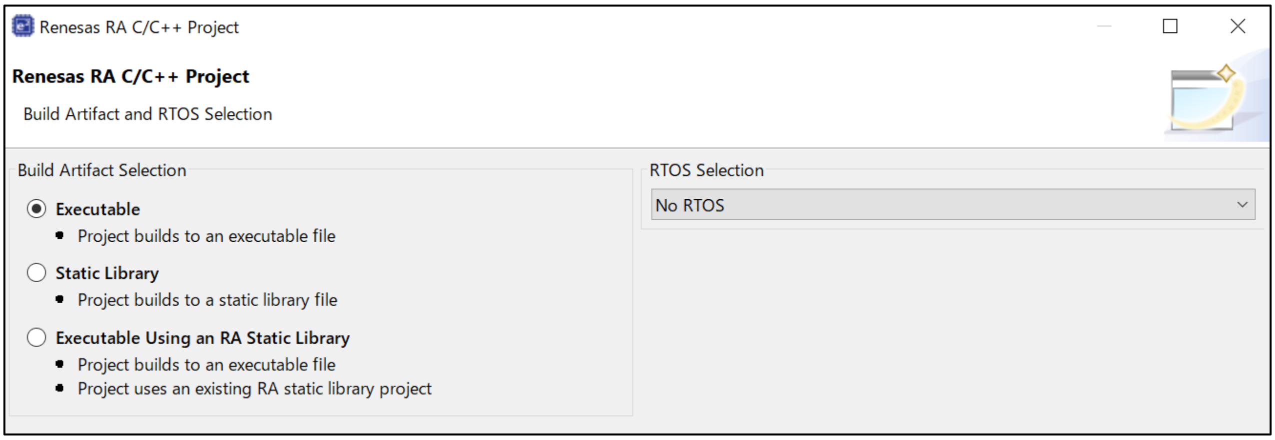

Figure 3-11 Build artifact settings

3.1.6 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**".

   Brief overview of the options:
   Bare Metal - Blinky will generate an application which toggles all on board LEDs determined from the BSP using a simple software delay.
   Bare Metal - Minimal will generate an empty application with basic C-runtime setup but no executable code.

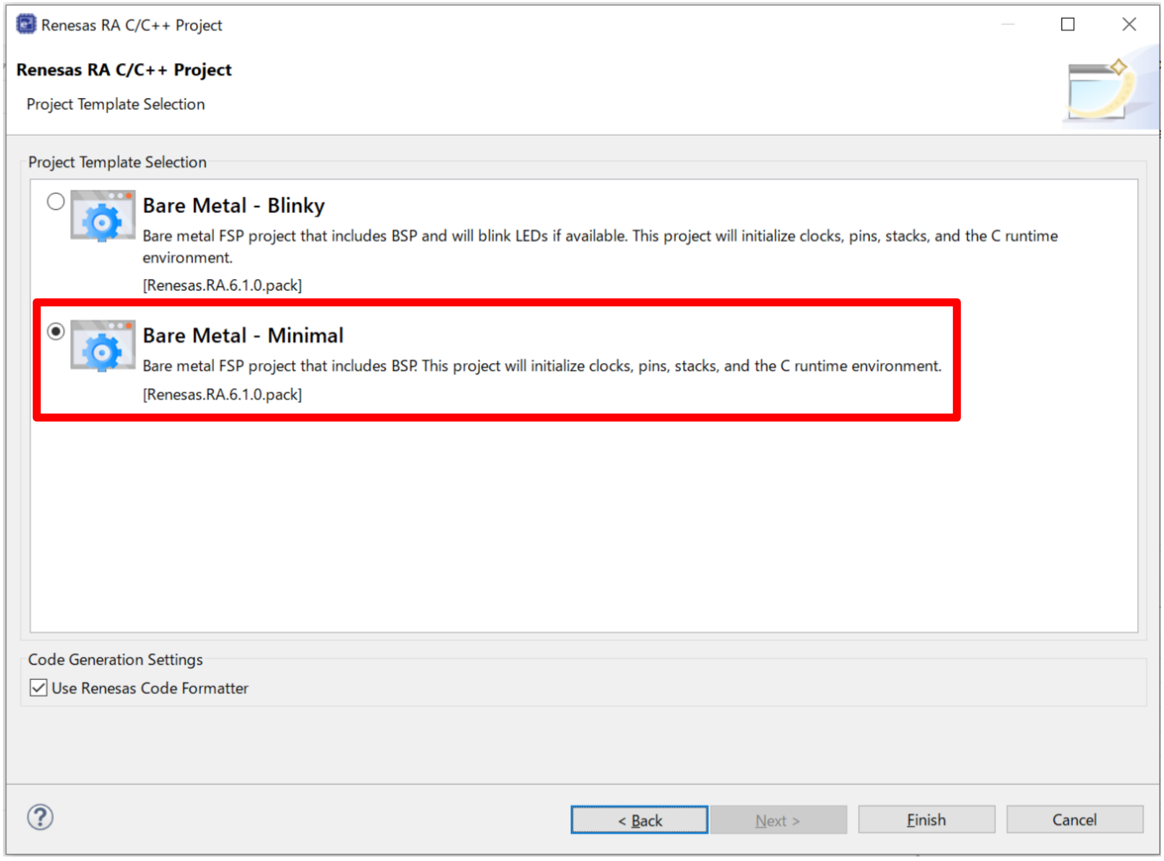

Figure 3-12 Template type settings

2. A popup for the perspective view will appear. Click "Open Perspective".
   Note: Clicking "No" is also acceptable.

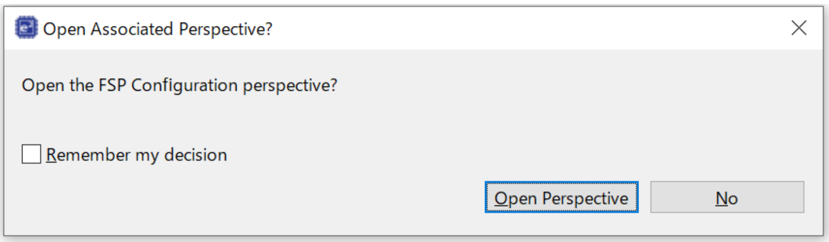

Figure 3-13 Popup

3. Project creation is complete.

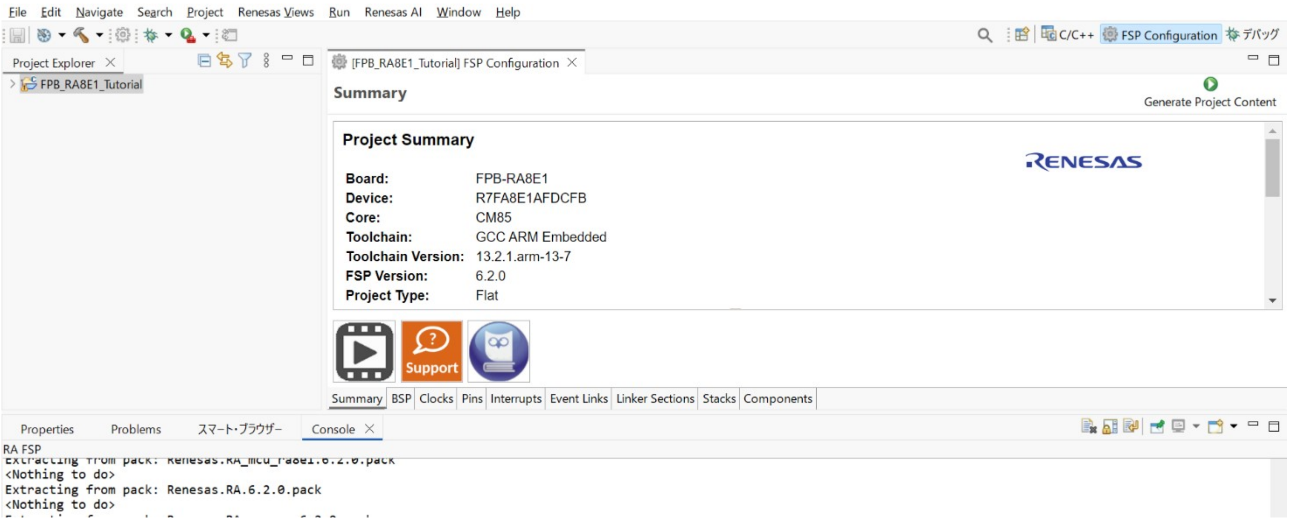

Figure 3-14 Project creation complete

3.2 FSP Configurator settings

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

3.2.1 How to call the FSP Configurator

The FSP Configuration page should be open by default. If it is not open, double-click the **configuration.xml** file in the Project Explorer to display the configuration screen.

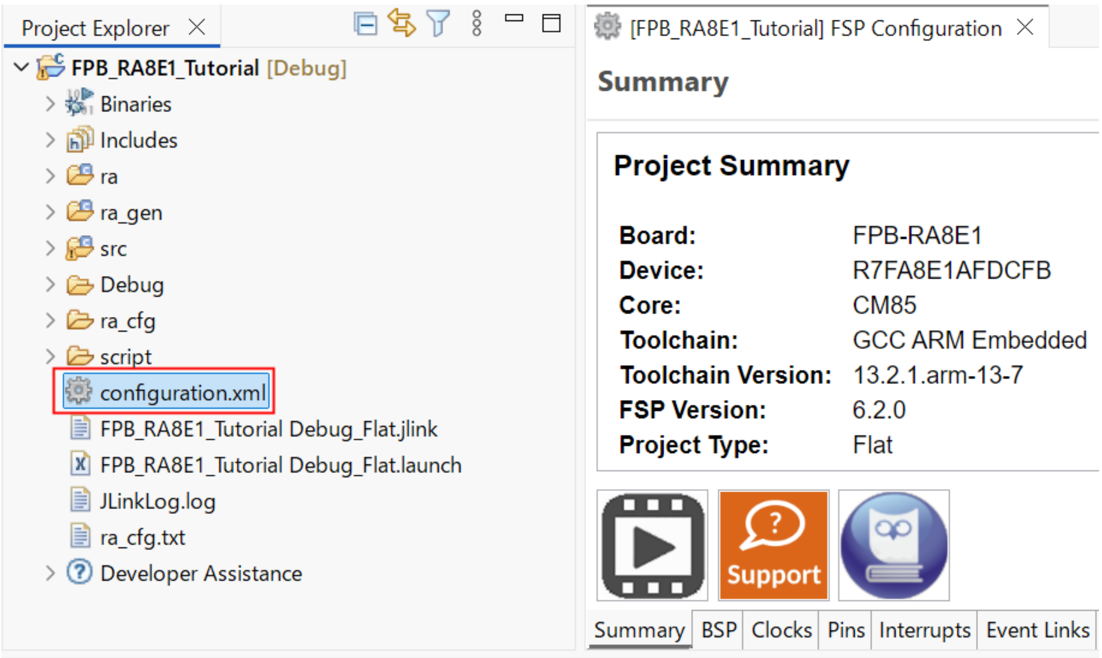

Figure 3-15 How to call the FSP Configurator screen

3.2.2 Clock setting

1. Select the "**Clocks**"tab to open the clock settings screen. Make sure the default screen looks like the one below.
   The path of each clock is indicated by a thick arrow.

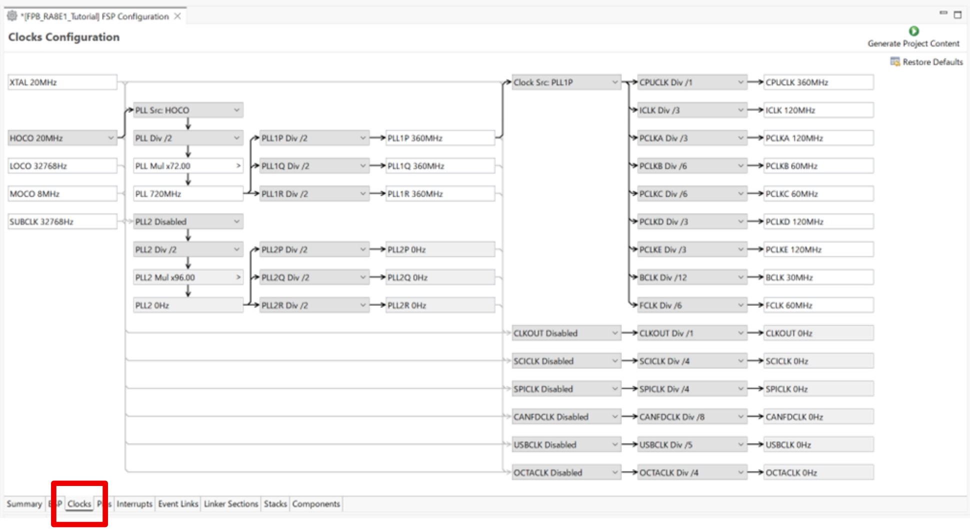

Figure 3-16 Clock setting

2. In this project, the High-Speed On-Chip Oscillator (HOCO) at 32 MHz will be used. Click on HOCO 20 MHz, then select **HOCO 32 MHz** from the list.

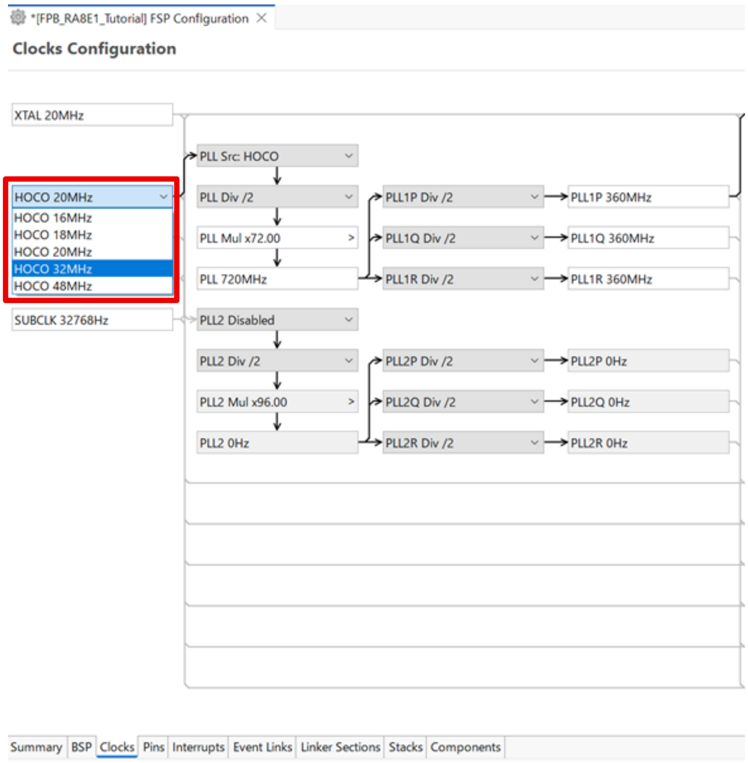

Figure 3-17 Clock setting

3. After selecting HOCO 32 MHz, click on "PLL Src: HOCO" and select "**PLL Src: Disabled**".

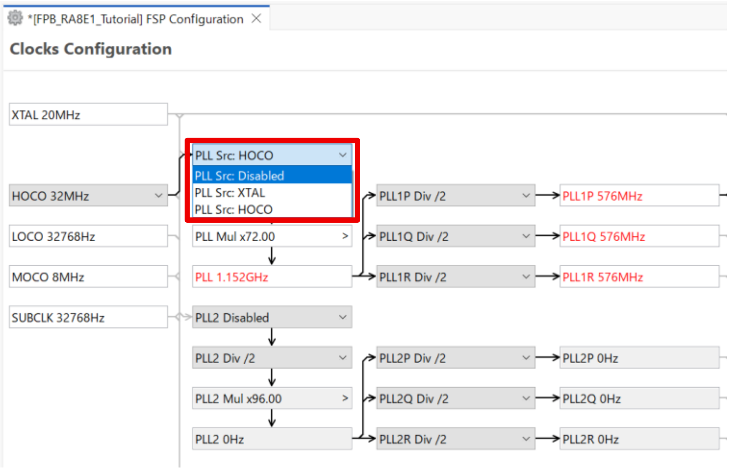

Figure 3-18 Clock setting

4. After selecting "PLL Src: Disabled", click on "Clock Src: PLL1P" and select "**Clock Src: HOCO**".

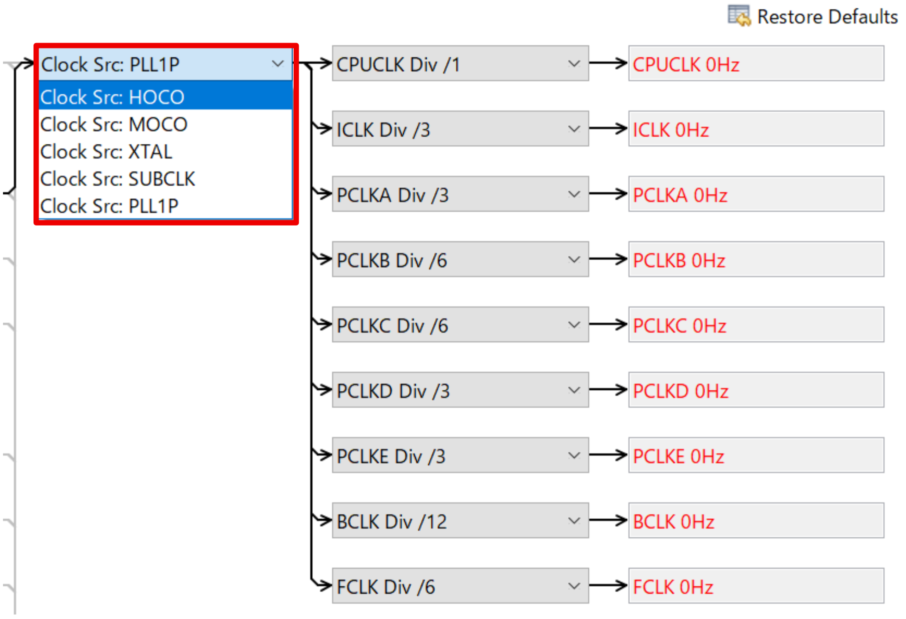

Figure. 3-19 Clock setting

5. The count clock for the General PWM Timer (GPT) is based on PCLKD. In this project, click on **PCLKD Div/3** and select **PCLKD Div/1**.

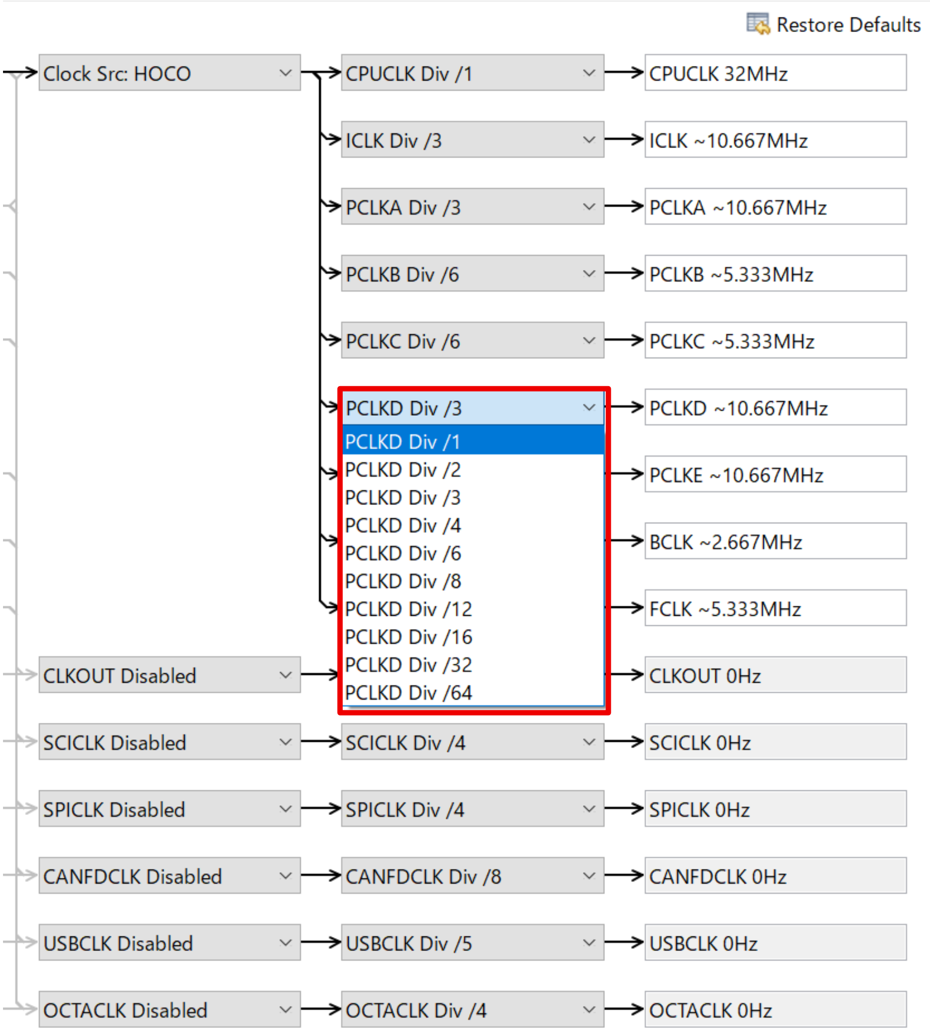

Figure. 3-20 Clock setting

3.2.3 Pin settings

1. Select the "**Pins**" tab and then the "**Pin Function**" tab. From the Pin Selection list, choose P404, which is assigned to LED1.
   Since the board was selected in section **3.1.2 Device and Tool Settings**, some pins already have symbol names and I/O settings configured. P404 has the symbol name **"LED1"**, which can be used in the program implementation.

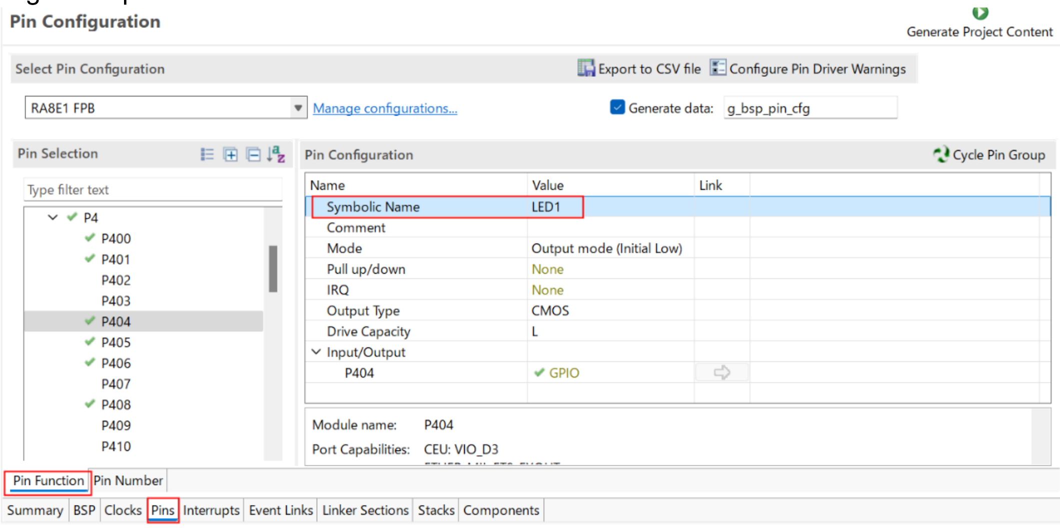

Figure 3-21 Pin settings

2. To set P404 to turn on (output High) by default, select "**Output mode (Initial High)**" from the Mode list.

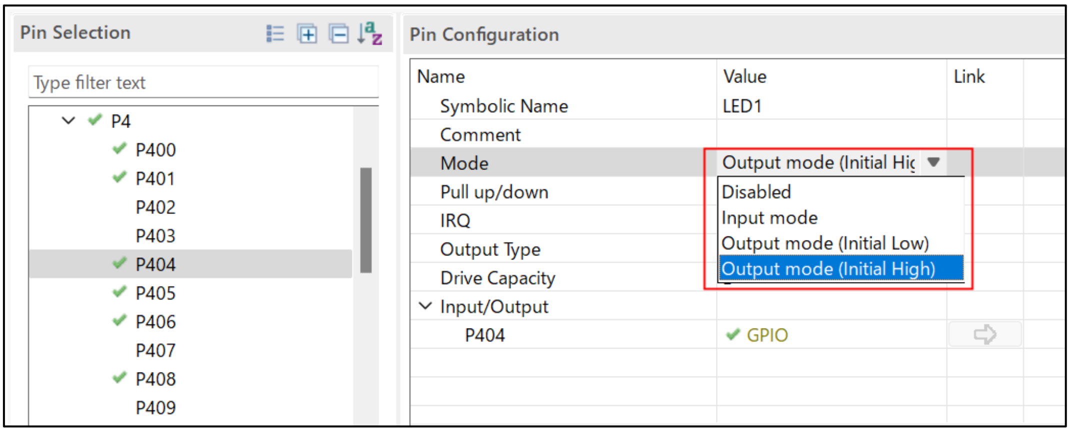

Figure 3-22 Pin settings

3. To set P408 to turn off (output Low) by default, select "**Output mode (Initial Low)**" from the Mode list.

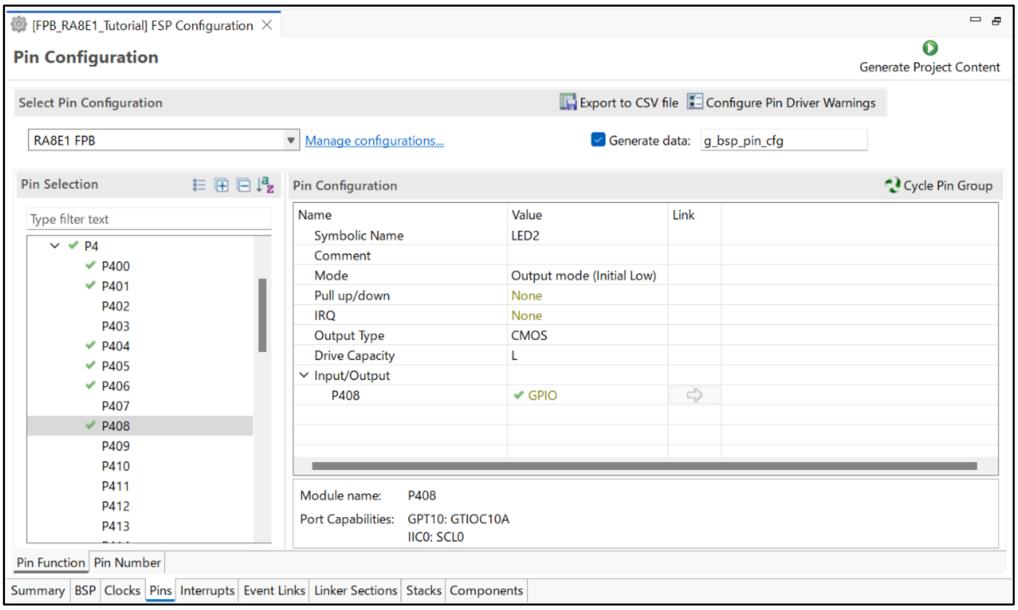

Figure 3-23 Pin settings

3.2.4 Adding the Timer

1. Select the "**Stack**" tab - it is from here we can add all drivers and middleware components.

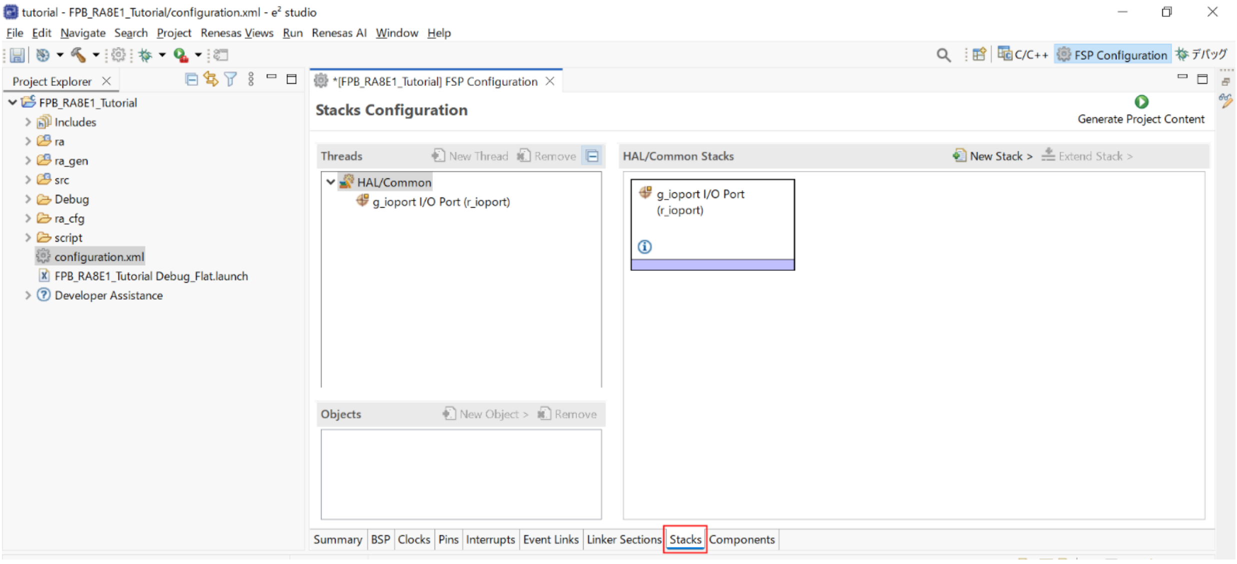

Figure 3-24 Setting the timer function

2. Click "**New Stack**" to display the list of functions, then select "**Timers**" > "**Timer, General PWM (r_gpt)**".

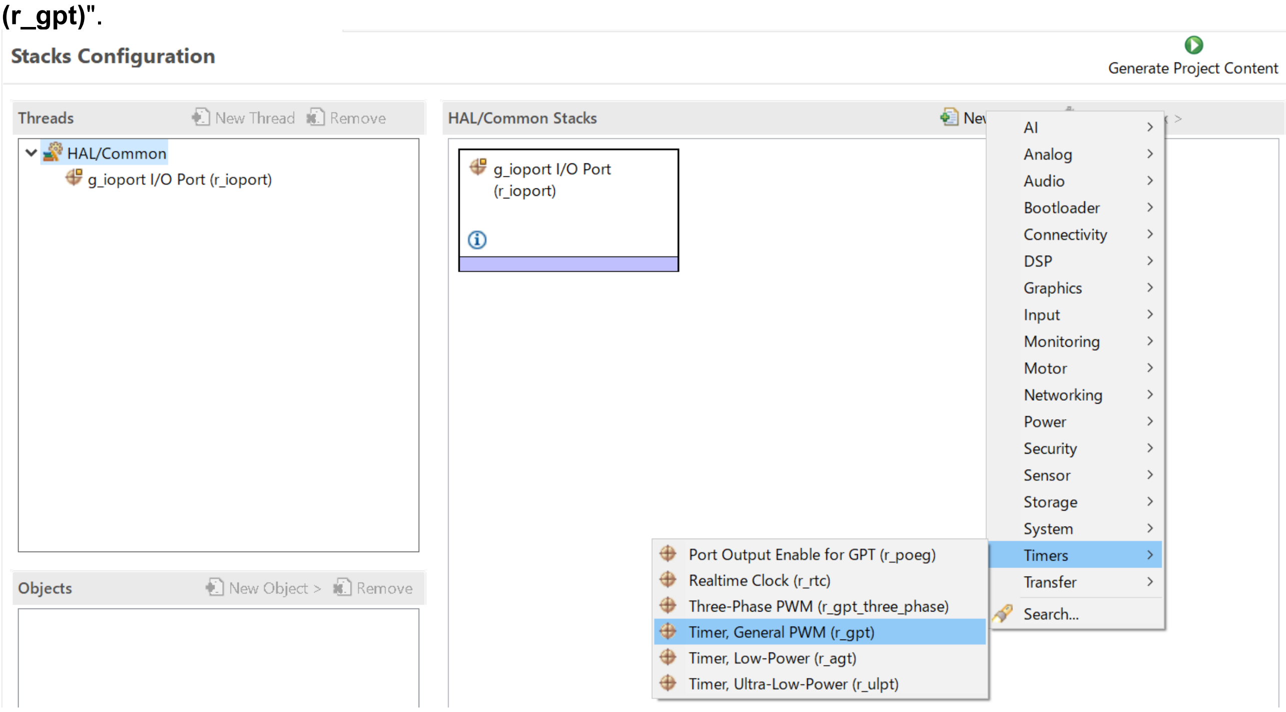

Figure 3-25 Setting the timer function

3. Confirm that a stack named "g_timer0" has been added. With the "**Properties**" window open, click on the "**g_timer0**" block to display detailed timer configuration options.

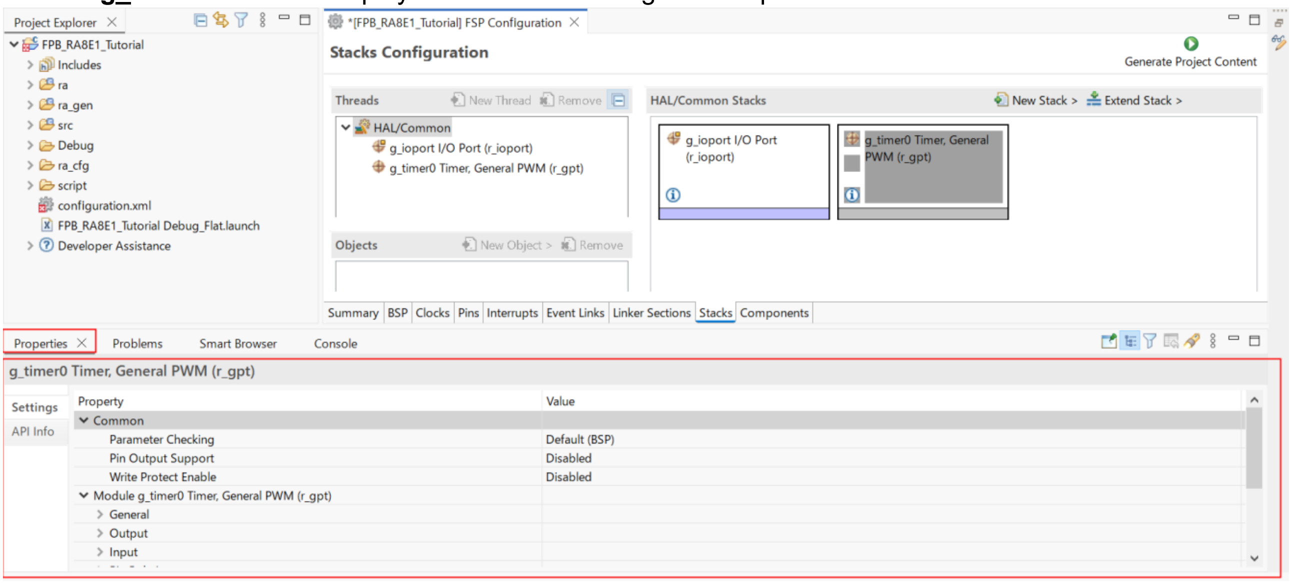

Figure 3-26 Setting the timer function

If you cannot find the "Properties" window, select "Window" → "Show View" → "Properties" from the e[2] studio menu bar to display it.

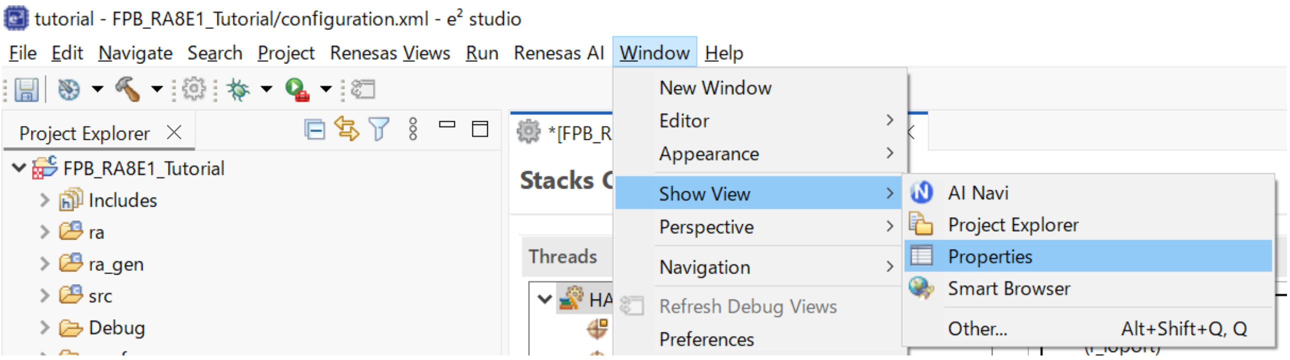

Figure 3-9 Setting the timer function

4. Open the "General" properties list in preparation of the following steps as shown in the red frame below.

Figure 3-28 Setting the timer function

5. Set the name of the timer module.
   Name : Set the name of the timer module. In this case, we will use **"MyTimer"**.

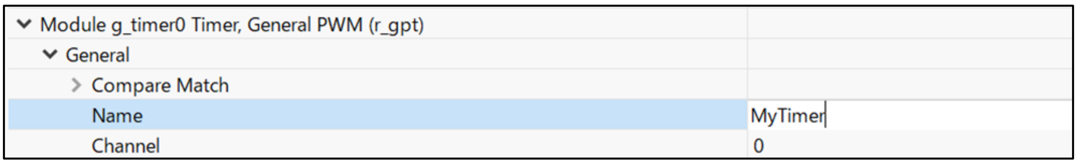

Figure 3-29 Setting the timer function

6. Set the following items.

Channel: 0

This is the GPT peripherals channel we will be using.

Function: Interval Timer

This is the function of the timer, here we will use it as an interval (periodic) timer.

Period: 500

This is the period in "units" of the timers counter.

Period Unit: Milliseconds

These are the units for the "period" being counted.

Figure 3-30 Setting the timer function

7. Now open the "Interrupts" properties list to configure the interrupt settings.

Figure 3-31 Setting the timer function

8. Set a user-implementable interrupt handling function to Call back.
   The default setting "NULL" means there is no function to implement.
   This time we will create a callback function named " **MyISR ".**

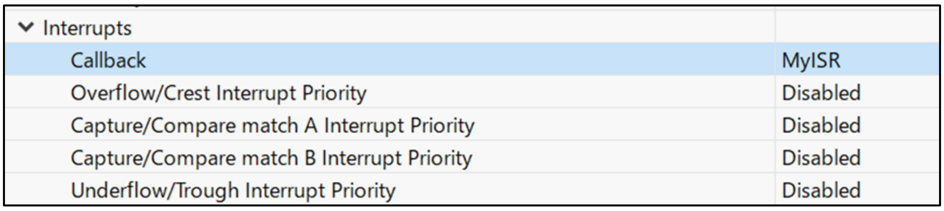

Figure 3-32 Setting the timer function

9. Set the interrupt priority in Overflow/Crest Interrupt Priority.
   The default "Disabled" means interrupts are disabled.
   Since we will be using interrupts this time, set the priority to a value between 0 and 15.

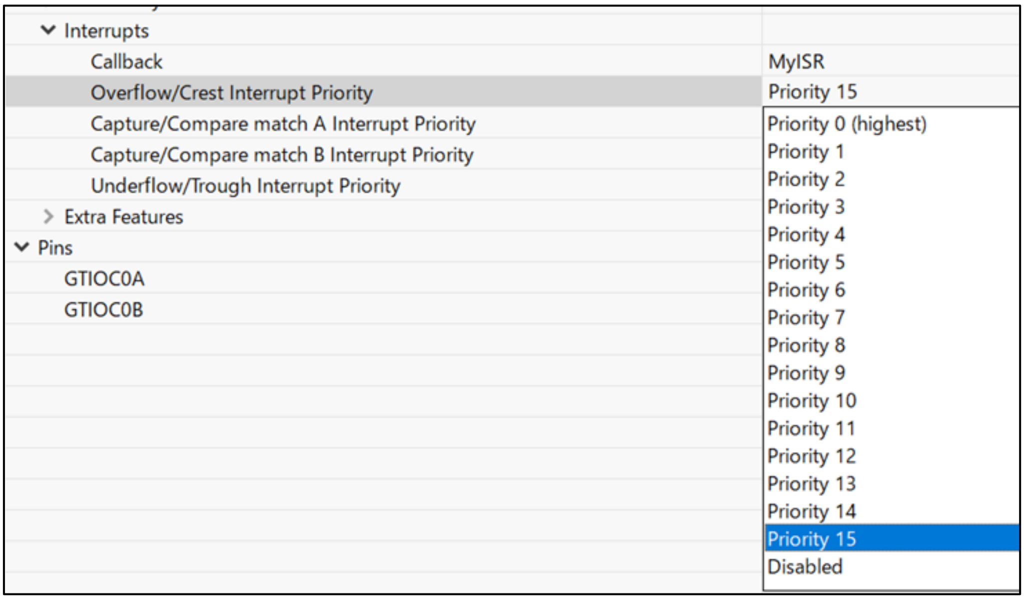

Figure 3-33 Setting the timer function

This completes the Timer settings.

10. After completing the necessary settings, press the "**Generate Project Content**" button to generate the code.

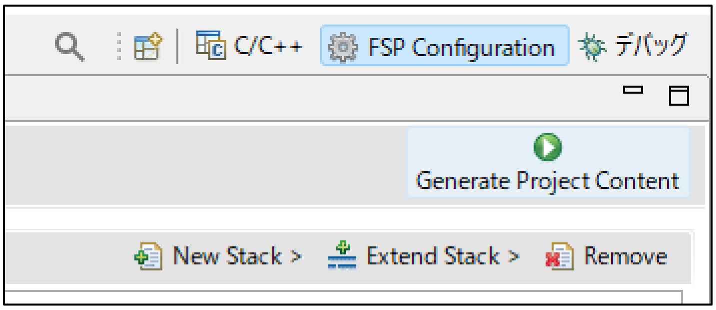

Figure 3-34 Generate Project Content

11. You will be asked if you want to save it to the Configuration.xml file, so click the "**Proceed**" button to save it.

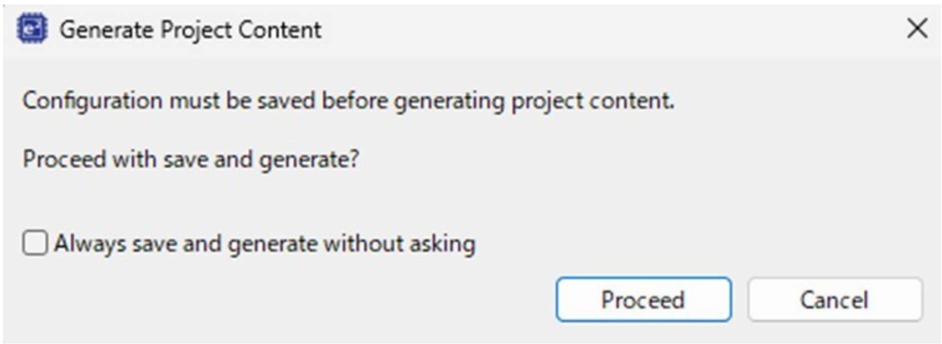

Figure 3-35 Generate Project Content

3.3 Coding

In this section we will introduce writing the application code.

The contents to be implemented are as follows.

・Main program

Starting the timer

・Interrupt program

LED toggling

3.3.1 Implementation of the main program

1. When developing using the FSP, the application entry point is in the src\hal_entry.c file.
   Double click this file to open it.

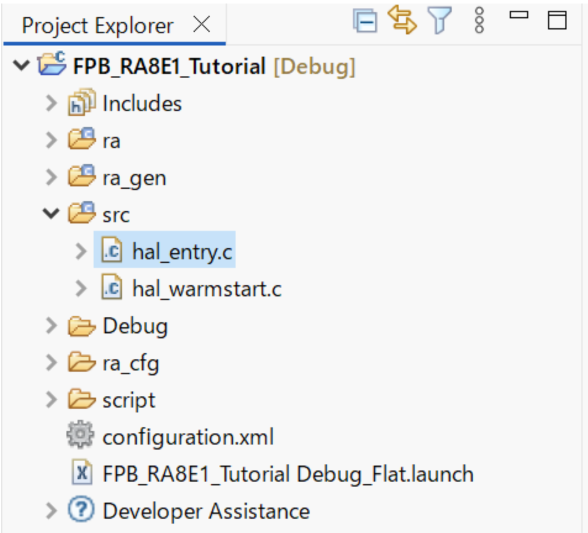

Figure 3-36 Implementation of the main program

2. The void hal_entry(void) function is the application entry point.
   The initial I/O settings and C runtime setup is already performed when reaching this point.
   The gray area on the screen indicates source code not being compiled due to preprocesor exclusion - thus it can be ignored.

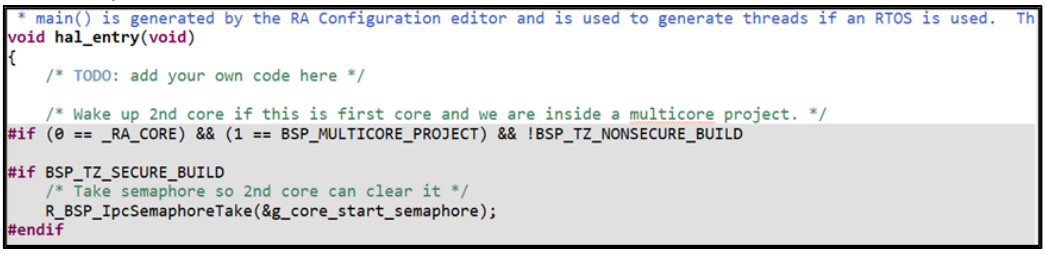

Figure 3-37 Implementation of the main program

3. Functions provided by the FSP return their execution result as a return value. To store this result, define a variable named err as follows:
   `fsp_err_t err;`

```
void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;
```

Figure 3-38 Implementation of the main program

4. Implement the timer open function.
   Open the "Developer Assistance" list in the "Project Explorer" window and you will see the timer module MyTimer, which was configured in "3.2.4 Timer Function Settings".
   Open the list further and you will see a list of functions.
   The function to open the timer is R_GPT_Open(). Drag and drop this function into the source file with the mouse.

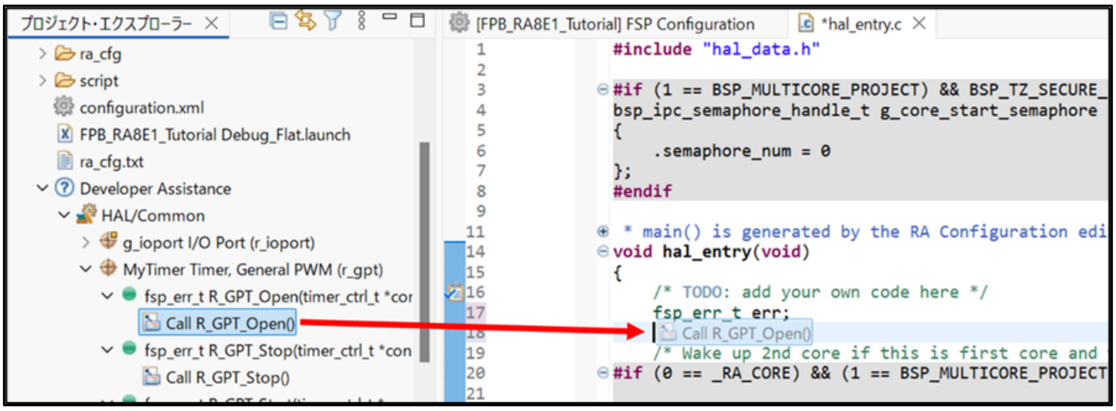

Figure 3-39 Implementation of the main program

5. Open API is implemented in the source code. By declaring a return value variable in advance, code containing assignment statements will be generated.
   Open API is used for peripheral setup, that is turning the peripheral on and making any one-time settings.

```
void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;
    err = R_GPT_Open(&MyTimer_ctrl, &MyTimer_cfg);
```

Figure 3-40 Implementation of the main program

If the return value of the R_GPT_Open() function is FSP_SUCCESS, it indicates **completion**. Details of the return values can be found in the following header file:

FPB_RA8E1_Tutorial > ra > fsp > inc > api > fsp_common_api.h

6. Implement the timer start function. The start API is R_GPT_Start ( ) .
   Drag and drop with your mouse below “R _GPT_ Open ()” in the source file.
   Start API is used to start peripheral operation, in this case it will begin the timers counting operation.

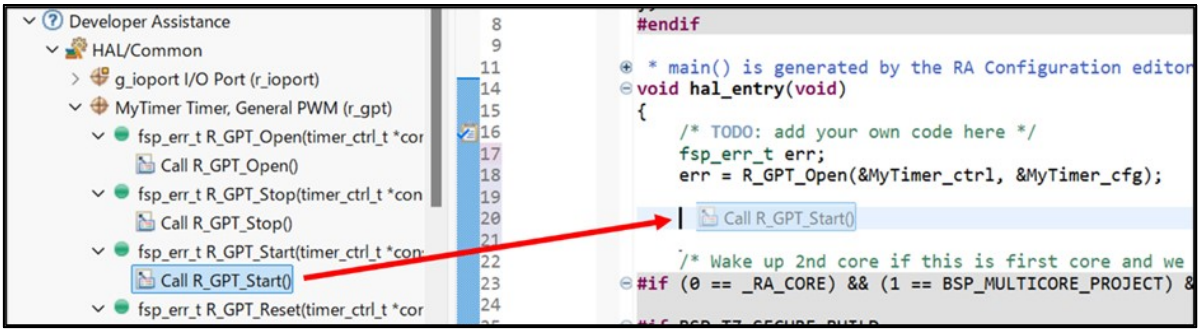

Figure 3-41 Implementation of the main program

7. A start function is implemented in the source code.

```
void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;
    err = R_GPT_Open(&MyTimer_ctrl, &MyTimer_cfg);

    err = R_GPT_Start(&MyTimer_ctrl);
```

Figure 3-42 Implementation of the main program

8. To determine run time errors, checking for non-zero returns and hanging the application is used to alert the developer of settings errors.

```
while(err);
```

If the program terminates abnormally, the above statement will loop infinitely.
Additionally upon successful execution, an infinite loop should be implemented.

```
while (1)
{
  __NOP();
}
```

The source code of the hal_entry() function is as follows.

```
void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;
    err = R_GPT_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);

    err = R_GPT_Start(&MyTimer_ctrl);
    while(err);

    while (1)
    {
        __NOP();
    }


#if BSP_TZ_SECURE_BUILD
    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif
}
```

Figure 3-43 Implementation of the main program

9. The whole code is as follows. (Excluding invalid codes in gray)

```
void hal_entry(void)
{
/* TODO: add your own code here */
fsp_err_t err;

err = R_GPT_Open(&MyTimer_ctrl, &MyTimer_cfg);
while(err);

err = R_GPT_Start(&MyTimer_ctrl);
while(err);

while(1)
{
__NOP();
}
}
```

3.3.2 Implementation of interrupt program

1. Implement the interrupts callback function. The implementation should be placed in src\hal_entry.c as in previous steps.
   Drag and drop the timer module's "**Callback function definition** " from the list of Developer Assistance onto the last line of the source file.

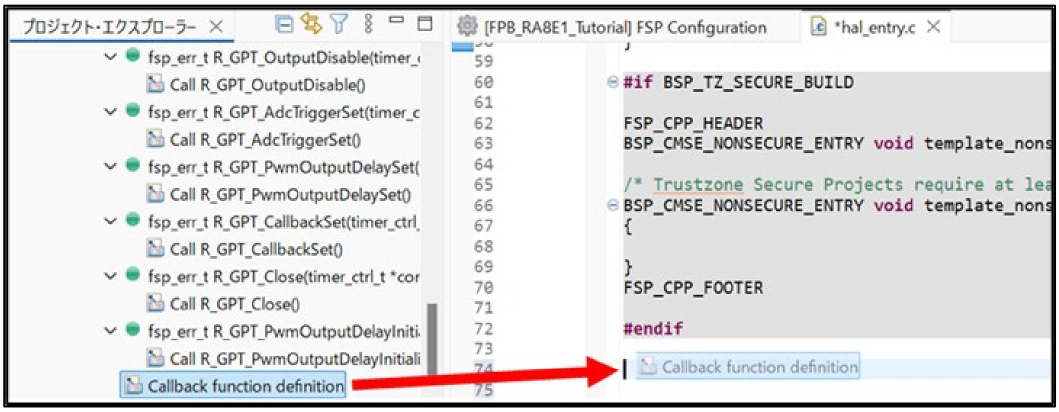

Figure 3-44 Implementation of interrupt program

2. The callback function name set in FSP will appear.
   This function is called every 500 ms, as configured in section **3.2.4 Timer Function Settings**.
   The argument “p_args” is automatically set by the FSP, so there is no need to assign a value manually.
   We will add the LED toggle output process within this function.

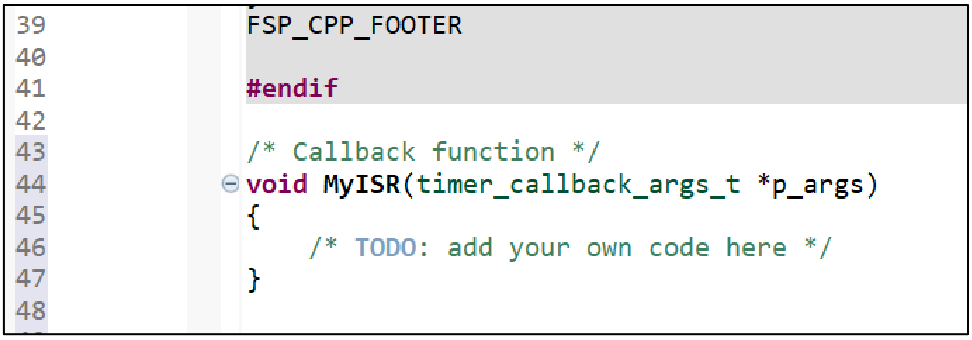

Figure 3-45 Implementation of interrupt program

3. Write the FSP return value type variable.
   `fsp_err_t status;`

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    fsp_err_t status;

}
```

Figure 3-46 Implementation of interrupt program

4. Read the current output state of the LED.
   The pin read function is R_IOPORT_PinRead ( ).
   Drag and drop the IO port module R_IOPORT_PinRead ( ) from the Developer Assistance list into the interrupt function body.

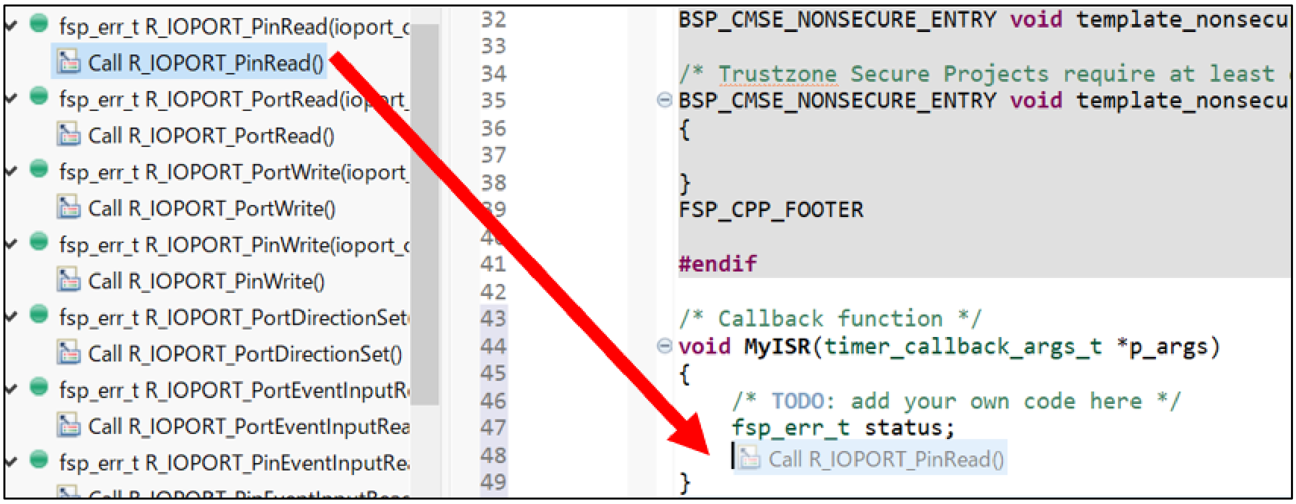

Figure 3-47 Implementation of interrupt program

5. The result should look as follows:

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    fsp_err_t status;
    status = R_IOPORT_PinRead(&g_ioport_ctrl, pin, p_pin_value);

}
```

Figure 3-48 Implementation of interrupt program

6. Above the R_IOPORT_PinRead() line, provide a definition for the BSP IO level type, like so:

```
bsp_io_level_t value;
```

7. The 2nd argument of R_IOPORT_PinRead ( ) is the pin number to read = “ LED1 ” ,
   Set the address of the variable (bsp_io_level_t value) that stores the read result in the 3rd argument .

   Write the following:

```
status = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
```

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    fsp_err_t status;
    bsp_io_level_t value;
    status = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);

}
```

Figure 3-49 Implementation of interrupt program

8. The symbol “LED1” is declared in ra_cfg\fsp_cfg\bsp\bsp_pin_cfg.h in the project tree and can be referenced from hal_entry.c.

```
#define CAM_XCLK (BSP_IO_PORT_04_PIN_03)
#define LED1 (BSP_IO_PORT_04_PIN_04)
#define CAM_D2 (BSP_IO_PORT_04_PIN_05)
```

Figure 3-50 Implementation of interrupt program

9. Stores the value for inverting the read value and outputting it in the variable next_led1.
   Write the following.

```
bsp_io_level_t next_led1 ;
```

Write the process to invert and output the read value.

```
while(status);
if(BSP_IO_LEVEL_HIGH == value)
{
next_led1 = BSP_IO_LEVEL_LOW;
}
else
{
next_led1 = BSP_IO_LEVEL_HIGH;
}
```

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    fsp_err_t status;
    bsp_io_level_t value;
    bsp_io_level_t next_led1;
    status = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
    while(status);
       if(BSP_IO_LEVEL_HIGH == value)
       {
           next_led1 = BSP_IO_LEVEL_LOW;
       }
       else
       {
           next_led1 = BSP_IO_LEVEL_HIGH;
       }

}
```

Figure 3-51 Implementation of interrupt program

10. Add processing to write LED1 state
    The pin write function is R_IOPORT_PinWrite( ).
    Drag and drop the IO port module R_IOPORT_PinWrite( ) into the interrupt function.

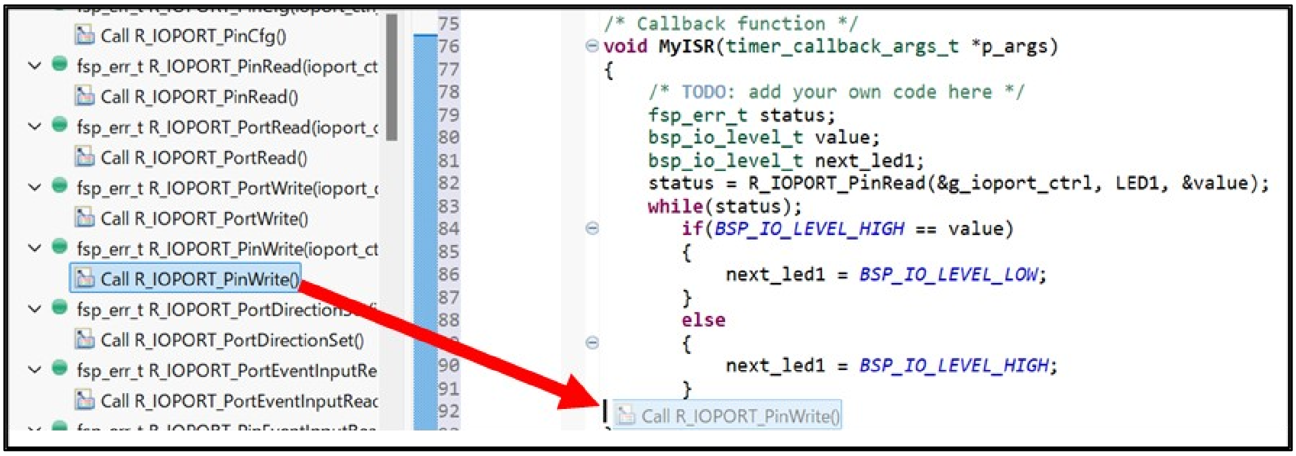

Figure 3-52 Implementation of interrupt program

11. The result is as follows:

```
else
{
    next_led1 = BSP_IO_LEVEL_HIGH;
}

status = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
```

Figure 3-53 Implementation of interrupt program

12. Set " LED1_G1 " to the 2nd argument of R_IOPORT_Pin Write( ) and the output value (=next_led1) to the 3rd argument .
    Write the following.

```
status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1 , next_led1 );
while(status);
```

```
status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, next_led1);
while(status);
```

Figure 3-54 Implementation of interrupt program

13. Write the process to output the previous value of LED1 to LED2 as follows.

```
status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, value );
```

14. The implementation of the interrupt function is as follows.

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    fsp_err_t status;
    bsp_io_level_t value;
    bsp_io_level_t next_led1;
    status = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
    while(status);
       if(BSP_IO_LEVEL_HIGH == value)
       {
           next_led1 = BSP_IO_LEVEL_LOW;
       }
       else
       {
           next_led1 = BSP_IO_LEVEL_HIGH;
       }
       status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, next_led1);
       while(status);

       status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, value);
       while(status);

}
```

Figure 3-55 Implementation of interrupt program

15. The whole code is as follows.

```
/* Callback function */
void MyISR(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */

    fsp_err_t status;

    bsp_io_level_t value;

    bsp_io_level_t next_led1;

    status = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);

    while(status);

    if(BSP_IO_LEVEL_HIGH == value)
    {
        next_led1 = BSP_IO_LEVEL_LOW;
    }
    else
    {
        next_led1 = BSP_IO_LEVEL_HIGH;
    }

    status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, next_led1);
    while(status);

    status = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, value);
    while(status);
}
```

3.4 Build

The following steps describe building the application executable in preparation for debugging.

Right-click the project name in the project tree and select Build Project.

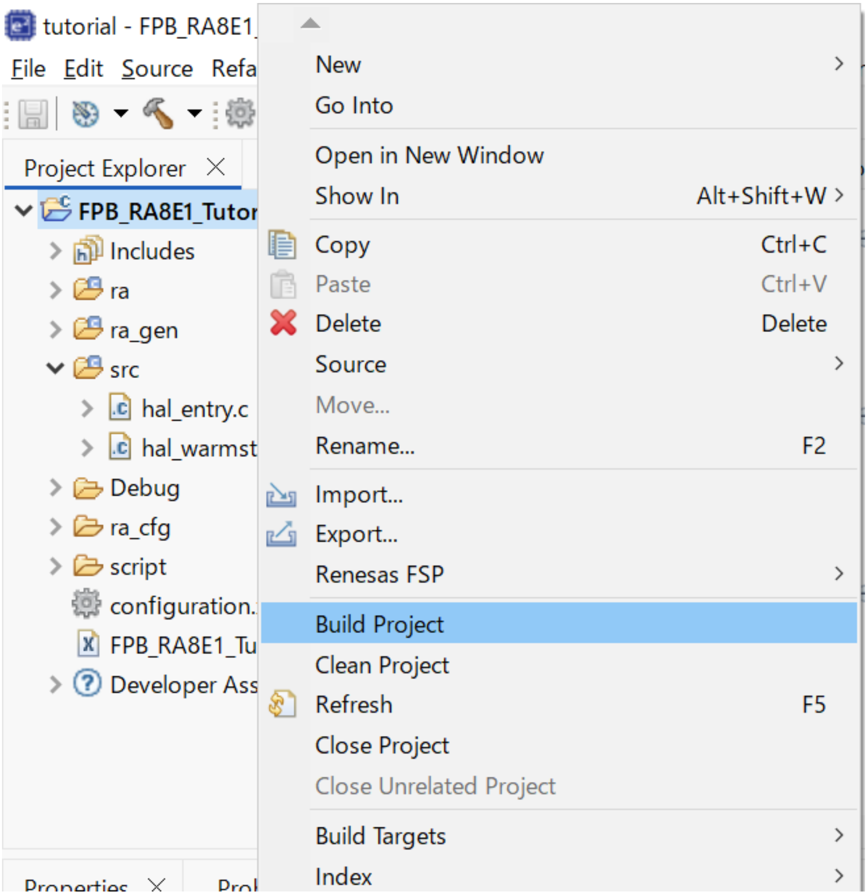

Figure 3-56 Build

Or you can build it by clicking the icon below.

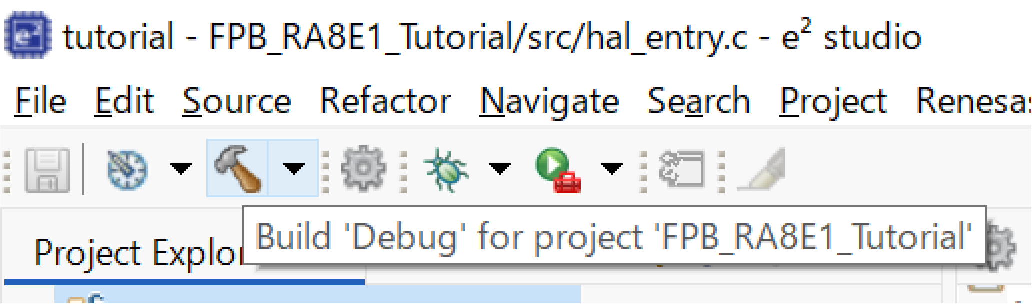

Figure 3-57 Build

1. The build log will be output to the " Console" window.
   "Build Finished. 0 errors," is displayed at the end, it means that the build completed successfully. Yellow bands indicate warnings detected by the compiler. Check the contents of the warning and correct it if necessary.

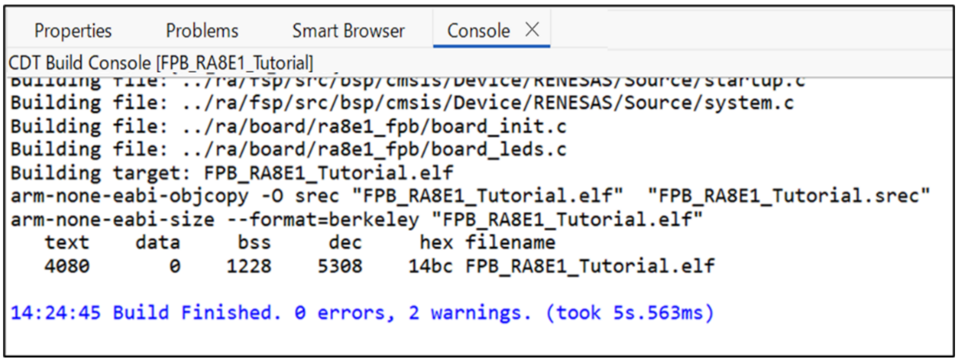

Figure 3-58 Build

If there is any coding mistake, a red band will appear. Please check the relevant line and revise the source code.

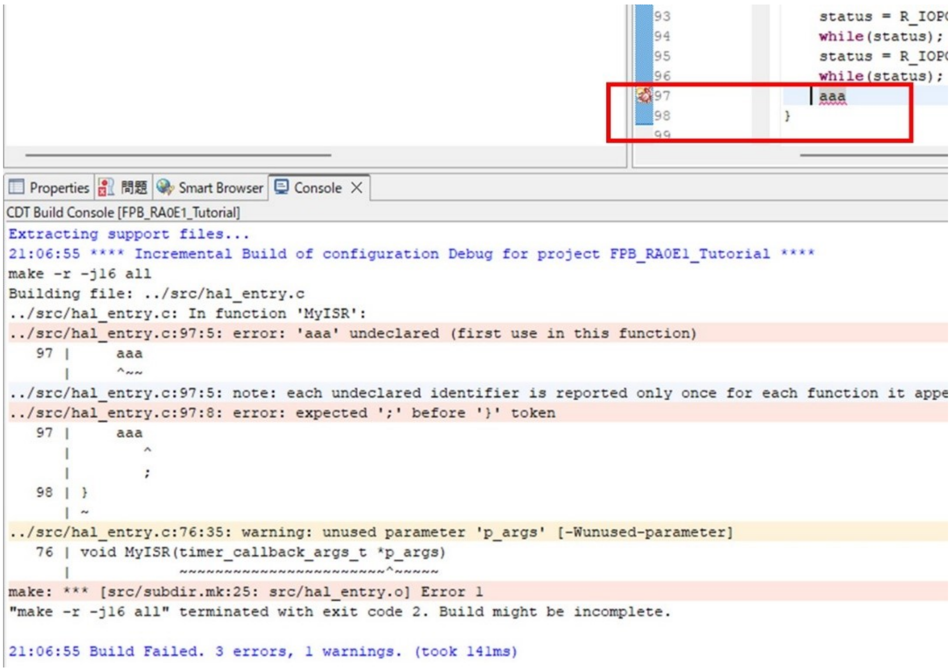

Figure 3-59 Build

4. How to debug

This chapter explains the settings required to run the program.

4.1 Debug settings and startup

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

For the first time only, check the settings for writing.

Connect the PC and FPB board with a USB cable.

1. Right-click on the project name in the Project Tree, then select "**Debug As**" → "**Debug Configurations**".

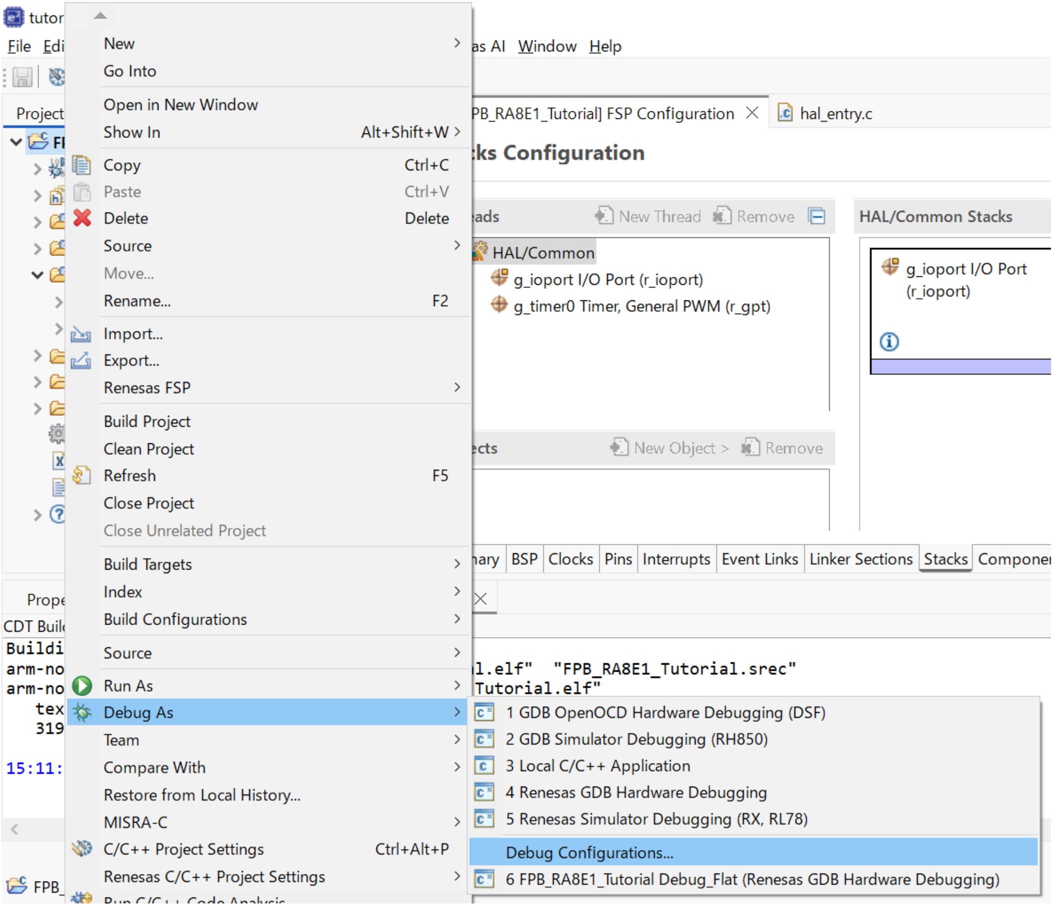

Figure 4-1 Debug settings and startup

2. The configuration screen will be displayed. Select Renesas GDB from the tree on the left. Select your project name under Hardware Debugging.

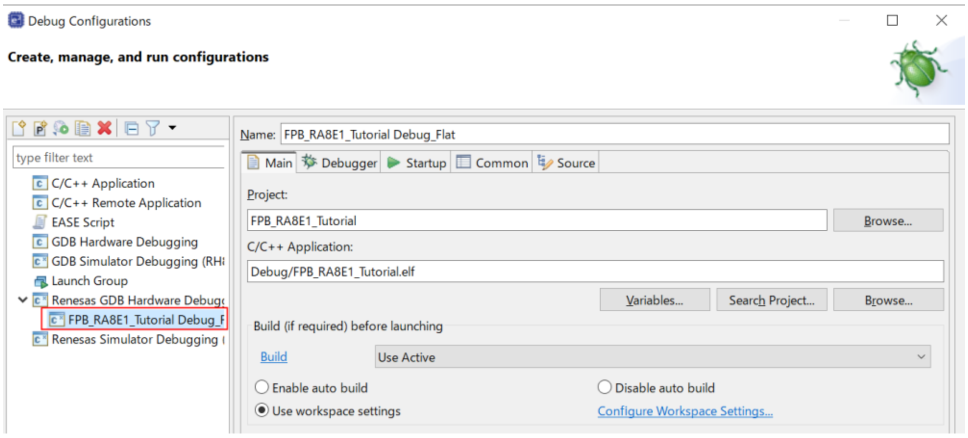

Figure 4-2 Debug settings and startup

3. Select the Debugger tab and please confirm the following settings:
   Debug hardware = “J-link ARM”
   Target Device = " R7FA8E1AF "

   Press the "Debug" button to start writing the program to the microcontroller.

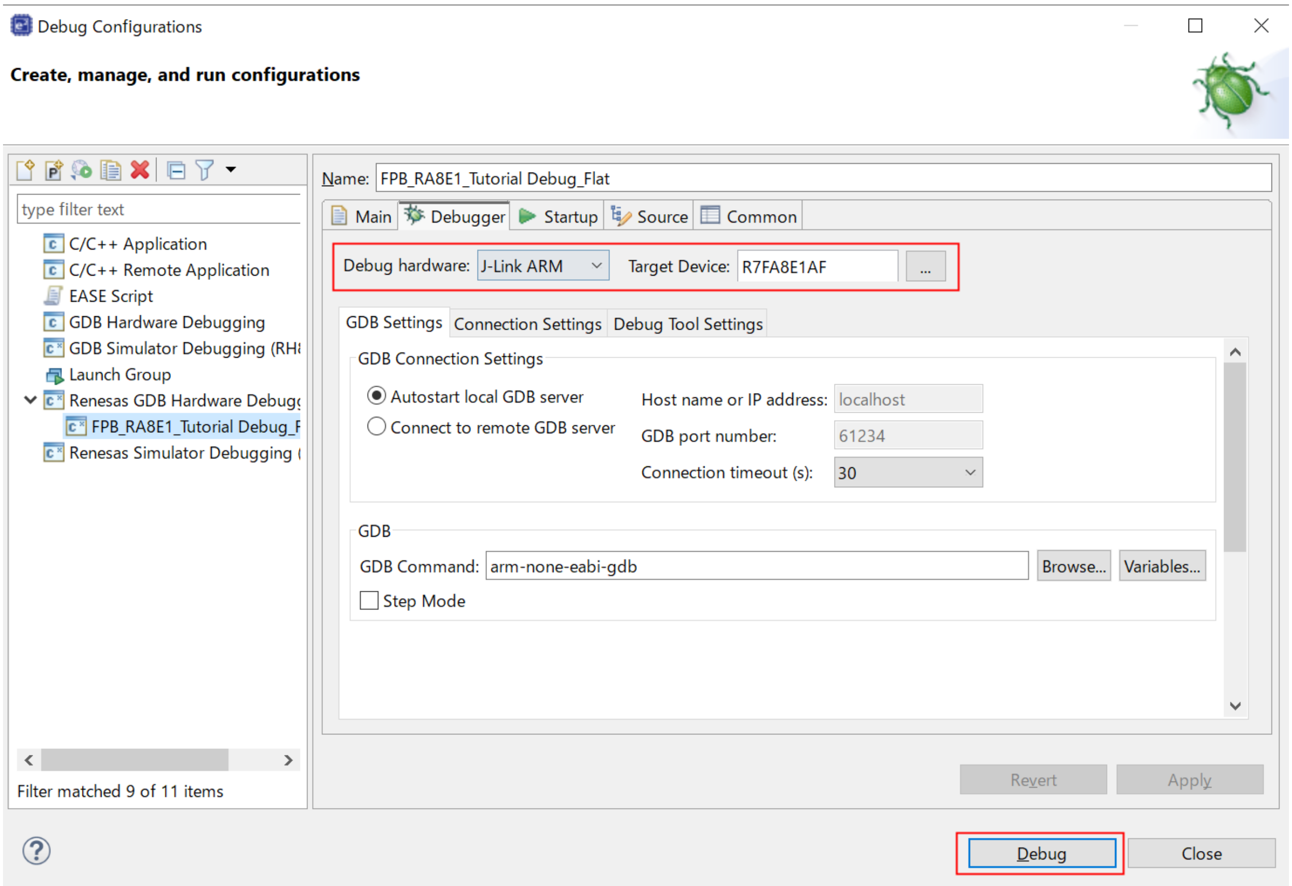

Figure 4-3 Debug settings and startup

4. After flashing is complete a pop-up will appear to ask if you would like to switch perspective to the "Debug" perspective.
   This operation optimizes the debug workflow and for this tutorial you should click "**Switch**".

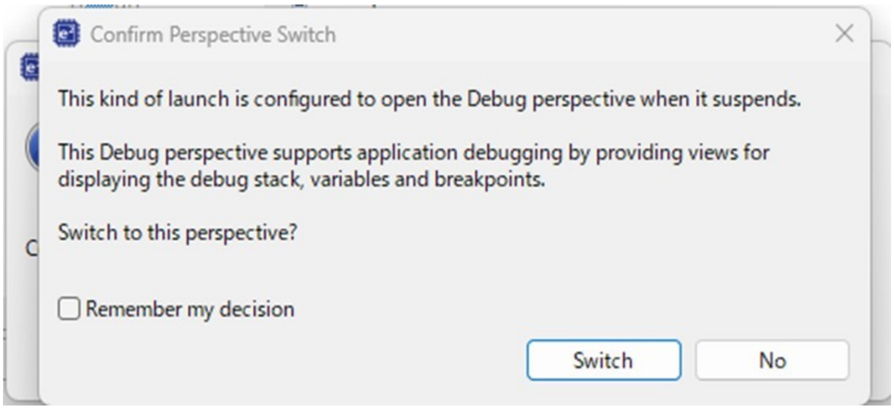

Figure 4-4 Debug settings and startup

*Even if you press "No", you can switch the screen later by pressing the "Debug（デバッグ）" button at the top right of e² studio.

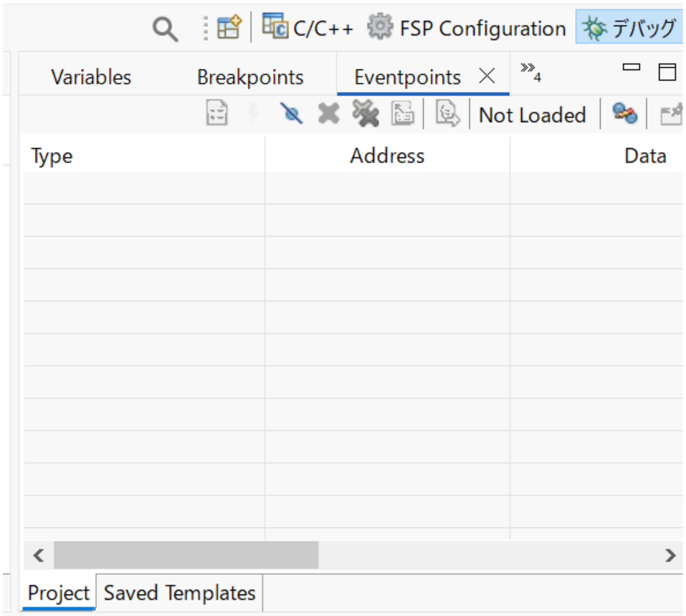

Figure 4-5 Debug settings and startup

5. When the program finishes writing, it pauses at the SystemInit () function in startup.c.

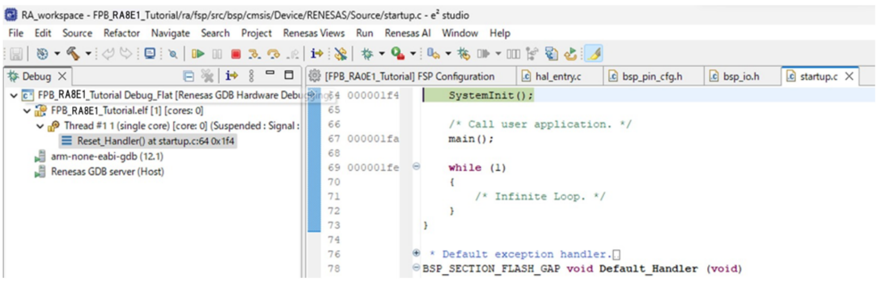

Figure 4-6 Debug settings and startup

4.2 Execution

In this state, the MCU has been reset and the program is not yet running.

You can confirm that neither LED1 nor LED2 on the board are lit.

1. Run the program by pressing the resume button in the red frame or the F8 key on your keyboard.

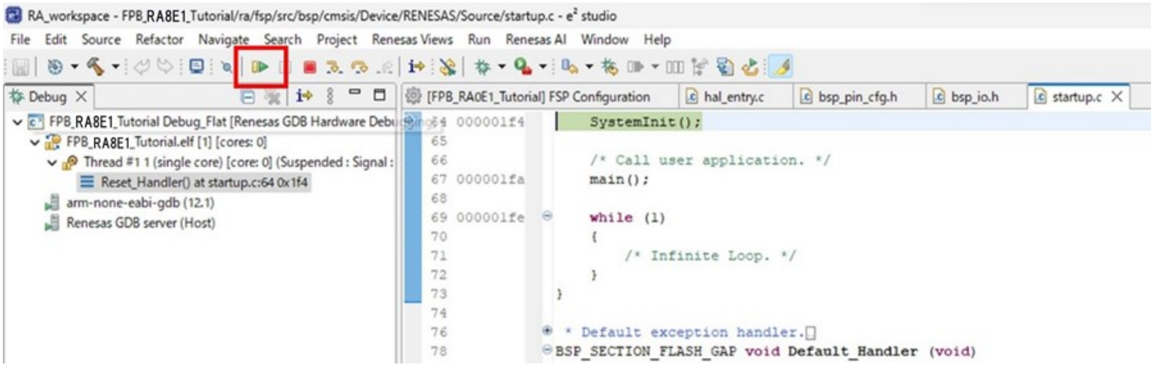

Figure 4-7 Execution

The program pauses at the beginning of the main function . (This is because the e² studio project default is set to pause at the main function.)

In this state, SystemInit ( ) has been completed and the initial settings have been completed.
If you check the board, you will see that LED1 is lit.

2. Press the resume button again to continue the program.

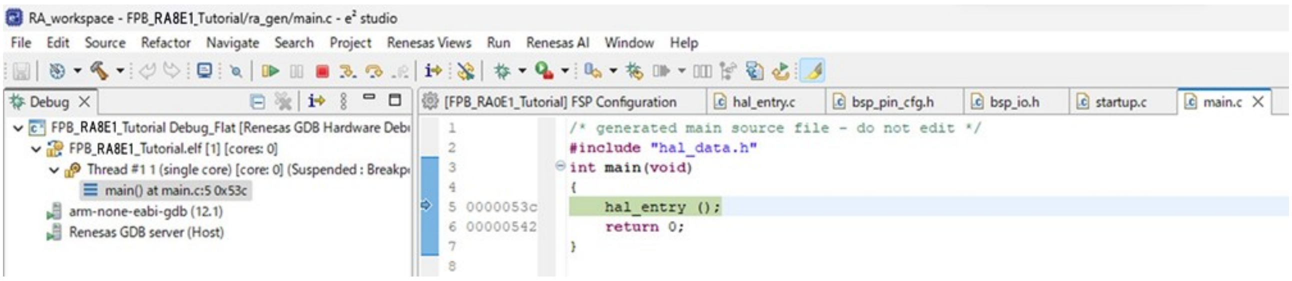

Figure 4-8 Execution

The execution status will be displayed at the bottom left of e² studio. When " Running " is displayed, the program is running.

3. If you check the board, you can see that LED1 and LED2 are lit alternately every 500ms.

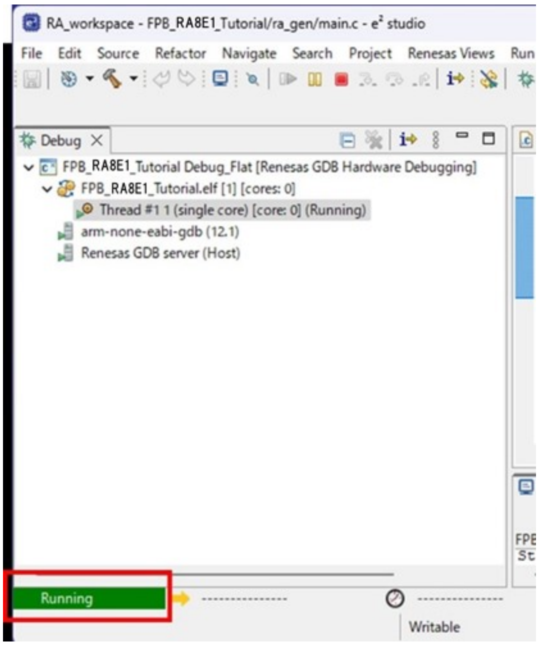

Figure 4-9 Execution

4.3 Quit debugging and restart

1. If you want to end debugging, press the " Terminate" button.

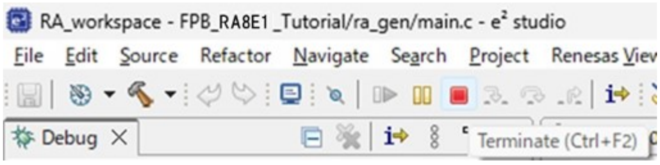

Figure 4-10 Quit debugging

2. The "Project Explorer" window is not displayed in the specified position because we are in the "Debug" perspective.
   To return to the C/C++ Perspective and navigate the project, click “C/C++” on the top right of e2 studio.

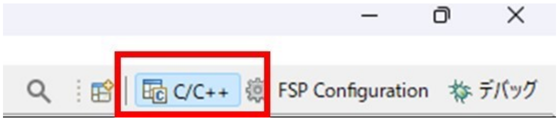

Figure 4-11 Quit debugging

3. If you debug again, your debug settings are remembered.
   You can start debugging by [Right clicking] the project → “Debug As” → “6 *Project Name*”.

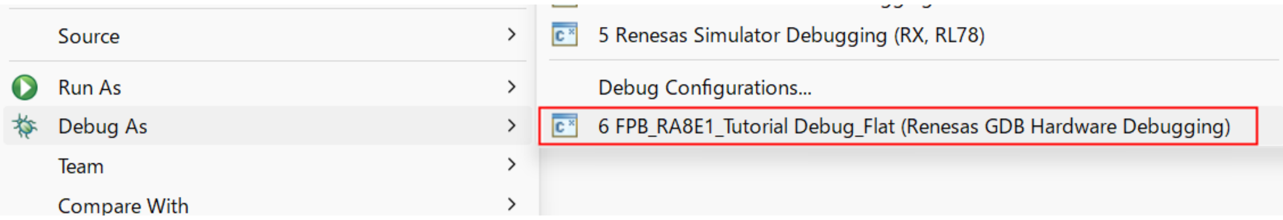

Figure 4-12 Quit debugging

Alternatively, you can start debugging by clicking the icon below.

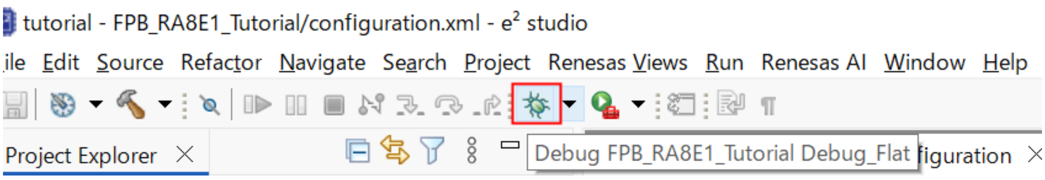

Figure 4-13

Revision History

Rev.	Date	Description	
		page	Summary
1.00	2025.10.27	-	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

Trademarks

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/.