

Renesas RA Family

Low-Level Register Programming Guide for RA MCUs

Introduction

There are scenarios where it is desirable to design custom drivers for RA-MCU's modules rather than adding and utilizing the Renesas Flexible Software Package (FSP) drivers.

To avoid adding the FSP HAL driver code files, you need to design custom hardware drivers for each of the peripherals you wish to use in your end application. The drivers write to and read from the peripheral hardware registers directly to achieve the desired functionality. The drivers must be designed to follow the peripherals' register specifications as outlined in the respective peripheral chapter of the RA MCU's Hardware User's Manual.

This application note serves as a programming guide for designing register-level functions based on the specifications in RA MCU's Hardware User's Manual. The accompanying application projects run on the FPB-RA0E1 device and together demonstrate the execution of the three-step approach to creating a custom driver application as outlined in this document.

Target Device

Application Note: Low Level Register Programming Guide – All RA MCUs

Application Project: Low Power Application – RA0E1

Project Requirements

Hardware:

- FPB-RA0E1
- DC Signal Generator from 0-3.3V

Software:

- e² studio v2024-10
- Flexible Software Package (FSP) v5.7.0
- GCC ARM Embedded Toolchain v13.2.1.arm-13-7
- TeraTerm or similar serial terminal console

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e² studio IDE and RA Family Flexible Software Package (FSP).

It intends to guide an advanced embedded programmer who already knows their intended application use case and target MCU. Before reading this application note you should read the RA Quick Design Guide for your MCU and thoroughly review the relevant chapters of the RA MCU Group Hardware User's Manual.

The low-level programming concepts from this application note can be applied to all RA MCUs, but it is most relevant to lighter devices from the RA0 and RA2 MCU Groups.

Note that the accompanying project demonstrates applying the guide for the RA0E1 device. Hardware and software available will differ across devices.

Contents

1. Low-Level Register Programming.....	4
1.1 The Three Step Workflow.....	4
2. Step 1: Create an FSP Reference Project	4
2.1 Creating a New FSP Project	5
2.1.1 Initial Project Organization	6
2.1.2 Change a Project's Element Version	6
2.2 Add FSP Components of Interest.....	7
2.2.1 Locate Driver in Stacks Tab	8
2.2.2 Change a Module's Driver Version Using Components Tab	9
2.2.3 Configure Modules Using the Stacks Tab	10
2.3 Clocks.....	11
2.3.1 Configure Clocks Using the Clocks Tab.....	11
2.3.2 Configure Clocks Using BSP API.....	11
2.4 Configure Peripheral Interrupts	12
2.4.1 Interrupt Setup and Control Using the FSP.....	12
2.4.2 Interrupt Setup and Control Using the BSP	14
2.5 The Project's Linker Script	15
2.6 Tool Highlight: Developer Assistance Examples.....	16
3. Step 2: Create Custom HAL Drivers	17
3.1 Accessing a Module's Special Function Registers (SFRs).....	18
3.1.1 Key Components in I/O Define Header File	19
3.1.2 Peripheral Register Template from the I/O Define File	21
3.1.3 Tool Highlight: CDT Include Browser	22
3.2 Writing Custom HAL Drivers	23
3.2.1 Tracing the FSP Project's Program Flow	23
3.2.2 General Tips when Writing Custom Drivers:	24
3.2.3 Tool Highlight: e ² studio Autocomplete for Register Access	24
3.2.4 Tool Highlight: Editor Hover	25
3.2.5 Tool Highlight: Debugging with I/O Registers View.....	26
4. Step 3: Create Custom Low-Level e ² studio Project.....	26
4.1 Overview of Generic LLRP Project.....	27
4.1.1 BSP Warm Start	28
4.1.2 MAIN program	30
4.2 Overview of Low Power Mode Project	31
4.3 Create Base Project	31
4.3.1 Edit the BSP Properties.....	33
4.4 Add the Custom Drivers	34
4.4.1 32-bit Interval Timer (TML).....	35

4.4.2	12-bit A/D Converter (ADC_D).....	36
4.4.3	Low Power Modes (LPM).....	37
4.4.4	External IRQ (ICU).....	38
4.4.5	Serial Interface UART (UARTA).....	38
4.4.6	Useful BSP APIs.....	40
4.5	Configure Clocks Using HAL Registers.....	41
4.6	Add Interrupts Using NVIC.....	42
5.	Running the Low Power Mode Application Projects.....	44
5.1	Script for Low Power Mode Debugging.....	44
5.2	Project Requirements.....	45
5.3	Build, Download, and Run the Project.....	45
5.4	Verifying the Low Power Mode Application.....	46
6.	References.....	48
	Revision History.....	50

1. Low-Level Register Programming

Renesas Advanced (RA) MCUs are supported by the Renesas Flexible Software Package (FSP), an optimized software package that provides drivers for the RA MCU's hardware and software modules. The FSP API HAL drivers are designed to completely cover a given module's common functionalities while minimizing the memory footprint as much as possible. Since the FSP code is tested, scalable, well-documented, and designed to meet common use cases in embedded systems, embedded developers can quickly and confidently build versatile applications when using the FSP.

It is not a requirement for RA MCU application developers to use the FSP driver code in their end applications. There are cases where it is more desirable for a developer to create and use custom-designed API drivers in place of the FSP API drivers. In some cases, custom drivers:

- Provide specialized module routines for the general FSP drivers.
- Utilize chip functionality that is not yet supported by the FSP.
- Enable code to align with internal code bases rather than relying on vendor code.

Custom drivers provide flexibility to design specialized module routines tailored for the end application's use case. In very rare cases, some users may wish to utilize chip functionality that is not yet supported by the FSP with custom drivers.

Ultimately the decision to write custom code is a tradeoff of memory, flexibility, and development time and should be made only after careful consideration of the desired system and end application.

This document serves as a guide for writing custom drivers for RA MCU modules with a technique referred to as "low-level register programming". The guide outlines a three-step workflow for creating a custom driver application in e² studio and offers insight into best practices. Primarily, low level register programming is best suited for RA0 and RA2 MCUs.

The accompanying low power example code is written for the FPB-RA0E1, so the direct examples in the document illustrate the RA0 device hardware and software. But the methods described in this document to create the application can be generalized and extended to apply to any RA MCU.

1.1 The Three Step Workflow

Developing custom drivers for Renesas RA MCUs involves a structured approach that balances efficiency, flexibility, and reliability. This section introduces the three-step workflow designed to guide developers through creating tailored drivers for RA MCUs:

1. Create an FSP based reference project.
2. Develop custom driver code for each hardware and software module required.
3. Make a minimal BSP project and integrate custom drivers to create the end application.

2. Step 1: Create an FSP Reference Project

It is highly recommended to first create an FSP project as a reference before writing custom driver code and attempting to make a full custom application. An FSP reference application allows the developer to:

- Quickly create a minimal Proof of Concept project.
- Identify differences in functionality provided out-of-box by the BSP vs FSP.
- Analyze the generated HAL drivers and configurations. The FSP HAL drivers are developed after a thorough study of the MCU Hardware User's Manual, are tested frequently, and are updated for bug fixes throughout MCU lifetime.
- Identify the order of operations performed with special function registers by analyzing the FSP API call flow.
- Understand the configuration for modules of interest and identify any incorrect settings easily with the GUI warnings.
- Have a comparison for operation when debugging their custom HAL drivers.

This is a non-exhaustive list, but in all, creating an FSP reference project greatly minimizes the time and effort required to create and test the custom driver project. And it is critical to note that Renesas will support the FSP drivers but there is NO support for any custom instances of drivers, including the ones in the accompanying example application.

2.1 Creating a New FSP Project

To create a minimal project, first follow the New Project creation in e² studio for an RA MCU, to set up the project name, toolchain, device, FSP packs, debugger, and optionally choose a project template.

Once the project is created through the set-up menu wizard, the project file structure is created in the workspace in the form of a minimal Board Support Package (BSP) project. From here, developers should be familiar with the procedure of using the FSP Configuration View GUI to edit the configuration.xml file to add and configure modules, pins, clocks, interrupts, and RTOS resources.

The BSP is a subset of the FSP and the BSP provides an interface for setting up FSP to work with selected MCUs and specific board hardware designs. It is both MCU and device specific. The BSPs for the Renesas RA kits are included in FSP packs, and the FSP packs are automatically included in an e² studio download.

The BSP is responsible for getting the MCU from reset to the user's application, and it offers a set of support functions. Before reaching the main user's application, the BSP sets up the stacks, heaps, clocks, interrupts, C runtime environment, and stack monitor. Any BSP functions required by the custom drivers for any given functionality will be referenced later in this document where applicable

The block diagram below depicts the relationship between the BSP in the bottom block and the FSP which encompasses the rest.

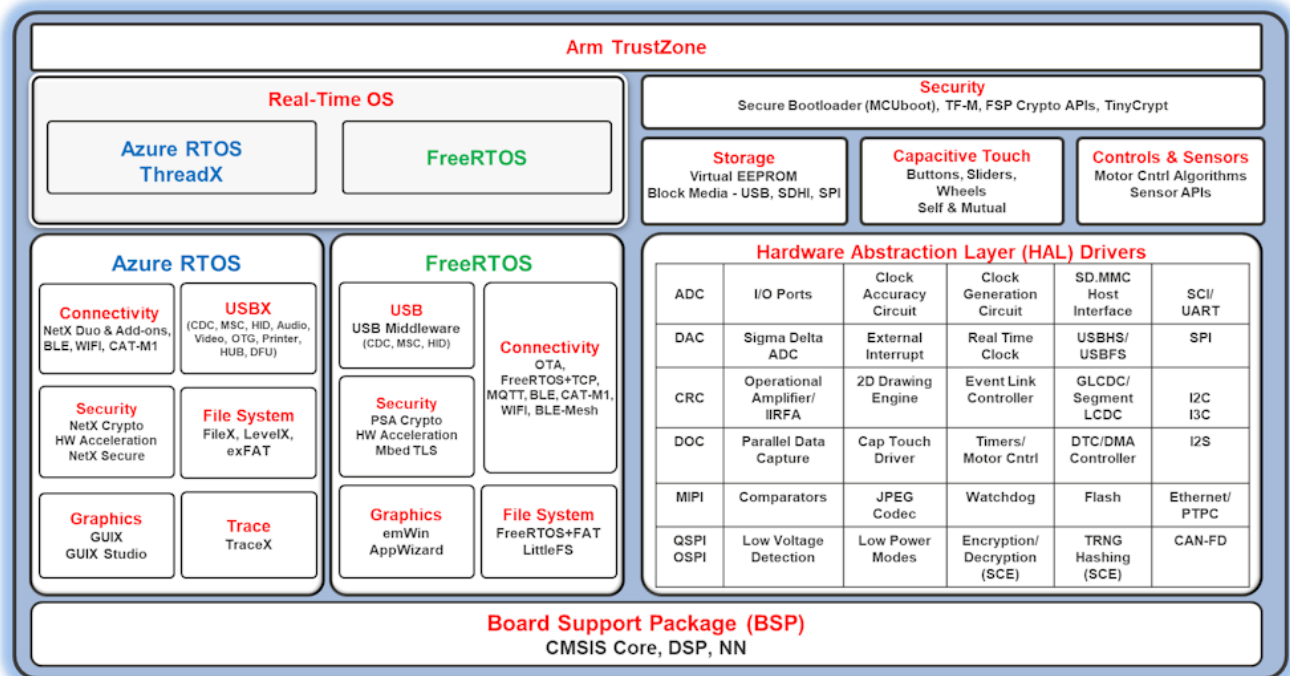


Figure 1. Block Diagram of the FSP and BSP Relationship

2.1.1 Initial Project Organization

The image below illustrates the higher-level workspace folder structure of a newly created e² studio project:

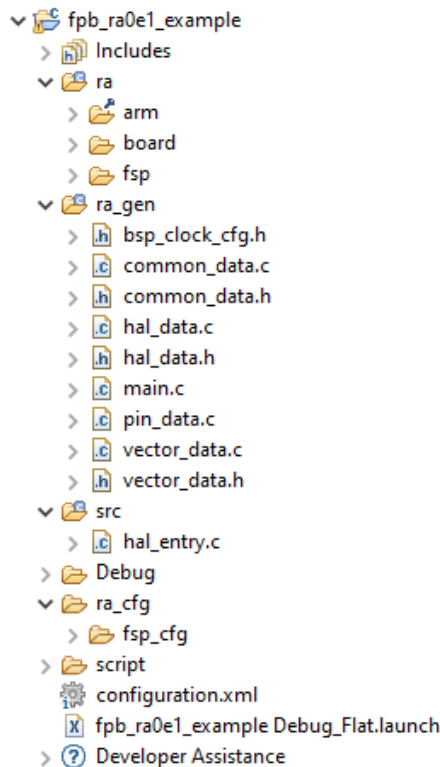


Figure 2. High-level Workspace Folder Organization of a Newly Created RA C/C++ Project

The **ra** folder contains core files for ARM CMSIS, the board's hardware, and for the FSP. The **ra/fsp** folder is further split into the **source** and **include** folders, kept separate for easy browsing and setting up of the include paths.

The **ra_gen** folder contains code generated by the RA Configuration editor. This includes global variables for the control structure and configuration structure for each module.

The **ra_cfg** folder is where configuration header files are stored for each module.

The **src** folder is typically where the application code will be created and stored. The "**hal_entry.c**" file contains the entry point for user code.

2.1.2 Change a Project's Element Version

Recall that the FSP version was initially selected during e² studio's New Project creation wizard. The **Summary tab** shown below gives a high-level overview of the project, showing key elements like the board, device, toolchain, and FSP version being used.

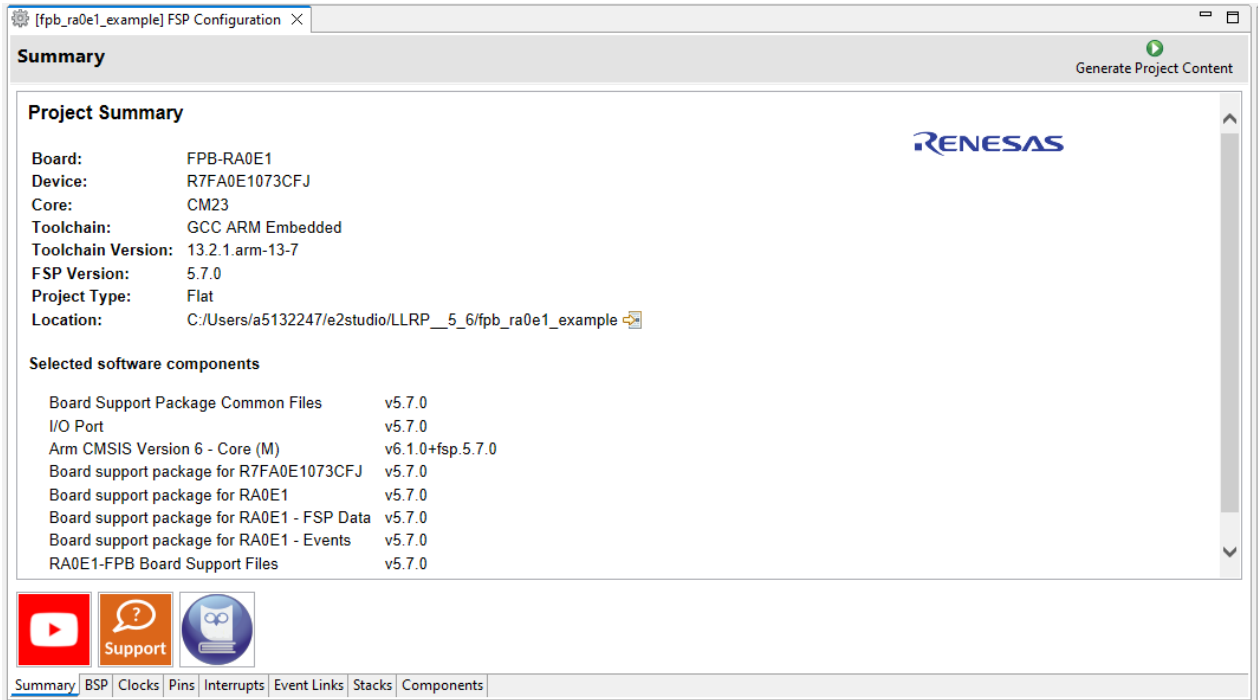


Figure 3. Summary Tab for Project Targeting the FPB-RA0E1 and FSP v5.7.0

The **BSP** tab shown below is used to configure the Board Support Package options for the project. Developers can reconfigure the board and FSP version in the main window, and using the **Properties View** window can modify the available board properties based on the project requirements as needed.

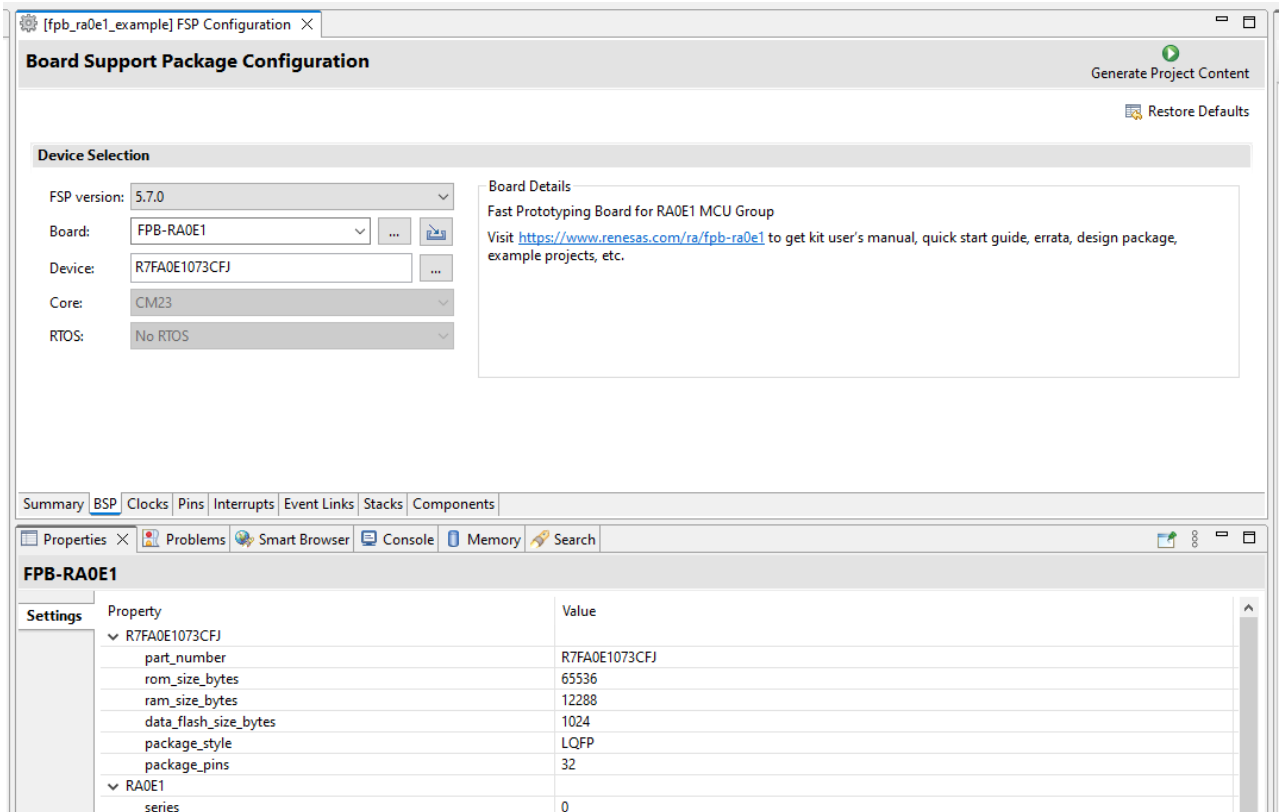


Figure 4. Example View of the BSP Tab and Property View Setting Options for the FPB-RA0E1

2.2 Add FSP Components of Interest

This section covers adding and configuring the desired FSP modules for an e² studio project, detailing the typical method using the FSP Configuration’s GUI interface.

When creating embedded systems solutions, developers typically first choose which MCU device will best serve their application, based on the hardware and software features of the board. The Hardware User Manual contains the complete hardware specifications and operation notes for every module on the MCU, and the RA FSP Documentation details the HAL and BSP API and software specifications for each module that has FSP driver support.

This section provides an overview and key considerations for each topic discussed, but it is not meant to replace the Renesas Flexible Software Package (FSP) User’s Manual. Refer to this manual for more complete information.

2.2.1 Locate Driver in Stacks Tab

After identifying the appropriate RA MCU to run the target application and establishing the minimal base project, it’s time to begin adding the modules of interest and their underlying FSP drivers to the project. Open the **configuration.xml** file using e2 studio’s **FSP Configuration View** to access the GUI interface, and navigate to the **Stacks** tab.

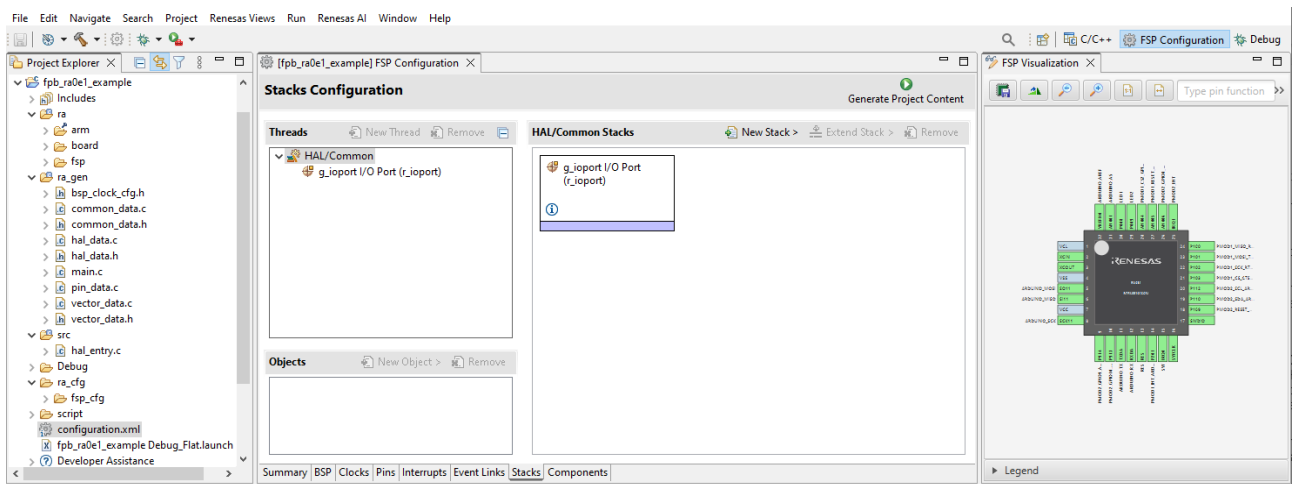


Figure 5. Add HAL Stacks by Opening the configuration.xml File with the FSP Configuration View

The **Stacks** tab in the RA Configuration editor allows developers to add and configure threads, add and manage FSP modules and RTOS objects within each thread, and edit the module and RTOS object properties.

To quickly add a module to the current thread, click on the **New Stack** button and utilize the search functionality for the module of interest. Click on the desired module from the results list to add the module to the project.

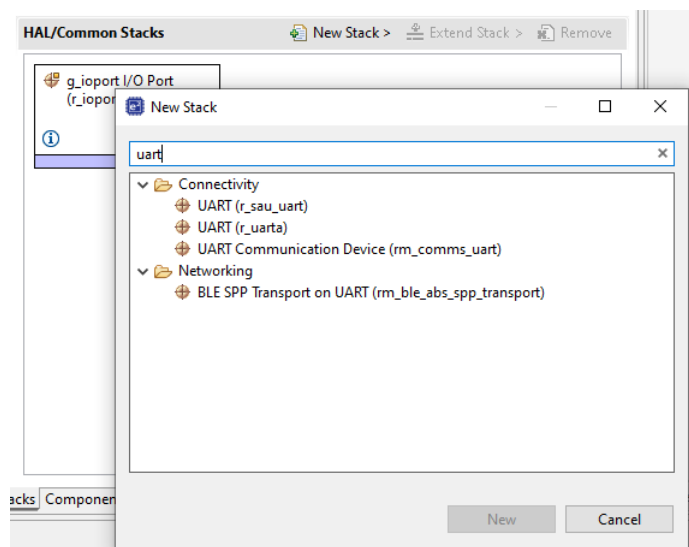


Figure 6. Example of Searching for the UART Stack to Add to an FPB-RA0E1 Project

2.2.2 Change a Module's Driver Version Using Components Tab

There are some occasions where it's desirable to change the FSP driver version to a specific version for a particular module or stack that has been added to a project. Mixing FSP versions in a single project is not typically required or encouraged, but there are a few scenarios where it may make sense:

- **Compatibility:** If you are working with a legacy project or have specific compatibility requirements, you might need a particular version of a module that is different than the one automatically selected.
- **Testing and Debugging:** For testing purposes, you may want to revert to an older version of a module to verify if a bug was introduced in a newer version.
- **Specific Features:** A specific module version may have some features that are not available in other versions. If the application depends on those features, you may want to switch and use the specific version.

The **Components tab** in the RA Configuration editor provides a way to manage the individual modules included in a project, which are listed in a categorized manner.

Each module's available versions listed here relates to the FSP Packs that have been installed into that instance of e² studio. Additional FSP Packs can be searched for and downloaded from the Renesas FSP Github repo: github.com/renesas/fsp

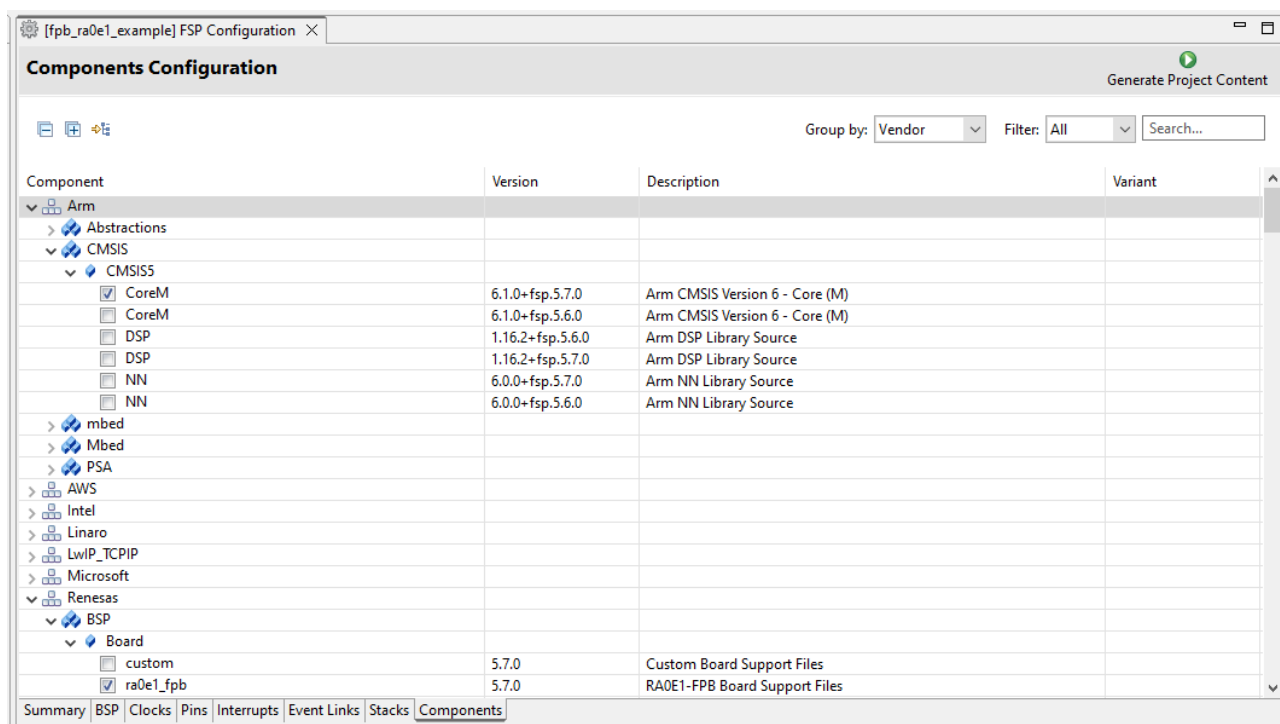


Figure 7. Components Tab Overview of FSP v5.7.0 Project When FSP v5.6.0 Packs are Installed

The key features of the **Components tab** are summarized below:

- **Module Inclusion/Exclusion:** The Components tab allows you to include or exclude individual modules required by the application. Modules that are common to all RA MCU projects are preselected by default.
- **Module Versioning:** The Components tab displays the available module versions, so switching versions only requires a simple selection of the desired version and deselection of the automatic version.
- **Automatic Module Selection:** All modules that are necessary to support the drivers of the modules selected in the Stacks tab are included automatically. This feature ensures that all required dependencies for the selected drivers are met.
- **Optional Module Selection:** You can include or exclude additional modules by ticking or un-ticking the box next to the required component. This allows for customization based on specific application needs.

- **Dynamic Driver Management:** If a box next to a module is checked and the Generate Project Content button is clicked, all necessary files for that module are copied or created. If that same box is then unchecked, those files will be deleted.

2.2.3 Configure Modules Using the Stacks Tab

The **Properties view** window connected to the **Stacks tab** is primarily used to configure the settings and parameters of individual modules in a project.

When you select a specific module in the Stacks pane, the Properties view dynamically displays all the configuration options available for that module, providing access to adjust the module's specific settings.

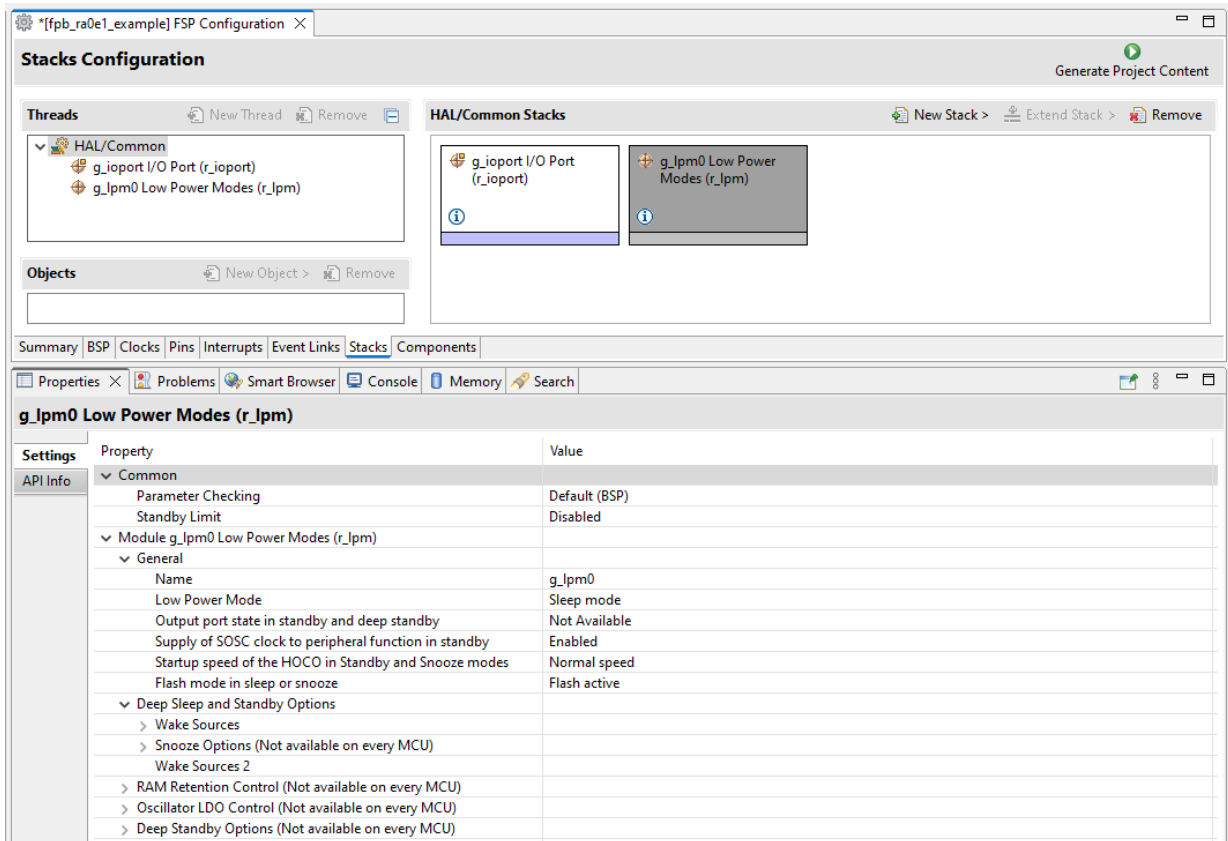


Figure 8. The Properties View of the r_lpm Stack on the FPB-RA0E1 Using FSP v5.7.0

This view allows you to fine-tune the behavior of each module by adjusting its specific configuration parameters, assigning callback functions, configuring interrupts, and more. This configuration step is crucial to set up the functionality of the modules within an application.

After configuring each module's settings, the **Generate Project Content** button is used to create and add the .c and .h files for each module in the Stacks tab into the appropriate folders:

- *ra/fsp/inc/api*
- *ra/fsp/inc/instances*
- *ra/fsp/src/bsp*
- *ra/fsp/src/<Driver_Name>*
- *ra_cfg/bsp_cfg*
- *ra_gen*

The .c and .h files generated comprise the complete FSP driver code for each module. The driver code is automatically generated using the same FSP version as the project. The supporting driver file(s) contents, such as "hal_data.c" and "hal_data.h", depend on the module-specific settings set in **Properties view**.

2.3 Clocks

Users must set up the MCU system clocks properly for their application according to the specifications of the MCU Group's Hardware User's Manual.

This section explains two methods of configuring the clocks: automatically by using the FSP Configuration editor, and manually by using provided BSP APIs.

2.3.1 Configure Clocks Using the Clocks Tab

The **Clocks tab** presents a graphical view of the MCU's clock tree, allowing the various clock dividers and sources to be modified. If a clock setting is invalid, the offending clock value is highlighted in red. It is still possible to generate code with this setting, but correct operation cannot be guaranteed.

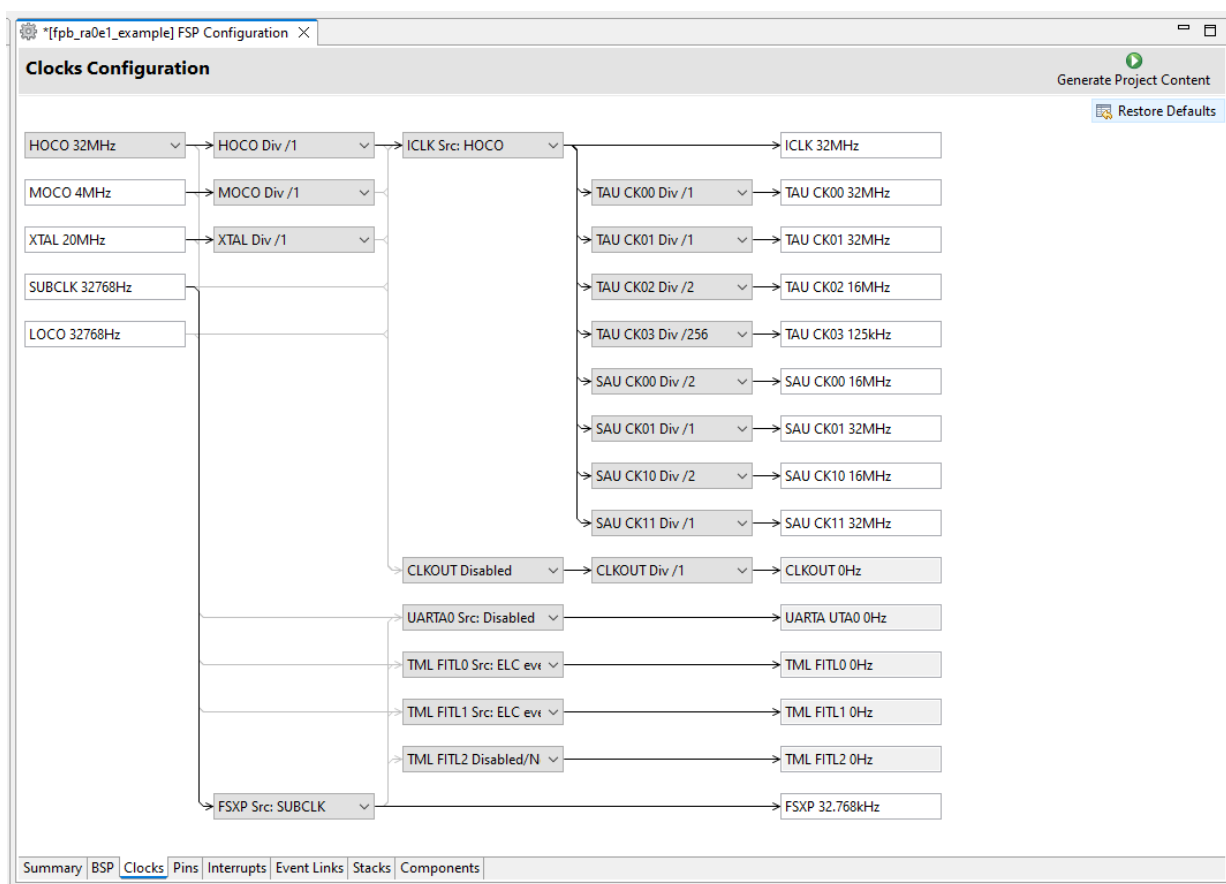


Figure 9. Default Clock Settings for the FPB-RA0E1 as Viewed from the Clocks Tab

After editing the Clocks tab in the FSP Configuration view, the file “**bsp_clock_cfg.h**” is updated with the selected clock configurations. This file is in the **ra_gen** folder as shown in [Figure 2. High-level Workspace Folder Organization of a Newly Created RA C/C++ Project](#). It is important to not manually edit this or any generated configuration file, because they are overwritten every time the “Generate Project Content” or “Build Project” is clicked.

2.3.2 Configure Clocks Using BSP API

While the Clocks tab in the RA Configuration editor provides a GUI for configuring clocks, the BSP provides API functions for programmatic control of clock settings. These functions can be used to retrieve or set clock frequencies and perform other clock-related operations. They are available as a subset of the FSP APIs and are automatically included in a new C/C++ RA Project as described in 2.1.1 Initial Project Organization.

The BSP APIs for clocks are defined in the file “**bsp_clocks.c**”, and some static inline functions for clocks are also defined in the file “**bsp_common.h**”. These files are both deep under the **ra** folder, found in the subfolder **ra/fsp/src/bsp/mcu/all**. The implementation may differ between MCUs so it's important to review the BSP clock API definitions carefully before invoking the functions.

It is possible to create a custom routine to configure clocks by calling the BSP API clock functions. A custom routine will only be needed when not using the FSP Configuration's Clocks tab to edit the system clock frequencies and division ratios. Be sure to call the custom clock setup routine before the program counter

reaches the main application, but after the BSP has configured the system clocks initially, during the Warm Start Post Reset event. The Warm Start function is explained in detail in 4.1.1 BSP Warm Start.

2.4 Configure Peripheral Interrupts

The Arm system interrupts are set up by the BSP, but users must set up the application’s interrupts properly for each peripheral.

Since RA MCUs are based on Arm Cortex-M architecture, the Nested Vectored Interrupt Controller (NVIC) handles the exceptions and interrupt configuration, prioritization, and interrupt masking. Refer to the Arm Cortex-M documentation corresponding to the RA device being used for details on the NVIC and its registers. For example, the RA0E1 MCU uses the Cortex-M23 architecture, so the appropriate reference is the Arm Cortex-M23 Processor Technical Reference Manual.

This section explains two methods of configuring interrupt priorities, configuring interrupt service routines (ISRs), and masking interrupts at runtime: automatically using the FSP, and manually by using the BSP.

2.4.1 Interrupt Setup and Control Using the FSP

From the **Stacks** tab, select a driver to view and edit its properties, which will appear by default below in the **Properties View**. If the peripheral has one or more interrupts, they can be enabled by setting the desired interrupt priority. Along with the priority level, the name of the user-defined ISR callback routine can be configured.

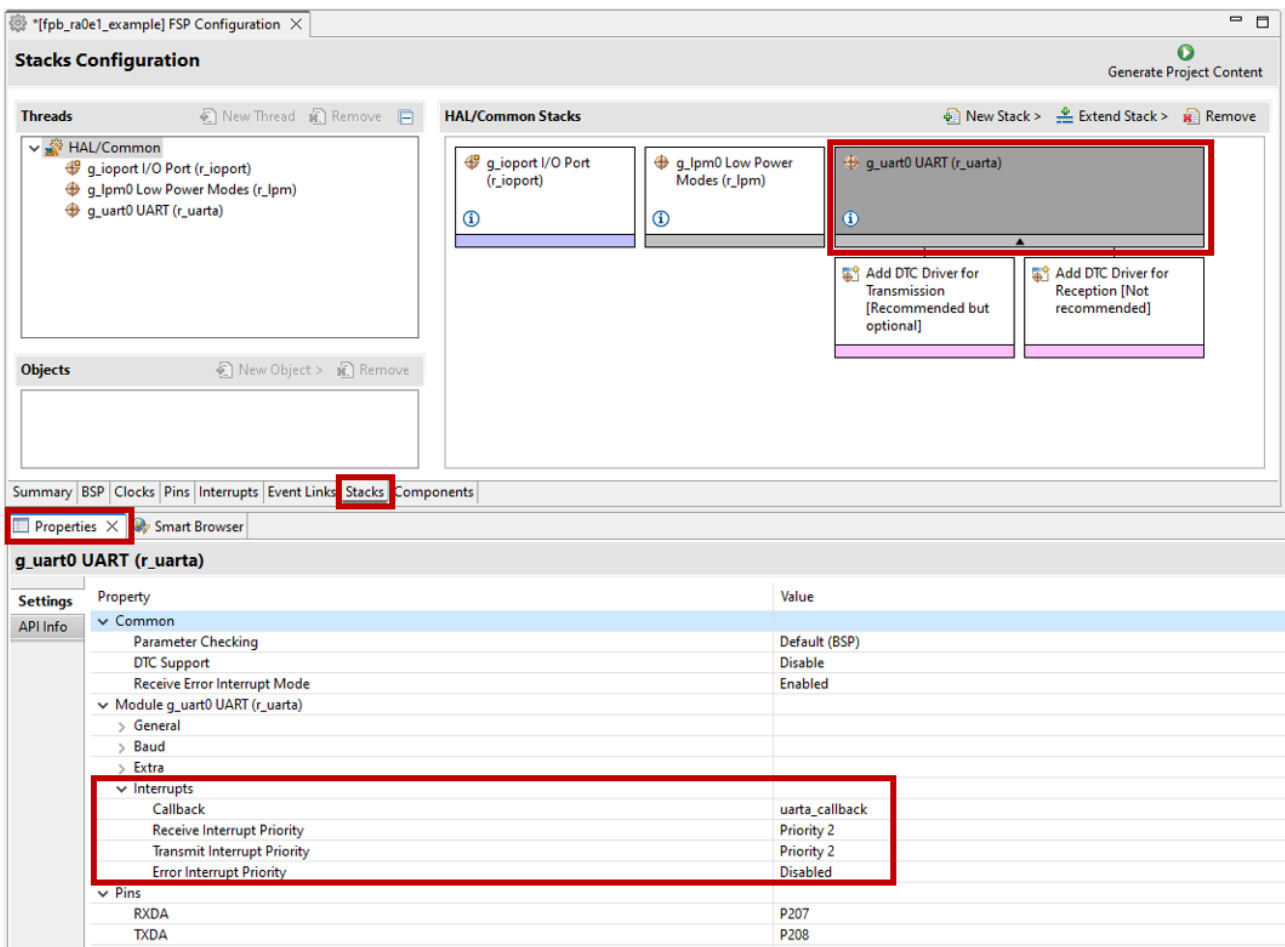


Figure 10. Example Configuring the UART Module's Interrupt Priorities and User-Defined Callback

When using the FSP driver code to implement an application, the FSP module drivers will automatically define and handle the interrupt service routines for RA MCU peripherals to ensure the required hardware procedures are implemented.

As the FSP drivers’ ISRs complete they will call the user-defined callback specified in **Stacks** tab to allow the application to respond. When calling a user-define callback, the FSP driver callbacks will pass relevant information to the user-defined callback through the `p_args` variable, which has varying fields depending on

the peripheral. The **Interrupts tab** shows a summary of the enabled FSP driver peripheral IRQs and shows the name of the ISRs defined in the FSP drivers.

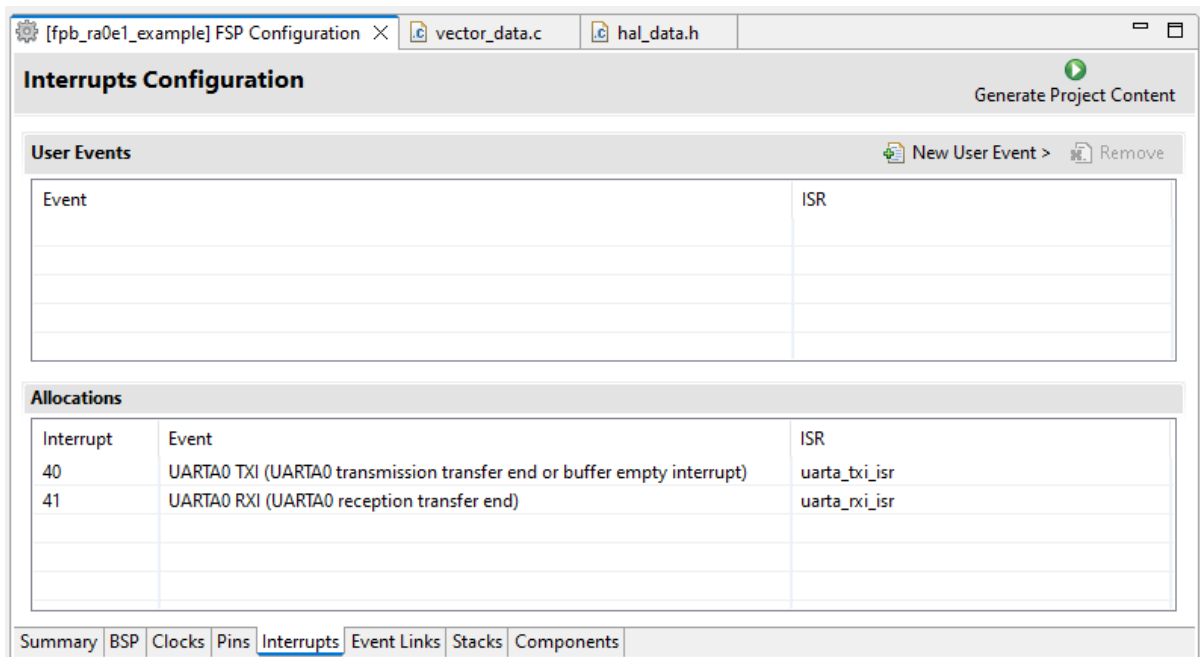


Figure 11. The Interrupts Tab Shows a Summary of Application Peripheral Interrupts

To understand this better with an example, review Figure 10. Example Configuring the UART Module's Interrupt Priorities and User-Defined Callback and Figure 11. The Interrupts Tab Shows a Summary of Application Peripheral Interrupts above. In the **Interrupts** tab the `uarta_txi_isr()` and `uarta_rxi_isr()` comprise the FSP driver peripheral callbacks, which were enabled by setting each IRQ's priority from the module properties in the **Stacks** tab. When either IRQ event is triggered during runtime, the respective `uarta_txi_isr()` or `uarta_rxi_isr()` will be called first and will handle reading from and writing to the UARTA module's hardware registers appropriately.

The final call of each peripheral ISR calls the user-defined callback setup in the **Stacks** tab as shown in Figure 10. Example Configuring the UART Module's Interrupt Priorities and User-Defined Callback, the `uarta_callback()` routine in this example. The peripheral ISRs pass relevant information to the `uarta_callback()` using the `p_args` variable which is defined separately for each module and detailed in the FSP User's Manual. For the UARTA, `p_args` includes information regarding the channel, event, and data.

After saving the configuration.xml or clicking Generating Project Content, the configuration information regarding any application peripheral interrupts that have been set using the **Stacks** and **Interrupt** tabs are saved in the `ra_gen` folder in the "`vector_data.c`" and "`vector_data.h`" files. You can see in the image below how the application peripheral vector table now includes the UARTA TXI and RXI ISRs.

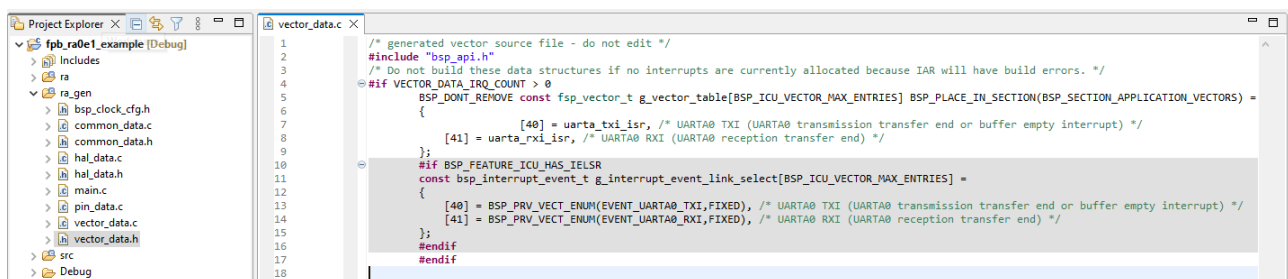


Figure 12. The File vector_data.c Contains the Application Peripheral Vector Table

The BSP initialization routine will map the user-enabled peripheral events from "`vector_data.c`" and "`vector_data.h`" to the appropriate NVIC interrupt number. The BSP initialization routine will set up both the Arm system interrupts and the application peripheral interrupts by pointing the NVIC VTOR register to the vector table.

For runtime control, the FSP driver routines automatically handle setting the specified priority level and masking/unmasking the IRQ as needed. The FSP drivers do this by calling the appropriate BSP APIs for IRQ handling. Refer to each module’s FSP driver code and the FSP User’s Manual to understand the implementation.

2.4.2 Interrupt Setup and Control Using the BSP

An alternative to configuring interrupts with the Stacks tab and using the FSP drivers to set priorities and control the masking/unmasking of the IRQs, you can manually setup peripheral interrupts with the **Interrupts tab** and use the BSP APIs to set priorities and control masking/unmasking of the IRQs.

The **Interrupts tab** can be used to bypass any FSP peripheral ISRs that have been enabled through the Stacks tab, which happens when assigning a stack module’s IRQ(s) a priority level. This allows users to define a custom ISR to be used in place of the FSP driver’s defined peripheral ISR, for any peripheral IRQ that is required in the application.

Or in cases where the **Stacks tab** is not used to configure modules or enable module IRQs, the Interrupts tab allows you to enable peripheral IRQs and create custom ISRs for each application event required.

To add a new event in the **Interrupts tab**: use the **New User Event** button, navigate to the peripheral of interest, add each desired IRQ, and provide the ISR an appropriate callback name.

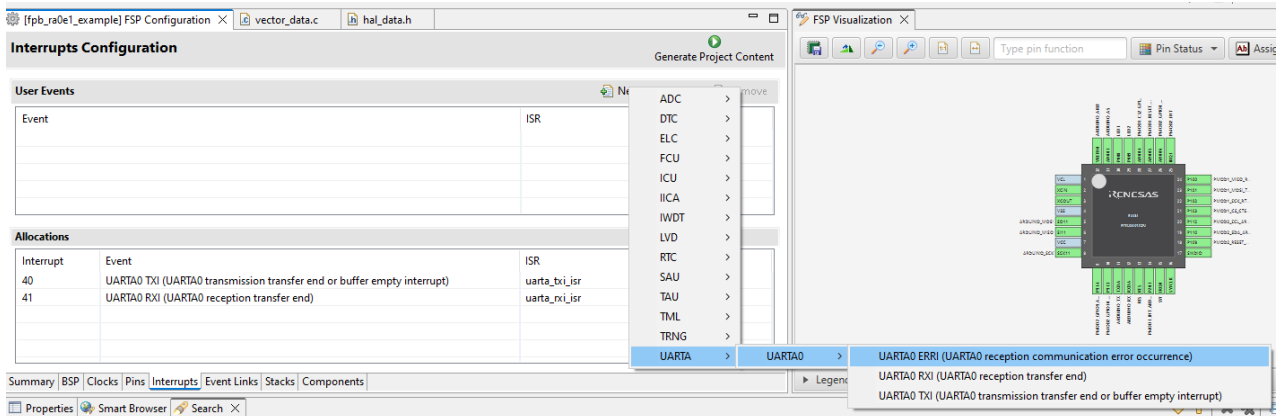


Figure 13. Add a User-Defined ISR Callback for the UARTA ERRI IRQ in the Interrupts Tab

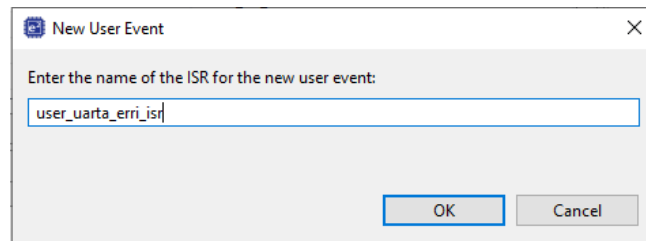


Figure 14. Create a Descriptive Name for the ISR

Again, after saving the configuration.xml or clicking Generating Project Content, the configuration information regarding any the application peripheral interrupts that have been set using the **Interrupt tab** is saved in the **ra_gen** folder in the “**vector_data.c**” and “**vector_data.h**” files. In the image below, you can see that the ERRI IRQ has been added to the vector table with the TXI and RXI IRQs. After a reset, the BSP initialization routine will automatically handle setting up the NVIC registers to point to the vector table.

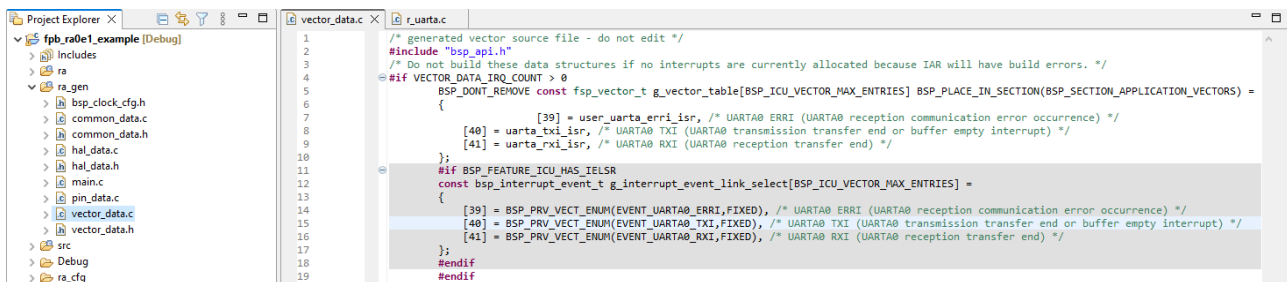


Figure 15. The FSP Peripheral ISRs and the User-Defined ISR are Combined in vector_data.c

For runtime control, the BSP provides a series of IRQ-related APIs can be used in the application code to set priorities, clear pending interrupts, and to mask and unmask the IRQs as needed. These BSP APIs for interrupts are defined in the file “**bsp_irq.c**” which is in the **ra** folder under **ra/fsp/src/bsp/mcu/all**. The following is a summary of the functions on the RA0E1, but refer to the BSP documentation for each MCU, as they may differ between architectures:

- **R_BSP_IrqCfg()**: Set the interrupt priority level.
- **R_BSP_IrqCfgEnable()**: Set the interrupt priority level, clear pending interrupts, and enable the interrupt.
- **R_BSP_IrqEnable()**: Clear pending interrupt and enable the interrupt.
- **R_BSP_IrqDisable()**: Disable the interrupt.
- **R_BSP_IrqClearPending()**: Clear pending interrupt.
- **R_BSP_IrqEnableNoClear()**: Enable the interrupt.

When an IRQ is triggered at runtime, any user-defined peripheral ISRs should first read from and write to the peripheral registers to control the module hardware correctly before responding at the application level.

2.5 The Project's Linker Script

Developers who want to modify the linker script for an RA project should have a strong understanding of the address space of the device and how their end application uses memory. The RA MCU Group's Hardware User's Manual Chapter 4 is dedicated to detailing the Address Space of the MCU, providing the address ranges of the different memory regions.

The linker script for an RA e² studio project is found in the **script** project folder and the file has a **.ld** extension for GNU and **.lld** for LLVM. The FSP has changed the linker script name over the years. With the FSP version 5.7.0, corresponding to the version with which the accompanying application projects run, have a linker script titled “**fsp.ld**”.

Some macros for memory regions that the linker script uses are automatically generated and defined in the file “**memory_regions.ld**” which is in the **Debug** project folder. The **.map** file is generated upon compiling the project based on the contents of “**fsp.ld**” and can be found in the **Debug** folder as “**<project_name>.map**”.

As an example, the image below shows these three files for one project. The boxes highlight the linker script file (fsp.ld), the memory region macro file (memory_regions.ld), and the resultant project memory layout .map file (LLRP_fpb_ra0e1_custom.map).

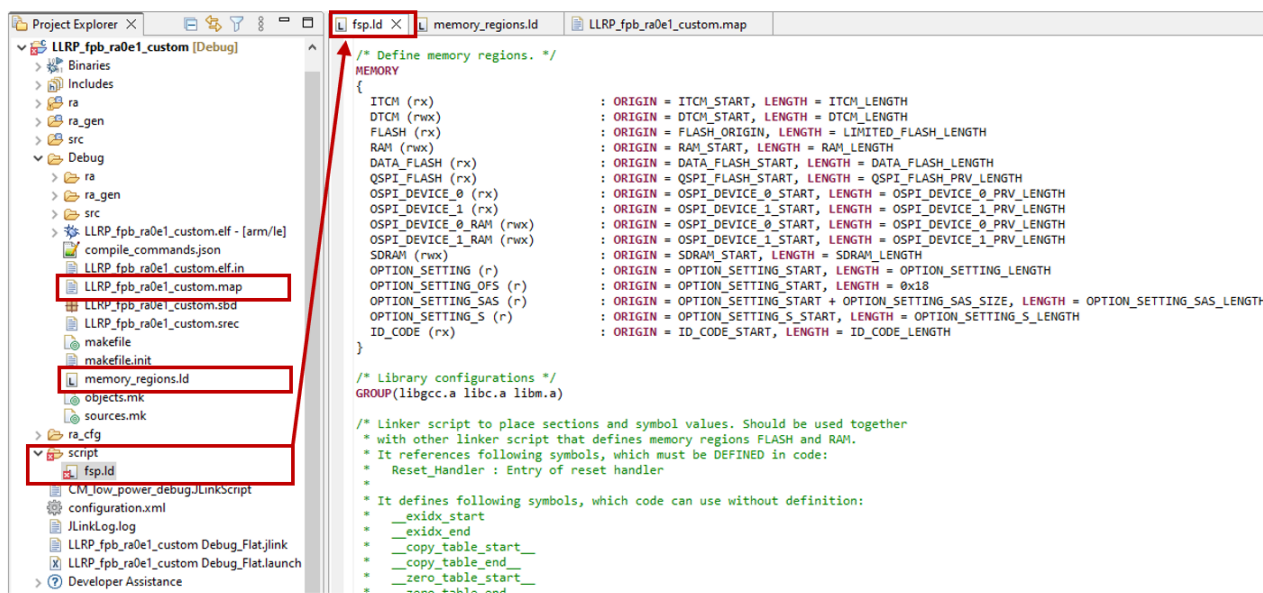


Figure 16. An e² studio Project's Linker Script and Related Files

2.6 Tool Highlight: Developer Assistance Examples

The **Developer Assistance** is an FSP coding tool designed to simplify and accelerate the development process by providing a drag-and-drop interface for API functions and other code elements. It helps developers quickly insert pre-configured code snippets into their projects, reducing the likelihood of errors and speeding up the coding process.

The Developer Assistance is found in the Project Explorer window at the bottom of each project's top level folder. The contents under the Developer Assistance are based on the modules that have been added to the project in the configuration.xml Stacks or Components.

For example, in Figure 17. Developer Assistance Options Reflect the Configured Modules of a Project the r_ioport and the r_adc_d stacks have saved configurations in the Stacks on the right, so they are the only member options available under the Developer Assistance in the Project Explorer on the left:

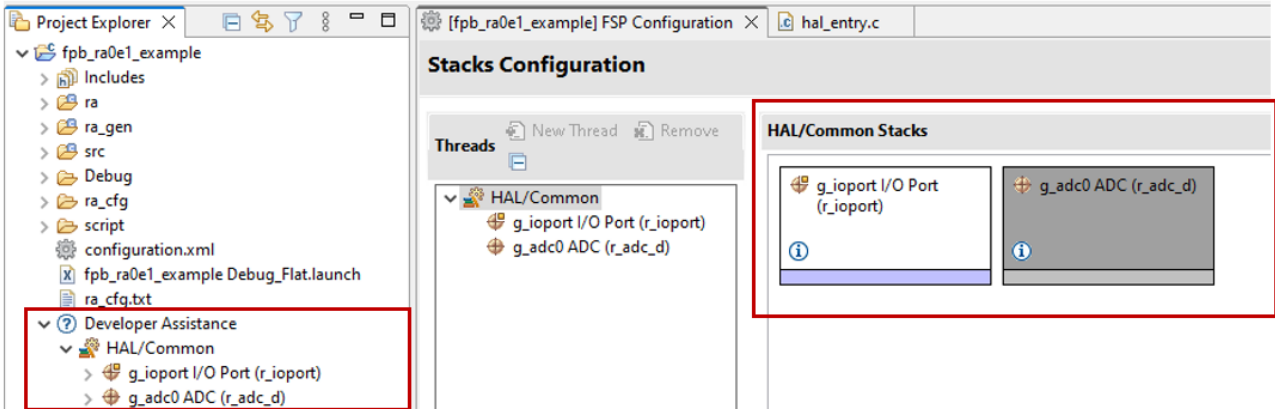


Figure 17. Developer Assistance Options Reflect the Configured Modules of a Project

Developer Assistance will list all FSP APIs for each module, and when applicable it also includes callback templates, quick setup functions for middleware, and the autocomplete functionality.

Some examples of using Developer Assistance to assist in development include:

(1) Adding API Calls:

Developers can drag and drop API function calls directly into their source code. For instance, to initialize an ADC driver, a developer can drag the R_ADC_D_Open() function from the Developer Assistance tool into their code, which automatically inserts the function call with the appropriate parameters.

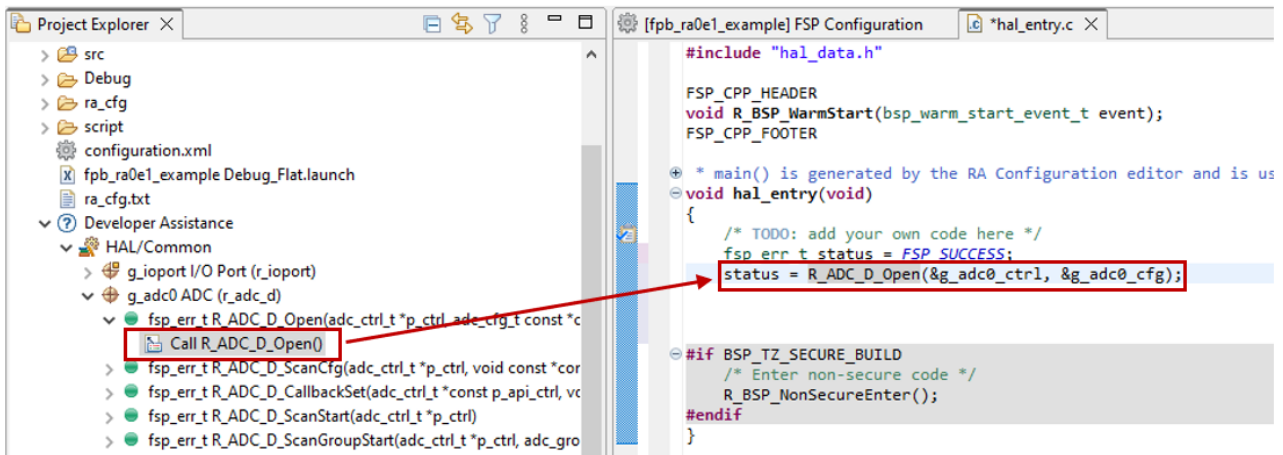


Figure 18. Example of API Drag-and-Drop with Developer Assistance

(2) Configuring Callbacks:

When a module requires a callback function to handle interrupts, Developer Assistance can be used to generate the callback template. For example, for the ADC module, the callback can be selected from the

Developer Assistance list and dragged into the project code, ensuring that the correct function prototype and initial code are included.

The callback name in the function prototype from Developer Assistance will have the same name as specified in the Stacks Properties, as shown below:

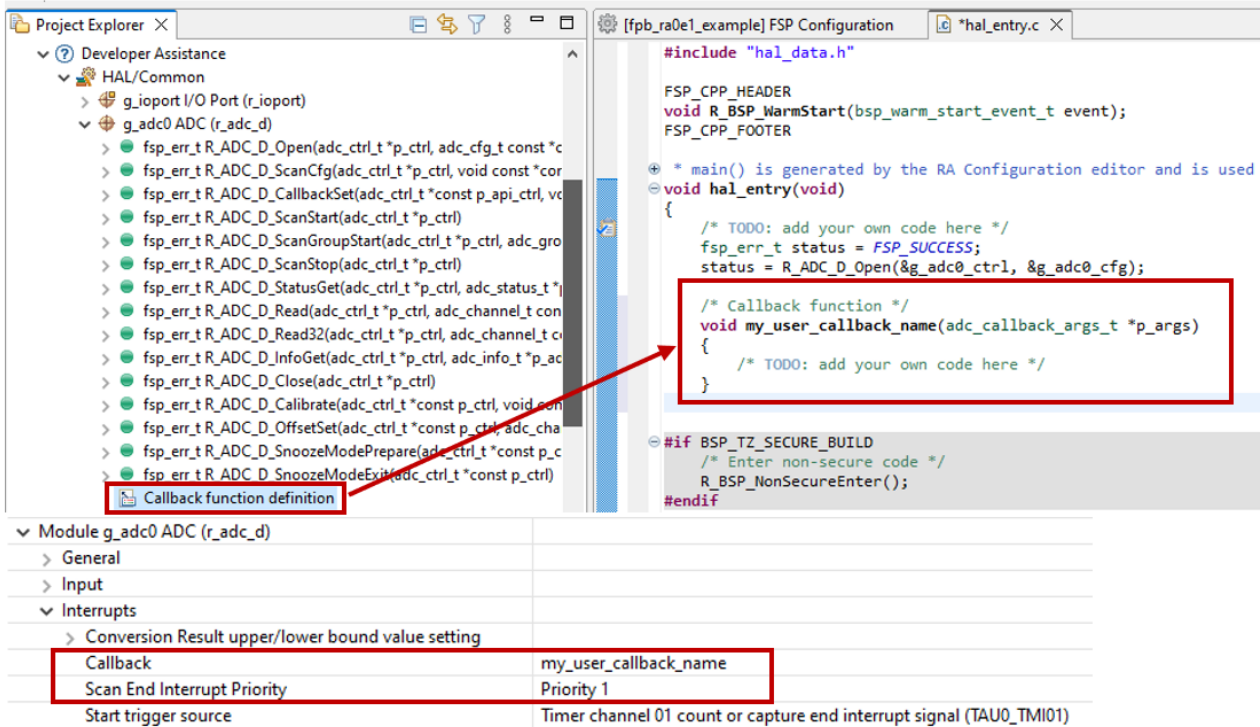


Figure 19. Example of Module Callback Prototype with Developer Assistance

(3) Quick Setup Functions:

Developer Assistance provides quick setup functions for various modules, typically for middleware stacks. For instance, in a project involving I2C communication with sensors, a developer can use Developer Assistance to drag and drop the g_comms_i2c_bus0_quick_setup and g_hs300x_sensor0_quick_setup functions into their code, streamlining the initialization process.

(4) Autocomplete Feature:

Developer Assistance combines with the Autocomplete feature in e2 studio, suggesting possible completions for partially typed code elements. For example, typing BSP_IO_ and pressing Ctrl + Space will bring up a list of context-aware options like BSP_IO_LEVEL_HIGH, which can be selected to complete the code.

Visit 3.2.3 Tool Highlight: e2 studio Autocomplete for Register Access for more details regarding utilizing autocomplete for low-level coding assistance.

3. Step 2: Create Custom HAL Drivers

To avoid using the FSP HAL drivers, you will need to design and create a set of custom hardware drivers for each of the peripherals you need in the end application. The custom drivers will write to and read from the peripheral's special function registers (SFRs) directly to operate the peripheral in the right manner for the application. The drivers must carefully follow the register specifications that are detailed in the respective peripheral chapter of the RA MCU's Hardware User's Manual.

This section provides guidelines and strategies for designing and testing custom HAL drivers for any given module on an RA microcontroller. It explains where to find the SFR definitions in the BSP code, how to write a custom HAL driver by manipulating the SFRs, and what built-in e2 studio tools aid in this procedure.

The information applies to all RA microcontroller peripherals but note that this section illustrates the topics using the FPB-RA0E1 and its hardware, so some implementation details may differ across devices.

3.1 Accessing a Module's Special Function Registers (SFRs)

This section explains how to access the SFRs for any RA microcontroller peripheral. To add custom driver functionality, it is recommended to use the special function register definition files. Official ARM CMSIS documentation refers to this format of related register definition files as “device header files”. Within Renesas, these files are often referred to as an “IO define” or more commonly “iodefine”.

Iodefine files contain definitions for all the peripheral registers on a device. In the RA BSP, one iodefine header is provided per device group (RA0E1, RA2A1 etc.) that contains all the register definitions provided in the hardware manual for that group. These headers are accessed through the BSP file or “**bsp_api.h**”, which includes the appropriate iodefine file based on the MCU configured in the initial project settings.

The BSP iodefine file is titled “<MCU_part#>.h” and it contains the structure definitions for all SFRs present on the MCU part # in question. The file also defines other helpful macros for each SFR, including peripheral base addresses, bitfield positions and bitfield masks.

For the FPB-RA0E1 the iodefine file “**R7FA0E107.h**” contains the SFR structure definitions for the MCU. The iodefine file is located in the folder **ra/fsp/src/bsp/cmsis/Device/Renesas/Include** folder. The image below shows the FPB-RA0E1's iodefine file's definition of the ADC special function registers ADM0 and ADS and their available bitfields.

```

/* ===== R_ADC_D =====
**
** @brief A/D Converter (R_ADC_D)
**
typedef struct                /*!< (@ 0x400A1800) R_ADC_D Structure
{
    union
    {
        __IOM uint8_t ADM0;    /*!< (@ 0x00000000) A/D Converter Mode Register 0

        struct
        {
            __IOM uint8_t ADCE : 1; /*!< [0..0] A/D Voltage Comparator Operation Control
            __IOM uint8_t LV : 2;   /*!< [2..1] Select Operation Voltage Mode
            __IOM uint8_t FR : 3;   /*!< [5..3] Select Conversion Clock (fAD)
            __IOM uint8_t ADMD : 1; /*!< [6..6] Specification of the A/D Conversion Channel Selection
                                   * Mode
            __IOM uint8_t ADCS : 1; /*!< [7..7] A/D Conversion Operation Control
        } ADM0_b;
    };

    union
    {
        __IOM uint8_t ADS;        /*!< (@ 0x00000001) Analog Input Channel Specification Register

        struct
        {
            __IOM uint8_t ADS : 5; /*!< [4..0] Selection of the Analog Input Channel (See to )
            uint8_t : 2;
            __IOM uint8_t ADISS : 1; /*!< [7..7] Select Internal or External of Analog Input (See to )
        } ADS_b;
    };
};

```

Figure 20. The R7FA0E107.h Iodefine File Showing the ADC Registers ADM0 and ADS

The FSP drivers are implemented using the SFR structure definitions from the iodefine file. As an example, look at Figure 20. The R7FA0E107.h Iodefine File Showing the ADC Registers ADM0 and ADS. It depicts the **R_ADC_D_ScanCfg()** API function which writes the configuration values set in the configuration.xml file to R_ADC_D registers ADS and ADTES.

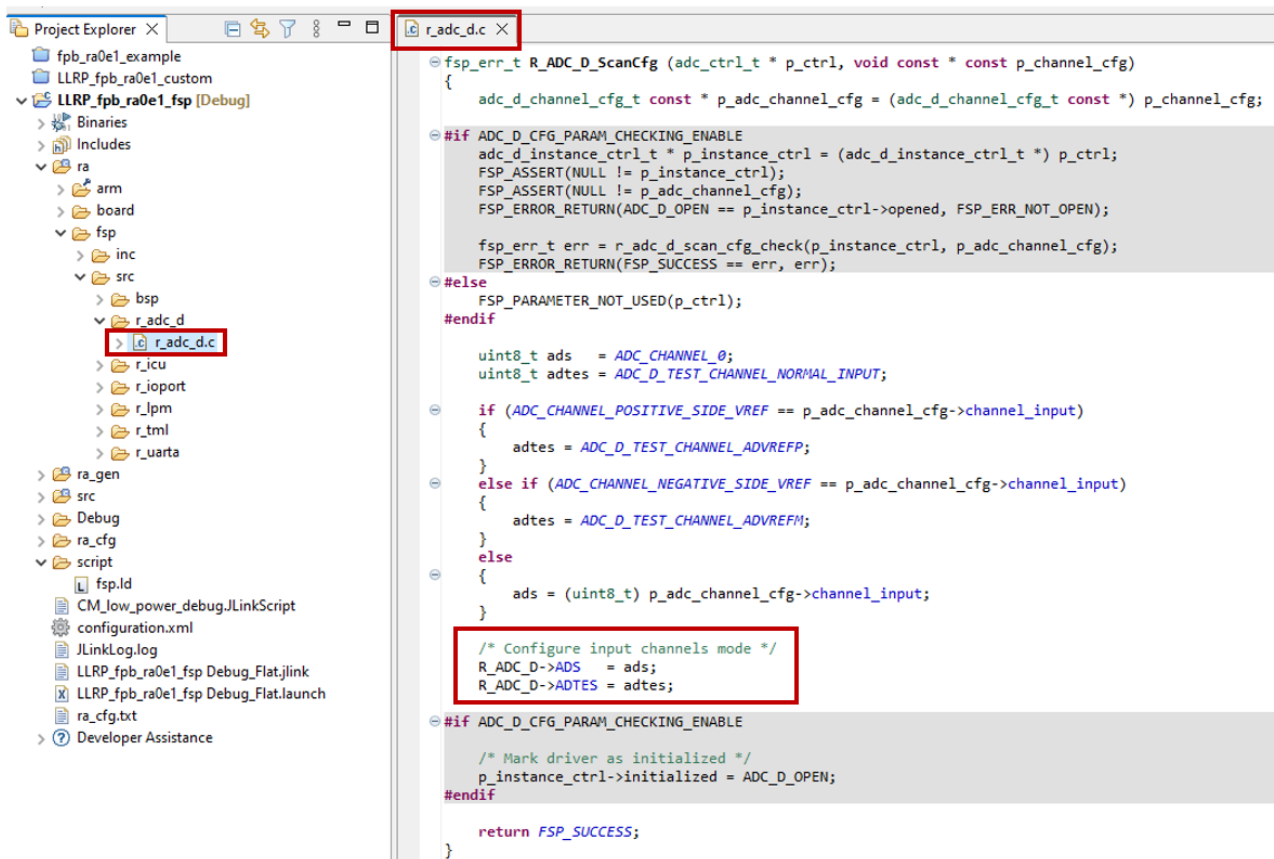


Figure 21. The FSP Drivers are Implemented with the SFRs Defined in the Device’s Iodfide File

3.1.1 Key Components in I/O Define Header File

The iodfide file titled “<MCU_part#>.h” provides access to peripheral register sets through a series of defined structs. It is a quite large file because it contains a definition for every register and its bitfields as defined in the Hardware User’s Manual, among other macro definitions to support accessing the structs and bitfields.

The iodfide file can be broken down into three main sections:

(1) Device-Specific Cluster Section:

The first main register definition section of the iodfide file is the **Device-Specific Cluster Section**.

Clusters describe groups of registers that are repeated multiple times. Clusters must be accessed similarly to arrays, where the registers are elements of the array values.

(2) Device-Specific Peripheral Section

The next section is the **Device-Specific Peripheral Section**, which contains the majority of the MCU’s SFR definitions. The section can be broken down further into three subsections:

First is the **Device-Specific Peripheral Section** which contains most of the register structs, excluding the already defined clusters. There is a struct definition for every peripheral on the MCU, whose members include every register for that peripheral. There is additional consideration to access to the register’s bitfields using unions.

Figure 22. The ADM0 Register is Defined as a Member of the R_ADC_D_Type Struct shows the beginning of the R_ADC_D’s Device-Specific Peripheral Section which begins by defining the ADM0 register and its bitfields.

```

/* ===== R_ADC_D ===== */
/* ===== R_ADC_D ===== */
/* ===== R_ADC_D ===== */

/**
 * @brief A/D Converter (R_ADC_D)
 */

typedef struct /*!< (@ 0x400A1800) R_ADC_D Structure */
{
    union
    {
        __IOM uint8_t ADM0; /*!< (@ 0x00000000) A/D Converter Mode Register 0 */

        struct
        {
            __IOM uint8_t ADCE : 1; /*!< [0..0] A/D Voltage Comparator Operation Control */
            __IOM uint8_t LV : 2; /*!< [2..1] Select Operation Voltage Mode */
            __IOM uint8_t FR : 3; /*!< [5..3] Select Conversion Clock (fAD) */
            __IOM uint8_t ADM0 : 1; /*!< [6..6] Specification of the A/D Conversion Channel Selection
            * Mode */
            __IOM uint8_t ADCS : 1; /*!< [7..7] A/D Conversion Operation Control */
        } ADM0_b;
    };
};

```

Figure 22. The ADM0 Register is Defined as a Member of the R_ADC_D_Type Struct

The second subsection is the **Device-Specific Peripheral Address Map**. This section defines the base addresses for every peripheral register set.

Figure 23. The Base Addresses are Defined for Each Peripheral Register Set on the RA0E1 shows the beginning of the Device-Specific Peripheral Address Map on the RA0E1, which organizes the peripheral address bases in alphabetical order.

```

/* ===== Device Specific Peripheral Address Map ===== */
/* ===== Device Specific Peripheral Address Map ===== */
/* ===== Device Specific Peripheral Address Map ===== */

/** @addtogroup Device_Peripheral_peripheralAddr
 * @{
 */

#define R_ADC_D_BASE      0x400A1800UL
#define R_BUS_BASE       0x40003000UL
#define R_CRC_BASE       0x40074000UL
#define R_DEBUG_BASE     0x4001B000UL
#define R_DTC_BASE       0x40005400UL
#define R_ELC_BASE       0x40041000UL
#define R_FACI_LP_BASE   0x407EC000UL
#define R_ICU_BASE       0x40006000UL
#define R_IICA0_BASE     0x400A3000UL
#define R_IICA1_BASE     0x400A3008UL
#define R_IWDT_BASE      0x40044400UL

```

Figure 23. The Base Addresses are Defined for Each Peripheral Register Set on the RA0E1

The third subsection is the **Peripheral Declaration**. This section defines “peripheral abbreviations” for each peripheral register set and it ensures that the register structs align in memory with the base addresses of the register set.

You can see how the peripheral declarations tie together the base addresses from Figure 24. The Peripheral Declarations for the RA0E1 to the struct definition in Figure 22. The ADM0 Register is Defined

as a Member of the R_ADC_D_Type Struct.

```

/* ===== */
/* ===== Peripheral declaration ===== */
/* ===== */

/** @addtogroup Device_Peripheral_declaration
 * @{
 */

#define R_ADC_D      ((R_ADC_D_Type *) R_ADC_D_BASE)
#define R_BUS        ((R_BUS_Type *) R_BUS_BASE)
#define R_CRC        ((R_CRC_Type *) R_CRC_BASE)
#define R_DEBUG      ((R_DEBUG_Type *) R_DEBUG_BASE)
#define R_DTC        ((R_DTC_Type *) R_DTC_BASE)
#define R_ELC        ((R_ELC_Type *) R_ELC_BASE)
#define R_FACI_LP    ((R_FACI_LP_Type *) R_FACI_LP_BASE)
#define R_ICU        ((R_ICU_Type *) R_ICU_BASE)
#define R_IICA0      ((R_IICA0_Type *) R_IICA0_BASE)
#define R_IICA1      ((R_IICA0_Type *) R_IICA1_BASE)
#define R_IWDT       ((R_IWDT_Type *) R_IWDT_BASE)

```

Figure 24. The Peripheral Declarations for the RA0E1

(3) Position and Mask Section:

The final main section is the **Position and Mask Section** for both Clusters and Peripherals. This section includes macro definitions for bitfield masks and bitfield positions. It is helpful to use the mask and position macros when setting specific bitfields in each peripheral SFR.

The image below shows the ADM0 register's macros for each bitfield in the SFR.

```

/* ===== */
/* ===== Pos/Mask Peripheral Section ===== */
/* ===== */

/** @addtogroup PosMask_peripherals
 * @{
 */

/* ===== R_ADC_D ===== */
/* ===== */

/* ===== ADM0 ===== */
/* ===== */
#define R_ADC_D_ADM0_ADCE_Pos    (0UL)    /*!< ADCE (Bit 0) */
#define R_ADC_D_ADM0_ADCE_Msk    (0x1UL)  /*!< ADCE (Bitfield-Mask: 0x01) */
#define R_ADC_D_ADM0_LV_Pos     (1UL)    /*!< LV (Bit 1) */
#define R_ADC_D_ADM0_LV_Msk     (0x6UL)  /*!< LV (Bitfield-Mask: 0x03) */
#define R_ADC_D_ADM0_FR_Pos     (3UL)    /*!< FR (Bit 3) */
#define R_ADC_D_ADM0_FR_Msk     (0x38UL) /*!< FR (Bitfield-Mask: 0x07) */
#define R_ADC_D_ADM0_ADM0_Pos   (6UL)    /*!< ADM0 (Bit 6) */
#define R_ADC_D_ADM0_ADM0_Msk   (0x40UL) /*!< ADM0 (Bitfield-Mask: 0x01) */
#define R_ADC_D_ADM0_ADCS_Pos   (7UL)    /*!< ADCS (Bit 7) */
#define R_ADC_D_ADM0_ADCS_Msk   (0x80UL) /*!< ADCS (Bitfield-Mask: 0x01) */

```

Figure 25. The Macro Definitions for the ADM0 Register of the ADC12 Peripheral on the RA0E1

3.1.2 Peripheral Register Template from the I/O Define File

Since there are many lines in the iodefinition file, it helps to use the register naming convention to navigate the document.

The general template to follow when accessing a register struct is:

R_ + peripheral abbreviation + channel number + -> + register name [+ _b. + bitfield]

The square brackets are an optional suffix to be used only when accessing a specific bitfield of the register, rather than accessing the register.

Using the FPB-RA0E1 and Figure 21. The FSP Drivers are Implemented with the SFRs Defined in the Device's Iodefinition File and Figure 23. The Base Addresses are Defined for Each Peripheral Register Set on the RA0E1 to illustrate an example, to access the ADC12 module's register ADM0 "A/D Converter Mode Register 0", the naming would be:

R_ADC_D->ADM0

The **R_** is common to all peripheral structs, the **ADC_D** is the peripheral abbreviation for the ADC12 module on the RA0E1, the **->** operation dereferences the pointer to the base address and accesses the struct members, and **ADM0** is the name of the register in question.

For example, to access the bitfield ADCE of the ADM0 register, the bitfield extension suffix in the brackets from the above template can be added:

R_ADC_D->ADM0_b.ADCE

It is worth noting that each bitfield access will cause a full read-modify-write of the register. The commands cannot be combined by the compiler because the register definitions are volatile by necessity. As such, writing to bitfields can be size and speed inefficient, particularly when the peripheral is on a slow clock. It is recommended to write whole registers wherever possible, and to use the bitfield masks and position macro definitions to set each bitfield accordingly when writing to a whole register.

For example, to set multiple bitfields like the AMDM, LV and FR in a single write to the ADM0 register, the respective bitfield masks can be used:

```
R_ADC_D->ADM0 =
    (uint8_t)( (CHANNEL_MODE << R_ADC_D_ADM0_ADM_Pos) |
              (OPERATION_VOLTAGE << R_ADC_D_ADM0_LV_Pos) |
              (CONVERSION_CLK_DIV << R_ADC_D_ADM0_FR_Pos) );
```

In this example, the macros ending in _Pos were defined in the Position and Mask Peripheral section of the iodefinition file shown in Figure 25. The Macro Definitions for the ADM0 Register of the ADC12 Peripheral on the RAOE1.

3.1.3 Tool Highlight: CDT Include Browser

The e² studio IDE comes installed with a software tool: the CDT Include Browser.

This view shows the <include> and #include relationships between source and header files. To use the Include Browser in e²studio, click on **Window > Show View > Include Browser**. Drag and drop any file of interest to review the file dependency tree.

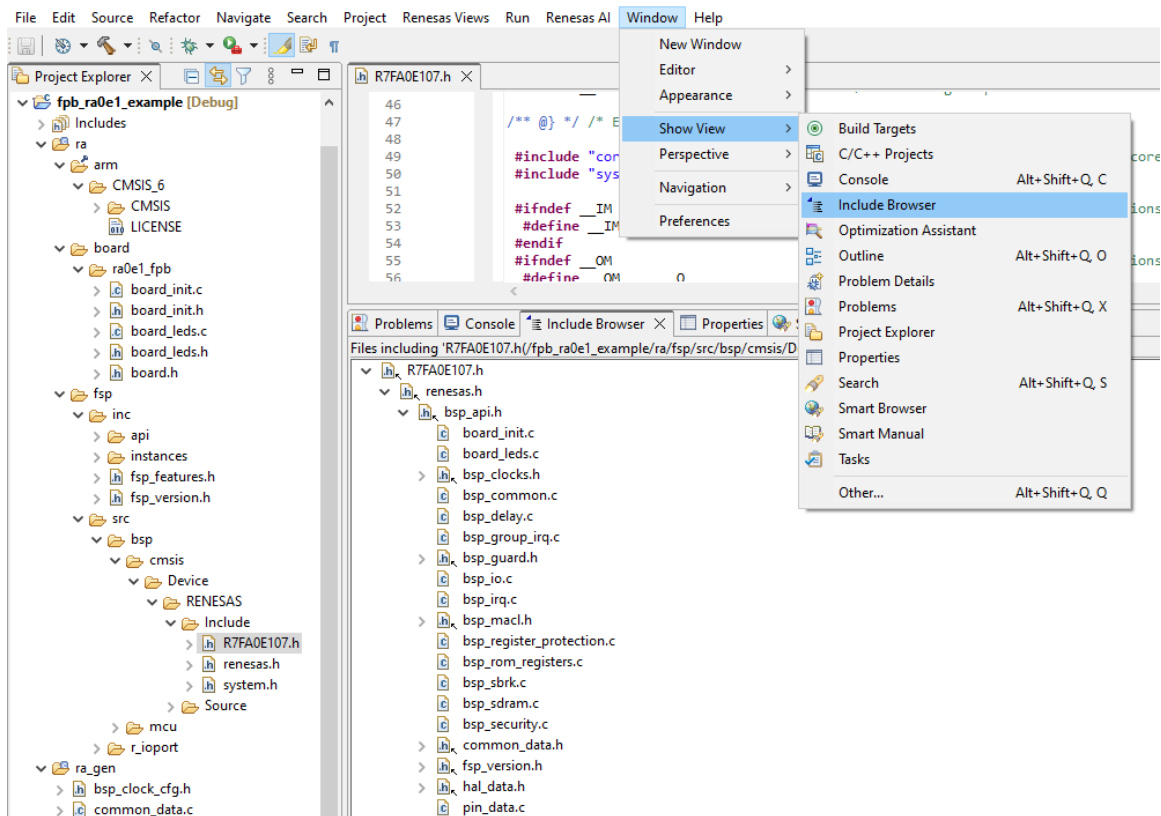


Figure 26. Checking File Dependencies for the R7FA0E107.h File in the Include Browser

This tool is useful for understanding the file dependencies for any application. It's important for developers to confirm that header files of interest are accessible to any C source file that is created or modified. The CDT Include Browser solves this problem efficiently making it a key tool for writing custom drivers.

As seen in Figure 26. Checking File Dependencies for the R7FA0E107.h File in the Include Browser above, the CDT browser results for the iodefinition file show that it is depended on through the BSP header files like “bsp_api.h”.

Any custom HAL driver files should include the BSP header file “bsp_api.h” to access the peripheral register structs and other supporting BSP API functions and macros. Including bsp_api.h rather than the other listed files like renesas.h, ensures all of the necessary files are added with the correct include order. To properly include renesas.h to a file, it is required that bsp_feature.h is included first.

3.2 Writing Custom HAL Drivers

This section provides a general method for writing custom low-level HAL drivers by reading from and writing to the peripheral SFRs and includes best tips for this process. During the process of creating custom low-level drivers, it is essential to strictly follow the specifications of the peripherals from the target MCU Group's Hardware User's Manual.

3.2.1 Tracing the FSP Project's Program Flow

It is a tedious effort to solely rely on the specifications of the Hardware User's Manual to set SFRs in the right order to create a complex end application that uses many peripherals as a system.

Instead, the recommended method is to trace the FSP reference project's flow to write custom HAL drivers for each peripheral. The three main reasons for creating and referencing an FSP version of the desired end application are:

1. to understand the application's flow as a whole,
2. to break down the application's order-of-operations into its constituent parts for designing each peripheral driver routine,
3. to test the custom drivers against the rigorously-validated FSP code.

You should still refer to the specifications of each peripheral's operation in the Hardware User's Manual. The manual contains a complete reference for every module's special function registers on the RA MCU. This reference includes the peripheral register's base address and offset, read/write access, register addresses, default bitfield values after reset, timing information, restrictions, and more.

To trace the FSP reference project's program flow, begin by understanding the application's flow as a whole system. It is helpful to create a flowchart of the order that the FSP APIs are called along with the necessary application subroutines and user callbacks.

Once the order of operations is understood, each peripheral operation/routine can be turned into a separate custom driver. It's recommended to create a separate .c and .h file pair for each peripheral used in the end application. For example, if using the ADC and the TML you should create a .c and .h file pair for the custom ADC drivers and another pair for the custom TML drivers.

The .c file will contain the definitions of the custom peripheral driver routines required in the end application. These should be designed based off of the FSP project's driver code. The .h file will be used to link the iodefinition file for SFR access and to declare the custom peripheral driver routines, making them visible to the application including the .h files and therefore callable. It's also recommended to use the .h file to define macros for the values of each peripheral register. The register values can be easily changed within the .h file without needing to update the .c file custom drivers. Give each configuration macro a meaningful name for code readability.

It's best to write the custom drivers based on the implementation of the FSP drivers' code since they are thoroughly tested. The FSP drivers' .c files are found in the folder *ra/fsp/src/r_<peripheral>* and have the naming convention “r_<peripheral>.c”. You can open the FSP driver file from the project's workspace and use the **C/C++ View's Outline Window** to navigate to the FSP API of interest to see the implementation.

As an example, the image below reflects opening the “r_tml.c” FSP driver file using the **C/C++ Perspective** and selecting the `R_TML_Open()` API in the **Outline Window** to the right. This causes the driver file in the center window to jump to the line where the API definition begins. In place of the FSP API parameters `p_ctrl` and `p_cfg`, you can create macros in your own custom driver .h file to store the appropriate configuration values for the peripherals in your application.

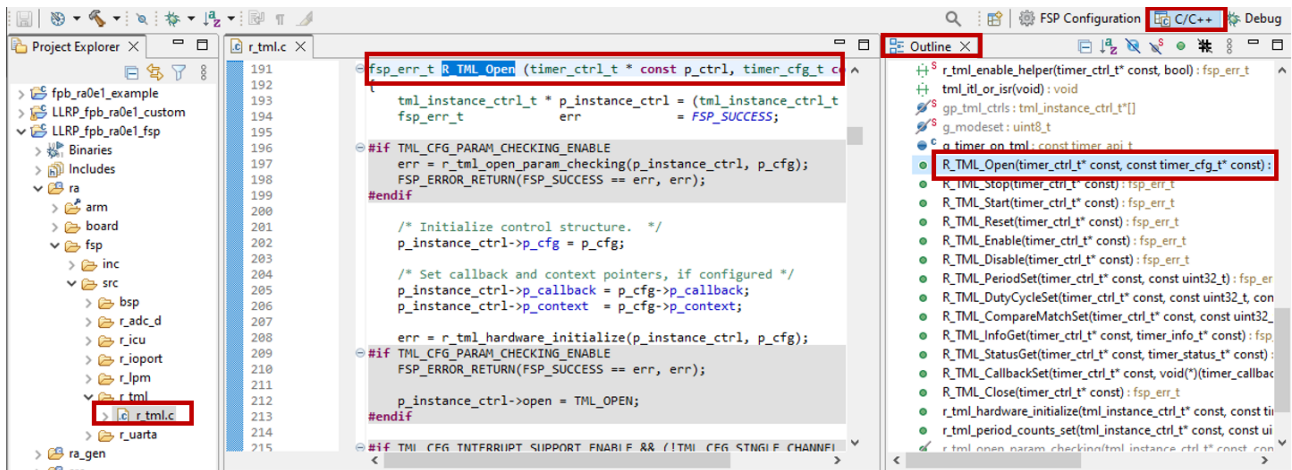


Figure 27. Clicking on an API in the C/C++ Outline View Opens the API’s Function Definition

Alternatively, if you are looking at the FSP API calls from the application level, you can **CTRL + left click** on an FSP API of interest to open the .c driver file at the line where the API definition begins.

3.2.2 General Tips when Writing Custom Drivers:

Some factors to be aware of when writing custom drivers:

1. Some registers must be set a specific way for other registers to be written to. For example, on the RA0E1, the TML’s Interval Timer Frequency Division Register’s (ITLFDIV00) bitfield FDIV0 can only be written to when the Interval Timer Control Register’s (ITLCTL0) bitfield EN0 is set to 0.
2. When setting pins directly or with with `R_BSP_PinWrite(pin, state)` you must call `R_BSP_PinAccessEnable()` before and `R_BSP_PinAccessDisable()` after.
3. Pay attention to `FSP_CRITICAL_SECTION` in the FSP driver code, and when you need to lock your code. Critical sections stop concurrent access to registers shared between modules.
4. Pay attention to any peripheral registers with write protections. You must set the corresponding bitfield of the system Protect Register PRCR (`R_SYSTEM->PRCR`) before writing to the registers and should clear the bitfield after. Alternatively, you can call the BSP functions `R_BSP_RegisterProtectEnable(registers)` or `Disable(registers)` to set the corresponding register’s PRCR bitfield. The definitions for different types of registers that can be protected on your RA device are defined in “`bsp_register_protection.h`”.
5. Pay attention to delay times for registers or bitfields that control peripherals that need to be stabilized. Specifications are in the Hardware User’s Manual.

3.2.3 Tool Highlight: e² studio Autocomplete for Register Access

The e² studio IDE has a built in **Autocomplete** function. Autocomplete is a context-aware coding accelerator that suggests possible completions for partially typed-in code elements. It works to fill in macros, enumerations, types, API functions, and on structs and their members like the SFRs definitions. The autocomplete function will only suggest items that are linked to the current file being edited. You don’t need to turn on the autocomplete function, it is always running in the background as you type in e² studio.

To use the autocomplete function with SFR access, first ensure that the iodefine file is linked to the .c custom driver file you are writing in. Next, begin to access the peripheral register struct by typing the appropriate `R_<peripheral>` phrase followed by the `->` operator to access the set of registers for the peripheral in question.

As you type, the autocomplete function should list the available members of the register struct; meaning it will list the available registers for a given peripheral. If it doesn’t open automatically use **CTRL + space** to activate autocomplete. Pressing **CTRL + space** again will navigate between the various autocomplete options.

As an example, Figure 28. Using Autocomplete to Access the ADM2 Register of the ADC Module on the FPB-RA0E1 shows how when you type “`R_ADC_D->`” then the available registers for the ADC12 module pop up in a list in the autocomplete box. You can scroll through the options to select the register in need; in this case the last line of the `ADC_Open()` routine will write a specified configuration to the “`ADM2`” register.

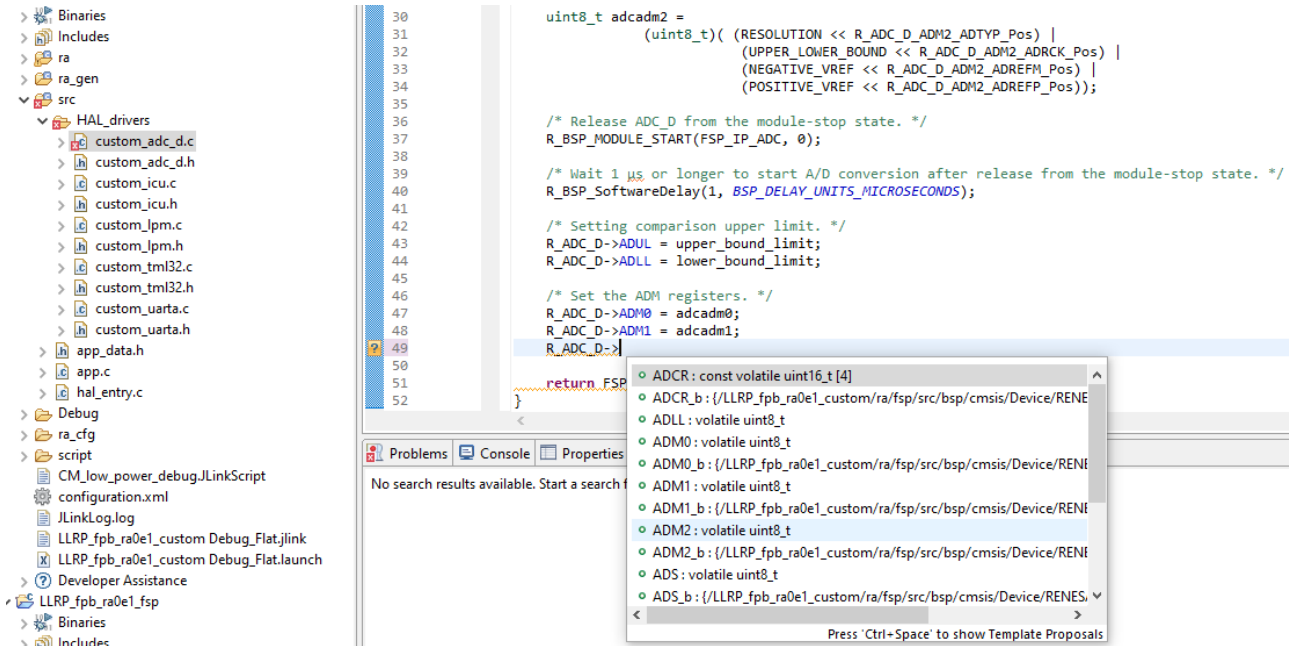


Figure 28. Using Autocomplete to Access the ADM2 Register of the ADC Module on the FPB-RA0E1

Note that the autocomplete functionality will only work for SFR access if the iodefinition file is linked to the file being edited. Anything that autocomplete can suggest is based only on the linked files as viewed by the eclipse CDT indexer i.e., the IDE.

3.2.4 Tool Highlight: Editor Hover

The **Editor Hover** feature allows users to view detailed information about identifiers in the source code by hovering over them. The pop-up window displays information related to the item under the cursor.

When you hover over a SFR register that’s written out in code, then the Editor Hover window pops up with the register definition from the device’s Hardware User’s Manual. In the image below when hovering over the **R_ADC_D->ADM2**, then the ADM2 Register Chapter of the RA0E1 Hardware User’s Manual appears in the pop-up window.

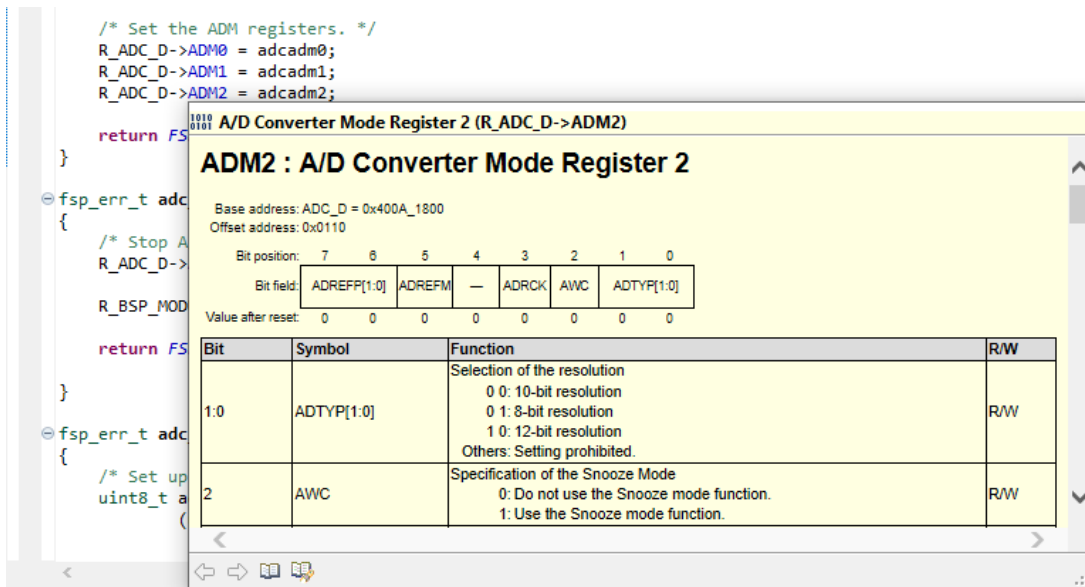


Figure 29. Hovering over R_ADC_D->ADM2 opens the ADM2 Chapter of the Hardware User’s Manual

This tool provides a useful and quick reference to review the bitfields of an SFR while you are coding. To enable/disable the hover editor functionality, use the **Window > Preferences > C/C++ > Editor > Hovers**.

3.2.5 Tool Highlight: Debugging with I/O Registers View

The **I/O Registers View** is an e² studio debugging tool that displays all SFRs on the MCU and shows their current values. To use the tool to view SFR values, a Debug Session must be active and in a paused state. The I/O Register View will only reflect the MCU’s SFR values at the currently paused point of the application.

To open the feature, click on the menu option **Renesas Views > Debug > IO Registers**. For each SFR on the MCU, the IO Register View lists the register’s abbreviation, its current value, its address, and its longer name. Clicking on any SFR will expand to show the bitfields, if applicable. When expanded to the bitfield level the tool lists each bitfield’s abbreviation, their current values, their bitfield positions, and an interpretation of what the assigned value means.

Figure 30. The I/O Registers View Shows the State of ADC_D SFRs at the Current Breakpoint provides an example of viewing the ADC_D peripheral’s SFRs at a breakpoint after the `init_hardware_routine()` function returns. Note from the menu bar and boxes at the top of the screen that the session is in a paused state and the Debug perspective is open. The ADM2 register has been expanded in the IO Register window on the right to show the register’s bitfield information.

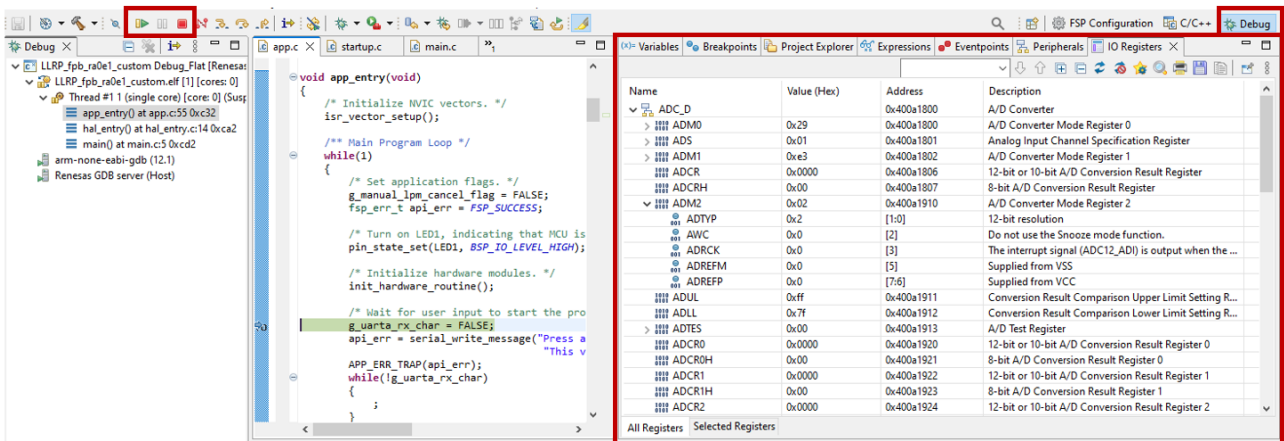



Figure 30. The I/O Registers View Shows the State of ADC_D SFRs at the Current Breakpoint

For low level register programming, the I/O Register View is helpful to validate the operation of the custom peripheral drivers before deploying them. It’s strongly encouraged to test each custom drivers’ implementation against the FSP drivers from the FSP reference project. One way could be to create a series of unit tests to ensure that the values of the SFRs are the same at the return point for each custom driver call vs each FSP driver call.

When the debug session is paused, clicking the **save** button  in the options bar located at the top of the **IO Register View** window will export the status of the SFRs to a .xml file with the name and location you specify. A simple difference between two exported I/O register status files will quickly reveal any differences between the register values and can give insight into differences in behavior or operation.

Note that the exporting process performs a read operation on every SFR; check the Hardware User’s Manual for restrictions on the registers’ read/write access.

4. Step 3: Create Custom Low-Level e²studio Project

This section describes the final step of the low-level programming guide: how to integrate the custom drivers into a project. The information can be generalized, but it will use the accompanying Low Power Mode Application Project implementation details as a practical example of the topics discussed.

This section begins by outlining an application flowchart for any RA e² studio project to better understand the role of the BSP.

Next is an overview of the Low Power Mode Application which accompanies this application note and runs on the FPB-RA0E1. The overview of the application project provides context for the custom operation and configuration of the peripherals.

Lastly, this section details the final step of this guide: creating a minimal BSP project and integrating the custom low-level drivers designed using the instructions in 3.2 Writing Custom HAL Drivers.

4.1 Overview of Generic LLRP Project

Note that the details in this section pertain to CM23 devices like the RA0E1. If you are using another device, refer to the corresponding processor core documentation from Arm.

The BSP is responsible for initializing the MCU and bringing it from reset to the user application. Before the user application is reached, the BSP configures the stacks, heap, clocks, interrupts, C runtime environment, and stack monitor. In this document, this process is referred to as the “BSP Initialization Routine”. If you use the FSP Configurator to configure a project, then the clocks, interrupts and pins settings are based on the settings in the configuration.xml.

NOTE: You must use the included BSP code otherwise you must create and import your own BSP to an e² studio RA project. It is recommended to continue using the Renesas BSP and the included BSP functions since they are all automatically invoked during the startup, before the PC reaches the main application. Custom BSP code is out of the scope of this document.

The BSP files *in fsp/src/bsp/cmsis/Device/RENESAS/Source*, “**system.c**” and “**startup.c**”, contain critical code for bringing the board out of reset adhering to the CMSIS specifications. The file “**system.c**” contains important functions for clocks, caches, and board initialization. The file “**startup.c**” contains the code for the **Reset Handler** where the MCU starts execution after the stack pointer is set up and the **Default Handler** which handles exceptions.

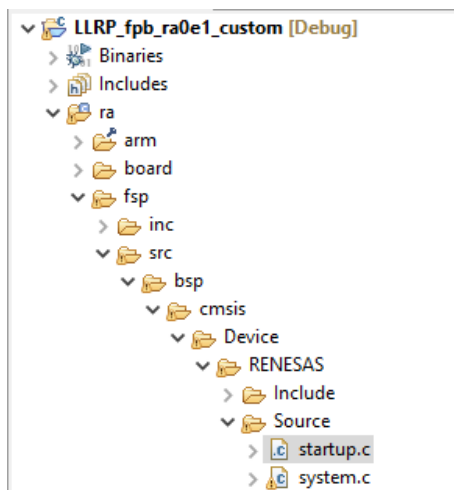


Figure 31. The Files "startup.c" and "system.c" Contain the BSP Initialization Routines

The Reset Handler implementation in “**startup.c**” is shown in Figure 32. The Reset_Handler() initializes the hardware with the BSP then calls the user application. First the Reset Handler calls the startup routine **SystemInit()** to set up the system hardware. After, it then calls the **main()** routine, bringing the program counter to the application code that you’ve added in **hal_entry()** in “**hal_entry.c**”.

```

/*****
 * MCU starts executing here out of reset. Main stack pointer is set up already.
 *****/
BSP_SECTION_FLASH_GAP void Reset_Handler (void)
{
    /* Initialize system using BSP. */
    SystemInit();

    /* Call user application. */
    main();

    while (1)
    {
        /* Infinite Loop. */
    }
}

```

Figure 32. The Reset_Handler() Initializes the Hardware with the BSP then Calls the User Application

The following flowchart below breaks down the two routines that the Reset Handler starts: the BSP Initialization routine half and the Main Application half.

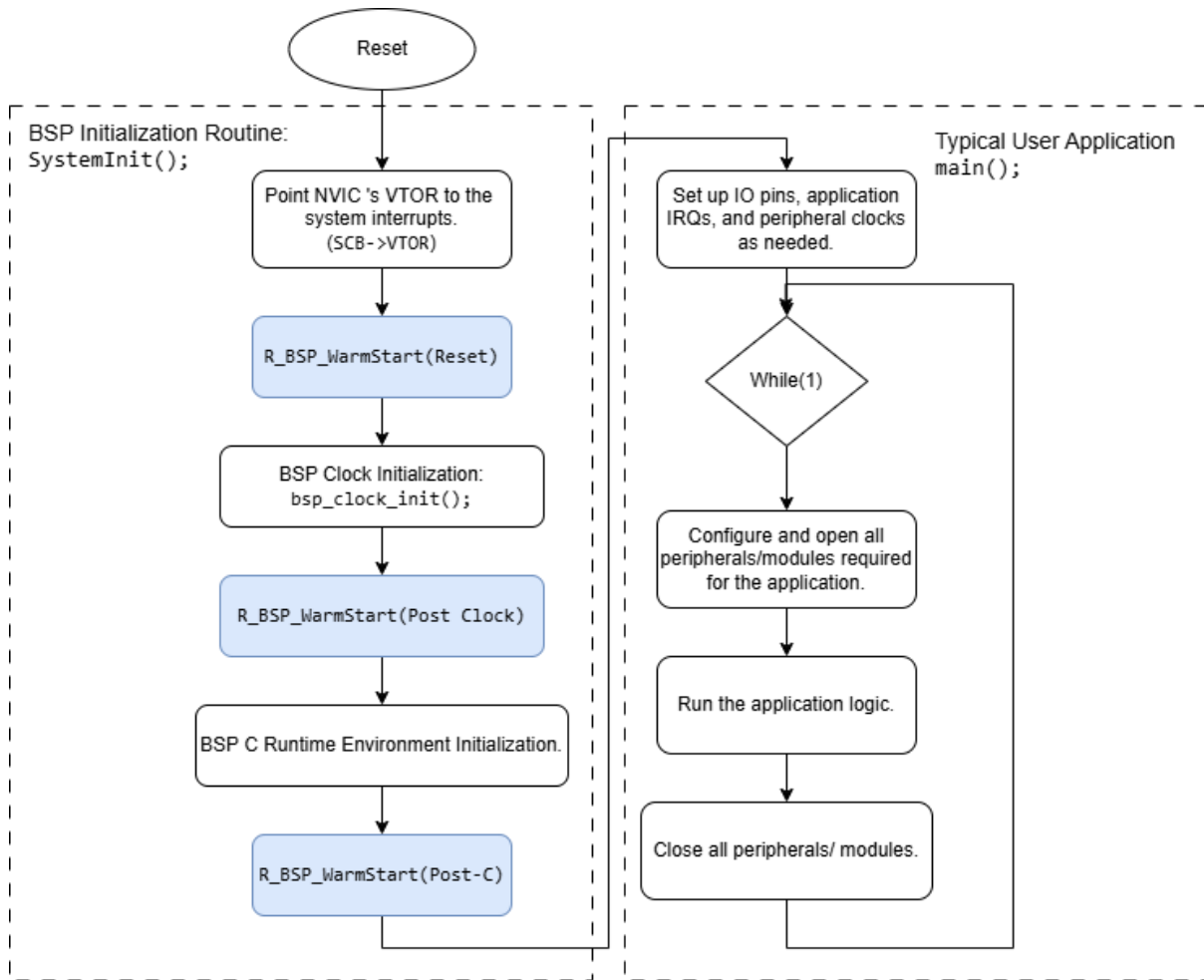


Figure 33. Flowchart of a Typical Embedded Application Using the Renesas BSP

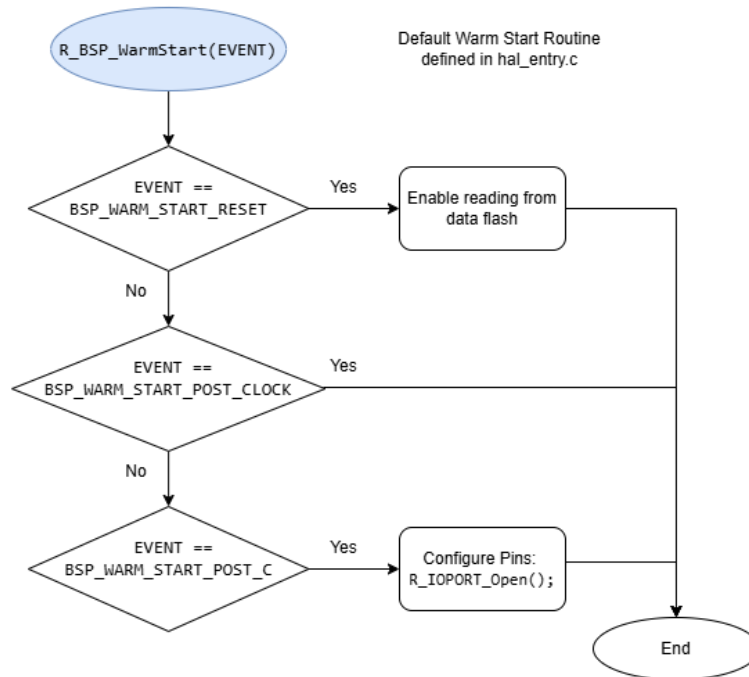
The BSP Warm Start routine and the Main Application are detailed further in the next sections.

4.1.1 BSP Warm Start

The function `R_BSP_WarmStart()` is a weakly defined function that is called multiple times during the startup process between reset and the main program.

It is called at three different event points by the startup function `SystemInit()`, which is defined in the file “`system.c`”. The argument passed of type `bsp_warm_start_event_t` indicates which of the three event points is currently executing.

The file “`hal_entry.c`” is generated with a default routine for the `R_BSP_WarmStart()` which is shown both as a flowchart and as it is implemented in code in Figure 34. The Default Implementation of the `R_BSP_WarmStart()` Callback.



```

hal_entry.c
*****
* This function is called at various points during the startup process. This implementation uses the event that is
* called right before main() to set up the pins.
*
* @param[in] event Where at in the start up process the code is currently at
*****
void R_BSP_WarmStart(bsp_warm_start_event_t event)
{
    if (BSP_WARM_START_RESET == event)
    {
        #if BSP_FEATURE_FLASH_LP_VERSION != 0

            /* Enable reading from data flash. */
            R_FACI_LP->DFLCTL = 1U;

            /* Would normally have to wait tDSTOP(6us) for data flash recovery. Placing the enable here, before clock and
            * C runtime initialization, should negate the need for a delay since the initialization will typically take more than 6us. */
        #endif
    }

    if (BSP_WARM_START_POST_C == event)
    {
        /* C runtime environment and system clocks are setup. */

        /* Configure pins. */
        R_IOPORT_Open (&IOPORT_CFG_CTRL, &IOPORT_CFG_NAME);

        #if BSP_CFG_SDRAM_ENABLED

            /* Setup SDRAM and initialize it. Must configure pins first. */
            R_BSP_SdramInit(true);
        #endif
    }
}
    
```

Figure 34. The Default Implementation of the R_BSP_WarmStart() Callback

The three different warm start callback event points are:

(1) **BSP_WARM_START_RESET:**

The **Reset** event point is reached almost immediately after reset, before the C runtime environment or clocks are initialized. For CM33 and CM23 devices, **SystemInit()** in “**system.c**” initializes the Arm system IRQs and any application IRQs defined in “**vector_data.c**” in the **ra_gen** folder before calling the Reset Warm Start callback.

The default routine for the Reset callback enables reading from the data flash. Any user routines that are defined in the callback should be basic and independent of the clock settings and the C runtime environment, for example called code should not require any non-const initialized variables.

(2) **BSP_WARM_START_POST_CLOCK:**

The **Post Clock** event point is reached after the clock initialization routine **bsp_clock_init()** returns but before the C runtime environment is initialized. The clocks are initialized by the BSP based on the information in “**bsp_clock_cfg.h**” in the **ra_gen** folder, derived from the Clocks tab settings in the configuration.xml. The BSP implements the required delays to allow the clocks to stabilize.

There are no default routines for the Post Clock callback event. Any user routines that are defined in the callback can rely on the clocks but not the C runtime environment.

(3) **BSP_WARM_START_POST_C**

The **Post-C** event point is reached after the clocks are configured and the C runtime environment has been set up.

The default routine for Post-C callback event is to configure the pins on the MCU with the BSP routine **R_IOPORT_Open()**. The pins are initialized based on the information in “**pin_data.c**” in the **ra_gen** folder, derived from the Pins tab settings in the configuration.xml. If the configuration.xml Pins tab was never edited, then the pins and ports are set to the default values as specified in the MCU's Hardware User's Manual.

If SDRAM memory is available on the MCU and is enabled, then the Post-C Warm Start callback also sets up and initializes the SDRAM after the pin configuration using **R_BSP_SdramInit()**. Note that there is no SDRAM memory on the RA0 MCU Group devices, which is why the SDRAM initialization code is greyed out in Figure 34. The Default Implementation of the **R_BSP_WarmStart()** Callback.

4.1.2 MAIN program

After the BSP Initialization Routine is completed, the Reset Handler calls the **main()** function which brings the program counter to the user application that you've defined in **hal_entry()**.

A typical embedded application will have the following operational flow:

1. Set up the pins, application-specific interrupts, and peripheral clocks if they were not set up by the BSP Initialization or Warm Start routine.
2. Initialize the modules and peripherals on the device that are needed in the application.
 - i. Make sure to initialize them in the correct order of dependencies, which is important when two or more modules work together on a specific task.
 - ii. Start the module by bringing it out of the module-stop state.
 - iii. Configure each module or peripheral to the desired operation, by setting the peripheral registers appropriately.
 - iv. If applicable, unmask any interrupts that may be used.
3. Run the application.
 - i. Operate the various peripherals and/or modules in the logical order to achieve the desired use-case application.
 - ii. Include error handling where applicable to avoid unwanted behavior.
 - iii. Integrate any callbacks and/or subroutines where needed.
4. Close all modules and peripherals before the program loop restarts.
 - i. Make sure to close peripherals and modules in the correct order of dependencies. This order may be different from the initialization order.

- ii. Mask any peripheral/module interrupts.
- iii. Put the module into the module-stop state.

4.2 Overview of Low Power Mode Project

Figure 35. High-Level Block Diagram of the Application Showing the Power Mode Transitions shows a high-level block diagram of the Low Power Modes Application:

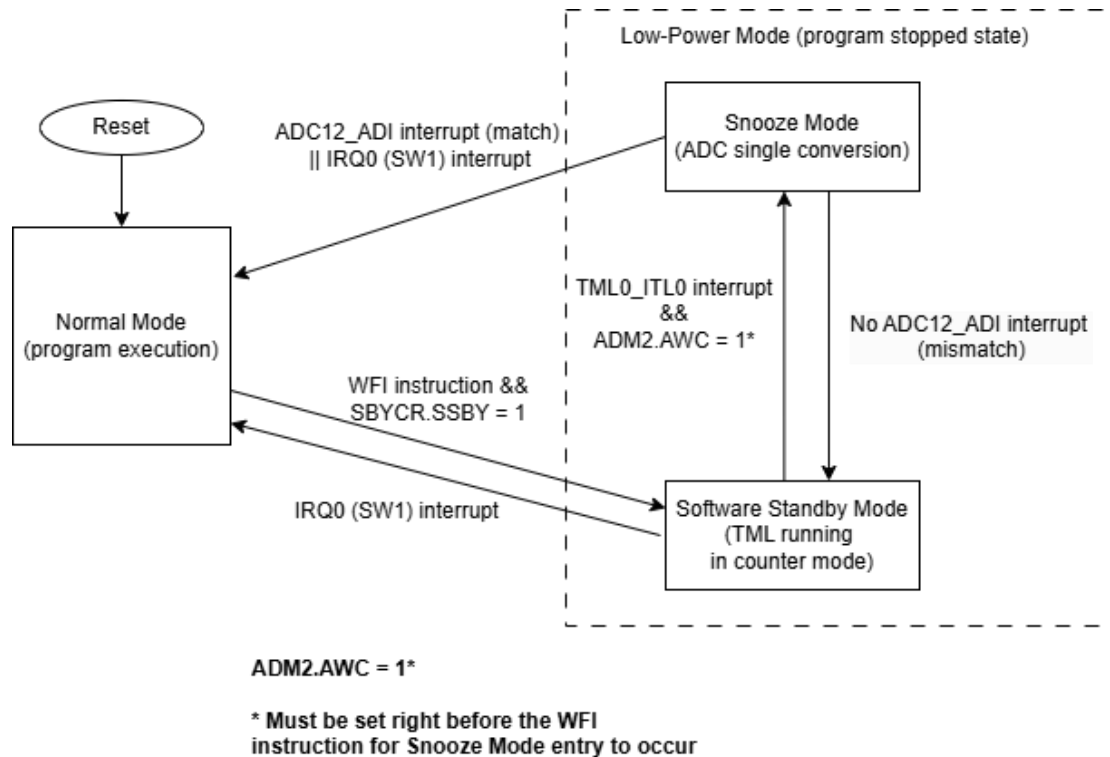


Figure 35. High-Level Block Diagram of the Application Showing the Power Mode Transitions

The application project demonstrates an example of operating the FPB-RA0E1 in the low power modes Software Standby Mode and Snooze Mode. In the application, the device begins running in Normal Mode until the WFI instruction and the low power mode (LPM) module settings transition the MCU to Software Standby Mode.

The TML32 timer is set to run in Software Standby Mode continuously, with a 5 second interrupt period. The timer end interrupt signal transitions the MCU from Software Standby Mode to Snooze Mode and the ADC performs a single conversion.

If the ADC conversion value lies between the upper and lower window bounds, the ADC match interrupt is triggered and transitions the MCU from Snooze Mode to Normal Mode. If the ADC conversion value lies outside the ADC window bounds, the resultant mismatch triggers the MCU to re-enter Software Standby Mode.

At any time, users can press the physical switch 1 (SW1) on the FPB-RA0E1, triggering the ICU interrupt to transition the MCU from Software Standby Mode or Snooze Mode to Normal Mode.

The UART module communicates via J-Link USB CDC VCOM to a serial terminal software on the host PC for controlling the application through keyboard input and viewing printed application messages.

Both the FSP and the Custom versions of the projects operate an identical use case but use different sets of peripheral drivers. The only observable “difference” is an initial message printed to the serial terminal to inform the user which project version is currently running.

4.3 Create Base Project

The final step of this programming guide is to create a project that is a low-level custom driver version of the FSP application project. At this point, you should have already created and tested your set of low-level

drivers for the application based on the FSP reference project, as explained in 3.2 Writing Custom HAL Drivers.

The following steps illustrate how to create a minimal BSP base project. The example pictures illustrate creating the custom low power application; a bare metal project targeting the FPB-RA0E1.

Open e² studio with the workspace of your choice.

Create a new project using the menu option **File > New > Renesas C/C++ Project > Renesas RA**. This will open the New C/C++ Project wizard shown in Figure 36. The New Renesas C/C++ Project Wizard Opens to the Template Options. Select **the Renesas RA C/C++ Project Template** and press **Next**.

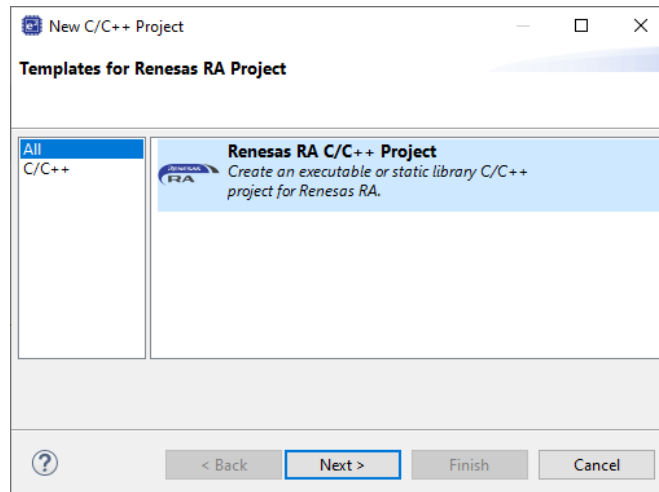


Figure 36. The New Renesas C/C++ Project Wizard Opens to the Template Options

In the next window, give the project a meaningful **name** and set the **location** where the project files are saved. The default location of the project files is in the current workspace. Press **Next**.

The next window is the Device and Tools Selection. This is where you set the **FSP version**, the **board** and **device number**, the **language** of the project, the **toolchain** and **version**, and the **debugger**. After making the desired selections, press **Next**.

The following settings come from Figure 37. Setting the Device and Tools for the Project and are used for both versions of the application project:

- FSP version: 5.7.0
- Board: FPB-RA0E1 (Selecting the board automatically populates the device number)
- Language: C
- Toolchain and version: GNU ARM Embedded v13.2.1.arm-13-7
- Debugger: J-Link ARM

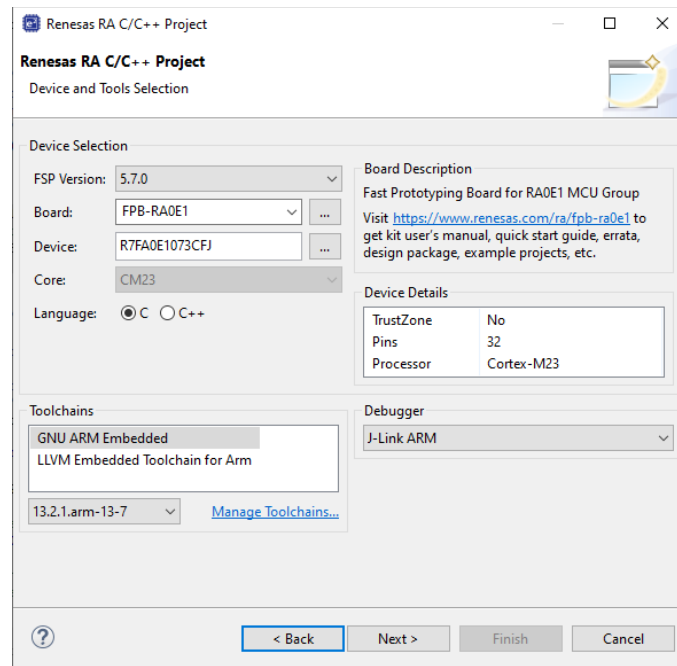


Figure 37. Setting the Device and Tools for the Project

In the next window select the desired **RTOS** and **version** if applicable and press **Next**. The application projects are bare metal so **No RTOS** was selected.

The final step is to choose the project template. Select the **Bare Metal – Minimal** option and press **Finish** to create the project.

4.3.1 Edit the BSP Properties

Open the **configuration.xml** file in the **FSP Configuration View**. While the configuration file won't be used to add module drivers to the project, it still needs to be used to configure the BSP settings of the project. Navigate to the **BSP tab** and open the **Properties**.

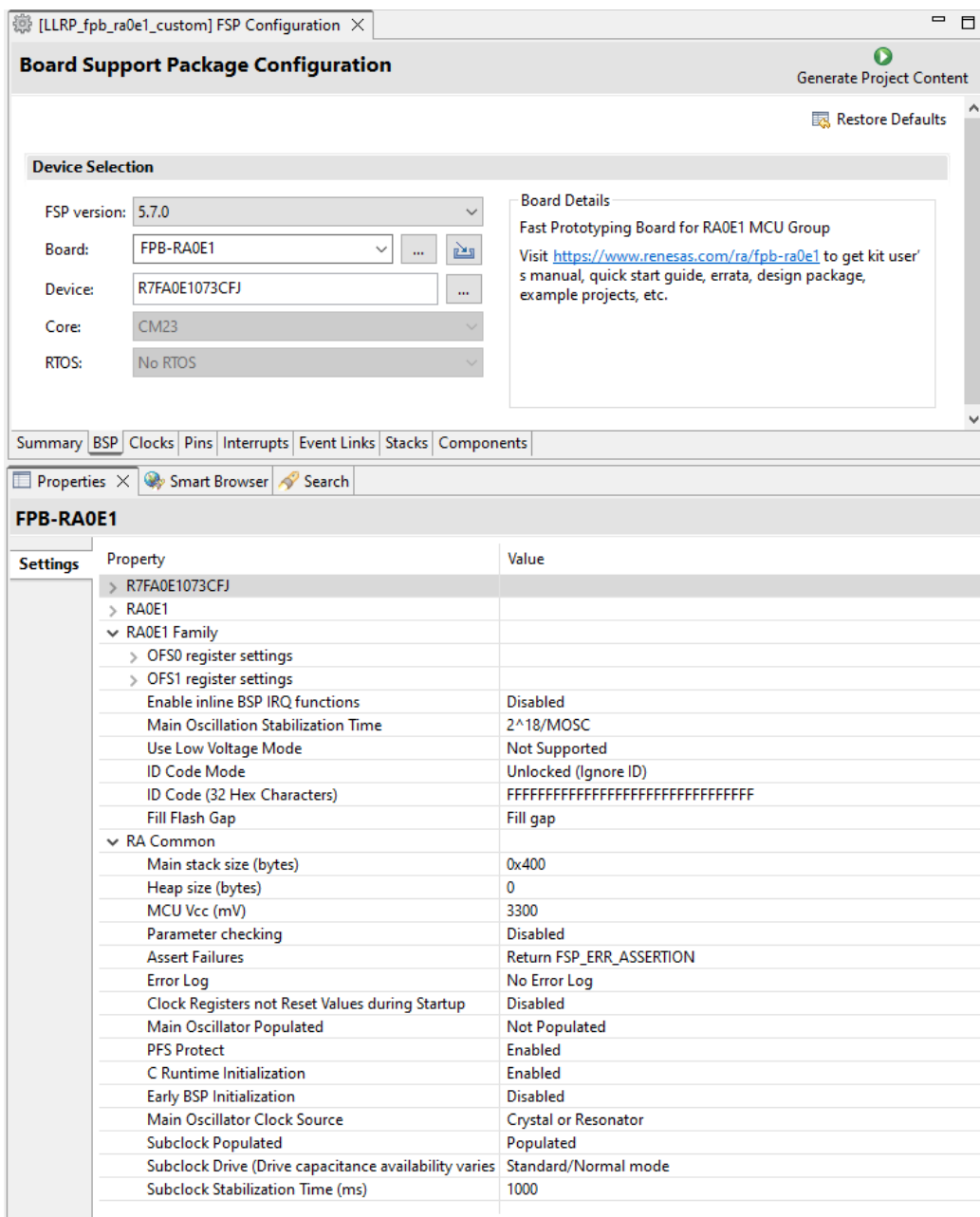


Figure 38. The BSP Settings Must Be Set in the configuration.xml

From the BSP tab you can set project features like stack size, heap size, default option function select register values, and more.

The low power application projects required no changes to the default BSP settings.

4.4 Add the Custom Drivers

The custom driver code files written for the Low Power Application are consolidated in the **src/HAL_drivers** folder. The custom driver's folder was added under the **src** folder, because all sub-folders and files in the **src** folder are automatically added recursively to the project's build.

You could add the driver folder anywhere in the project workspace, but then you need to ensure the folder and files are added to the project build. Modifications to the toolchain (e.g., compiler optimization level or adding a library to the linker) can be made via **Project > Properties > C/C++ Build > Settings**.

For each peripheral/module required in the low power application there is a **.c** and **.h** file pair that comprise the custom peripheral drivers. The driver files are named using the convention "**custom_<peripheral>.c**" and "**custom_<peripheral>.h**" and they are shown in the custom project's workspace below.

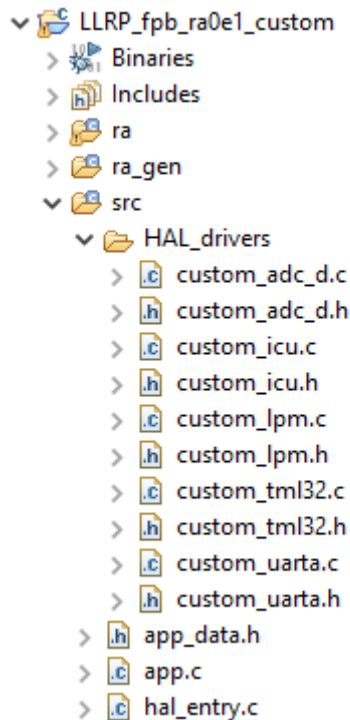


Figure 39. The Custom Driver Files are Located in the /src Folder

The .c files contain the definitions of the custom peripheral driver routines required in the low power application based off the FSP reference drivers.

The .h files are used to link the iodefinition file for SFR access and function declarations. They also contain macro definitions for the configured values to set for each register's bitfield. That way, peripheral register bitfields can be configured differently by simply changing the macro values in the .h file, without needing to update the custom drivers implementation in the .c file.

The following sections explain how the custom HAL drivers were created to meet the desired operation of each peripheral in the low power application.

DISCLAIMER: These custom drivers are not official Renesas drivers and are only meant for reference purposes. There will be no support available for the custom instance of the drivers.

Before reviewing each set of peripheral drivers, these are the major high-level differences between FSP drivers and the custom low-level drivers:

FSP Drivers: Provide a generalized set of drivers to enable complete coverage of possible usage of the module, designed in a way to follow the operation specifications from the Hardware User's Manual. The FSP implementation includes parameter checking, error control, and is updated regularly for bug fixes and new features. The FSP configurator's Stacks tab is used to generate supporting peripheral configuration and control variables that are passed to the FSP drivers.

Low-Level Custom Drivers: Provide a typically smaller set of customized drivers tailored for a specific use case of the module, and it is up to the developer to follow the operation specifications in the Hardware User's Manual. In general, all custom driver files have an accompanying .h file that contains definitions for all of the configurations required for operating that module for a specific mode.

4.4.1 32-bit Interval Timer (TML)

The 32-bit TML timer is configured to run in Software Standby Mode. It operates channel 0 in counter mode with a 5 second interval. Each time the 5 second period elapses, the TML timer triggers an interrupt signal, which is connected to the ADC.

The timer's peripheral register struct in the iodefinition file is called the `R_TML`.

The custom low-level TML drivers are comprised of the "`custom_tml32.c`" and "`custom_tml32.h`" files. The following custom drivers are included in the low power application, and their routines are summarized below:

1. **tml_open()**: Initializes the TML32 timer.
 - a. Enables the specified TML channel and takes the module out of the stop state – channel 0.
 - b. Sets the timer's operation mode – configured for counter mode.
 - c. Sets the timer's source clock – configured for SOSC.
 - d. Sets the timer's frequency division ratio – set for 5 second period.
 - e. Calls the private driver support function **period_counts_set()** to set the number of clock ticks to reach the configured period based on the timer mode – set for 5 second period.
 - f. Enables interrupts for the timer using the NVIC functions.
2. **tml_close()**: Closes the TML32 timer.
 - a. Disables interrupts for the timer using the NVIC functions.
 - b. Disables interrupt generation and stops the selected timer channel.
 - c. Resets the frequency division setting.
3. **tml_start()**: Starts the timer counting.
 - a. Clears any pending interrupt status flags.
 - b. Starts the timer by enabling the selected channel – channel 0.
4. **tml_it1_or_isr()**: ISR for the timer.
 - a. Clears the interrupt flag upon a compare match or capture event.
5. **tml_stop()**: Stops the timer counting.
 - a. Disables the selected channel – channel 0.
6. **tml_reset()**: Restarts the timer counting.
 - a. Clears the timer counter by disabling and re-enabling the channel.
 - b. Ensures the timer restarts from zero.

The main difference between the FSP and the custom driver implementation is regarding the number of TML instances that are running simultaneously on the device.

The FSP driver code considers multiple TML instances running simultaneously on multiple channels, so it has code that handles resource sharing, ultimately making the FSP drivers a more complex and thorough implementation.

The custom driver code is written based on knowing the final application's use case. Therefore, the custom driver implementation only needs to consider one TML instance and knows that the instance will run on channel 0.

4.4.2 12-bit A/D Converter (ADC_D)

The ADC is configured to run in Snooze Mode, in hardware trigger wait mode. When the TML IRQ is triggered after 5 seconds in Software Standby Mode, then the MCU transitions from Software Standby to Snooze Mode. In Snooze Mode, the ADC channel 1 converts a single input value.

The ADC window match compare is set up with an upper bound of 4095 and a lower bound of 2048. The ADC's input voltage range is from 0-3.3V.

If the ADC conversion value lies between the upper and lower window bounds that are configured, then an ADC match interrupt is triggered. The interrupt causes the MCU to transition from Snooze Mode to Normal Mode. If the ADC conversion value lies outside the ADC window bounds, the mismatch causes the MCU to re-enter Software Standby Mode until the TML timer expires again.

The specific module for the ADC on the FPB-RA0E1 is the 12-bit ADC that Renesas has named the "ADC_D" version. The ADC_D peripheral register struct in the iodefinition file is **R_ADC_D**.

The custom low level ADC drivers are comprised of the "**custom_adc_d.c**" and "**custom_adc_d.h**" files. The following custom drivers are summarized below:

1. **adc_d_open()**: Initializes the ADC.

- a. Configures analog input pin for ADC operation – sets up AN001 or channel 1.
 - b. Takes the module out of the stop state and ensures proper delay period.
 - c. Sets lower and upper window bounds for conversion comparison – 2048 and 4095.
 - d. Configures ADC operation settings, including mode, voltage reference, clock division, and trigger source.
2. **adc_d_close()**: Closes the ADC.
 - a. Stops ADC operation by disabling key control bits.
 - b. Puts the module into the stop state.
 3. **adc_d_scan_cfg()**: Configures the ADC scan settings
 - a. Sets the input channel, the input select (internal vs external signal) and test mode.
 4. **adc_d_scan_start()**: Starts the ADC scan.
 - a. Enables voltage comparator operation to start an ADC scan.
 5. **adc_d_snooze_mode_prepare()**: Prepares the ADC for operation in Snooze Mode (low-power state with hardware trigger).
 - a. Enables voltage comparator operation
 - b. Sets necessary AWC bit to enable ADC operation in Snooze Mode. This MUST occur just before the WFI instruction according to 25.7.2 A/D "Conversion by Inputting a Hardware Trigger" in the RA0E1 user manual.
 6. **adc_d_snooze_mode_exit()**: Recovers the ADC for operation in Normal Mode
 - a. Clears the AWC bit to enable ADC operation in Normal Mode.
 7. **adc_d_status_get()**: Determine whether a scan is in progress.
 - a. Retrieves the current ADC status to determine if a scan is in progress or if the ADC is idle.
 8. **adc_d_read()**: Retrieve the ADC conversion result.
 - a. Reads the 12-bit ADC conversion result and stores it in the provided 16-bit variable.

4.4.3 Low Power Modes (LPM)

The LPM drivers are responsible for determining which hardware and software sources will trigger the MCU to transition between the various power mode states available. The FPB-RA0E1 has Sleep Mode, Software Standby Mode, and Snooze Mode low power options.

In the low power application, the LPM is configured so the MCU enters Software Standby Mode rather than Sleep Mode when the WFI instruction occurs.

For transitions, the LPM module is configured to transition the MCU from Snooze Mode to Normal Mode when the ADC match IRQ occurs. Additionally, the LPM is configured to transition the MCU from either Software Standby Mode or Snooze mode back to Normal Mode when the SW1 external interrupt is triggered by the user.

The LPM's peripheral register struct in the iodef file is called **R_LPM**. The custom low level LPM drivers are comprised of the "**custom_lpm.c**" and "**custom_lpm.h**" files. The following custom drivers are summarized below:

1. **lpm_open()**: Initializes the LPM.
 - a. Enables writing to the LPM registers.
 - b. Configures the startup speed of the HOCO when entering Snooze Mode – normal speed.
 - c. Sets the SOSC state in Snooze Mode – SOSC is enabled so the TML can continue running.
 - d. Chooses the Snooze Mode return source – the ADC match interrupt.
 - e. Sets what low power mode is entered when the WFI occurs – Software Standby Mode.
 - f. Sets the flash mode in Snooze Mode – flash stopped.

- g. Disables writing to the LPM registers.
2. **lpm_enter()**: Transitions the MCU to Software Standby Mode.
 - a. Enables writing to the LPM registers – set the corresponding bitfield of the PRCR.
 - b. Stops and stores the state of the DTC if it is not used as a Snooze request source – no DTC peripheral used in low power application but added as an example.
 - c. Calls the DSB and WFI instructions, transitioning the MCU to Low Power Modes.
 - d. After returning to Normal Mode, restores the DTC registers – no DTC used but included this part of the routine as a demonstration.
 - e. Disables writing to the LPM registers – clear the corresponding bitfield of the PRCR.

4.4.4 External IRQ (ICU)

The ICU drivers are responsible for setting up the external interrupt signals for the low power application. Switch 1 on the FPB-RA0E1 is an external interrupt signal to cancel Software Standby or Snooze Mode. The SW1 interrupt is mapped to the external IRQ Channel 0.

The ICU peripheral essentially acts as a filter for all external interrupt signals available on the FPB-RA0E1, enabling and disabling the channels as configured.

The ICU peripheral is controlled by a register cluster with the peripheral struct base **R_ICU**. Ultimately though, the NVIC is what controls all interrupts on the MCU and its registers can be accessed using the NVIC APIs.

The custom low level ICU drivers are comprised of the “**custom_icu.c**” and “**custom_icu.h**” files. The following custom drivers are summarized below:

1. **icu_external_irq_open()**: Enables the external interrupt signal for SW1.
 - a. Sets the trigger mode for the channel 0 external IRQ in the ICU – falling signal for active low SW1.
 - b. Configures the SW1 IRQ priority in the NVIC – level 2.
2. **icu_external_irq_enable()**: Enable the SW1 IRQ in the NVIC.
 - a. Clear the interrupt status and pending bits
 - b. Enable the interrupt.
3. **icu_external_irq_disable()**: Disables the SW1 IRQ in the NVIC
 - a. Disable the interrupt.

4.4.5 Serial Interface UART (UARTA)

The UARTA drivers are used in the low power application to communicate via J-Link USB CDC VCOM to a serial terminal software on the host PC. They help to control the application with reception of keyboard inputs and transmission of printed application messages.

The drivers control the reception and transmission of the UARTA peripheral on the FPB-RA0E1 which has the peripheral register base **R_UARTA0**.

Their implementation is in the “**custom_uarta.c**” and “**custom_uarta.h**” files. The following custom drivers are summarized below:

1. **uarta_open()**: Opens the UARTA peripheral.
 - a. Enables the UARTA – channel 0.
 - b. Calls **uarta_baud_rate_set()** to configure the baud rate.
 - c. Configures UARTA communication settings (parity, data bits, stop bits, etc.).
 - d. Enables RX and TX operations.
 - e. Configures and enables TX, RX, and Error interrupts.
2. **uarta_baud_rate_set()**: Sets the baud rate to 115200.

- a. Sets the baud rate generator control register – must calculate the configurations based on the UARTA input clock frequency and the target baud rate.
 - b. Configures the division ratio for the UARTA clock - the calculation expressions are found in FPB-RA0E1 Hardware User's Manual Chapter 23.3.4.
3. **uarta_close()**: Closes the UARTA peripheral.
 - a. Disables TX and RX operations.
 - b. Disables baud clock output.
 - c. Disables UARTA-related interrupts.
 4. **uarta_op_stop()**: Stops the UARTA operation.
 - a. Stops UARTA transmission and reception.
 - b. Disables the baud clock.
 5. **uarta_op_start()**: Starts the UARTA operation.
 - a. Starts UARTA transmission and reception.
 - b. Enables the baud clock.
 6. **uarta_write()**: Transmit message.
 - a. Disables TX interrupt to prevent race conditions.
 - b. Sets the transmission buffer pointer and byte counter.
 - c. Enables TX interrupt.
 - d. Sends the first byte manually to start the transmission.
 7. **uarta_txi_isr()**: TX Interrupt Service Routine.
 - a. Sends the next byte if there are remaining bytes.
 - b. When transmission is fully complete, it ends by calling **uarta_user_callback()** passing the event `UART_EVENT_TX_COMPLETE`
 8. **uarta_rxi_isr()**: RX Interrupt Service Routine.
 - a. Reads a received byte from UARTA.
 - b. Checks for errors and clears error conditions if necessary.
 - c. Calls **uarta_user_callback()** to notify the user of received data. Will send a `UART_EVENT_RX_CHAR` event for each byte received and one `UART_EVENT_RX_COMPLETE` event when the full message is received.
 9. **uarta_eri_isr()**: Error Interrupt Service Routine. (Not used in the application, due to the value of the UARTA's `ASIMA00` register's `ISRMA` bitfield. This bitfield sets the Receive Interrupt Mode Select where either the RX ISR or ER ISR exclusively handles both RX and ER IRQs.)
 - a. Handles reception errors (e.g., framing or parity errors).
 - b. Clears the error flag.
 - c. Notifies the error using **uarta_user_callback()** passing the relevant error event.

Key Takeaways

- The routines **uarta_open()** and **uarta_close()** manage UART initialization and shutdown.
- The routine **uarta_write()** handles non-blocking transmission by enabling the TX interrupt.
- The routine **uarta_baud_rate_set()** sets the `BRGCA0` and `UTA0CK` registers based on the input clock and the calculations found in the FPB-RA0E1 Hardware User's Manual Chapter 23.3.4.
- The Interrupt Service Routines **uarta_txi_isr()**, **uarta_rxi_isr()**, **uarta_eri_isr()** process TX completion, received data, and errors and end by calling **uarta_user_callback()** defined in "app.c".

- The user callback `uarta_user_callback()` notifies higher-level application code about transmission completion, received characters, or errors.
- The different events for transmission, reception, and errors that can be passed to the `uarta_user_callback()` are defined in the file “`custom_uarta.h`” under the `e_sf_event` variable

4.4.6 Useful BSP APIs

If the BSP is not removed from the custom application, it's best to take advantage of the BSP APIs. They are included through “`bsp_api.h`”, which is already included in each peripheral driver header file for accessing the iodefinition file.

Here are some of the most useful APIs for custom peripheral drivers:

- **R_BSP_RegisterProtectEnable(register_set) or Disable(register_set):**
 - Sets or clears the right bitfield in the device's Protect Register (`R_SYSTEM->PRCR`).
 - The parameters for different types of registers that can be protected on the device are defined in “`bsp_register_protection.h`”
- **R_BSP_MODULE_START(module) or STOP(module)**
 - Starts or stop the module state by setting or clearing the corresponding bitfield of `MSTP`.
 - Inline functions defined in “`bsp_module_stop.h`”
 - The parameters for available modules on the device are in “`fsp_features.h`”
- **R_BSP_SoftwareDelay(32-bit delay_value, delay_units)**
 - Enacts a software delay of *at least* the specified value. Delay units can be seconds, milliseconds, or microseconds.
 - Function defined in “`bsp_delay.c`”
 - Parameters for available delay units are in “`bsp_delay.h`”
- **R_BSP_PinAccessEnable() or Disable()**
 - Enables or disables writing to device pins PFS registers, by setting or clearing the Write-Protect Register (`R_PMISC->PWPR`)
 - Function defined in “`bsp_io.h`”
 - Void input
- **R_BSP_PinWrite(pin, state)**
 - Series of inline functions for writing to pins on the MCU, by setting the Port/Pin Function Select Register (`R_PFS->PORT[m].PIN[n].PmnPFS`)
 - Inline function defined in “`bsp_io.h`”
 - Parameters for pins and states also in “`bsp_io.h`”

As an example, the subroutine `pin_state_set(pin, state)` defined in both application projects' file “`app.c`” utilizes the BSP pin functions to set the pin to the specified state. The implementation is shown in the image below.


```

/* Supporting function to set the state of a pin. */
void pin_state_set(bsp_io_port_pin_t pin, bsp_io_level_t state)
{
    /* Enable pin access. */
    R_BSP_PinAccessEnable();

    /* Set the pin state. */
    if(state == BSP_IO_LEVEL_HIGH)
    {
        R_BSP_PinWrite(pin, BSP_IO_LEVEL_HIGH);
    }
    else
    {
        R_BSP_PinWrite(pin, BSP_IO_LEVEL_LOW);
    }

    /* Disable pin access. */
    R_BSP_PinAccessDisable();
}

```

Figure 40. The BSP APIs for Pin Access Are Used in the Subroutine pin_state_set()

The subroutine first calls the `R_BSP_PinAccessEnable()` function to enable accessing and writing to the IO pins. Next the `R_BSP_PinWrite(pin, state)` performs the actual pin write and level change. The subroutine ends with a call to the `R_BSP_PinAccessDisable()` function to protect the IO pins from accidental access as the application continues.

4.5 Configure Clocks Using HAL Registers

The BSP initialization routine sets up all system clocks in the clock generation circuit (CGC) based on the settings in “`bsp_clock_cfg.h`”, which is derived from the Clocks tab of the configuration.xml. The BSP clock initialization sets the CGC registers in a precise order and with exact wait times for clock stability.

It is out of the scope of this guide to explain writing a custom implementation of the BSP clock initialization routine, since that is an integral part of the BSP startup code. If you don't wish to use the BSP then you will need to recreate your own version, adhering to Arms's and Renesas's specifications for the device.

It's highly recommended to take advantage of the Clocks tab of the configuration.xml if the core system clocks, like the HOCO, MOCO, LOCO, etc., require a specific configuration out of MCU restart. That way, the BSP will initialize the system core clocks properly during board bring-up. However, any peripheral clocks on the device can be more easily adjusted as the application runs using the peripheral drivers.

For the custom version of the low-power application, the Clocks tab was never edited, so the default values of the configuration were set by the BSP clock initialization. Only the UARTA and TML peripheral clock sources needed to be adjusted for the application, which happens at application runtime.

This can happen at runtime rather than in the initialization because the peripheral clock registers for the UARTA and the TML are a part of each respective peripheral's SFRs, rather than a part of the CGC registers.

Figure 41. Example Setting the TML's Peripheral Clock Source to the SOSC in the `tml_open()` Driver shows how the TML32's Interval Timer Clock Select Register 0 (`R_TML->ITLCSEL0`) is set in the custom driver `tml_open()`. The lines highlighted sets the clock source to the sub-clock source FSXP, which is the SOSC.

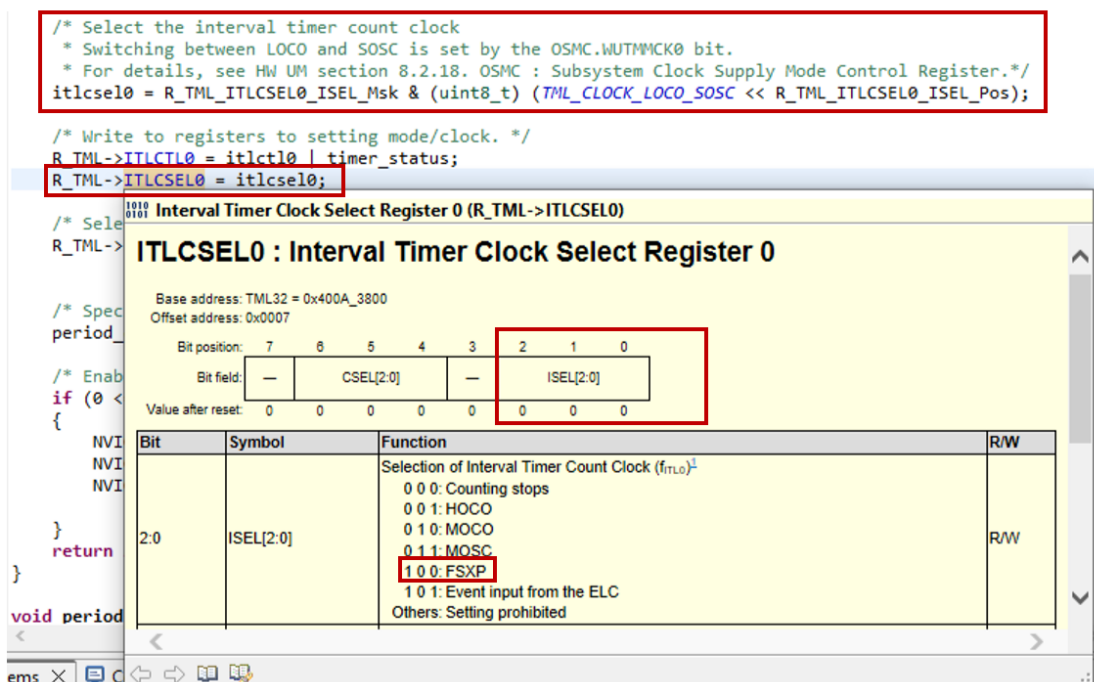


Figure 41. Example Setting the TML’s Peripheral Clock Source to the SOSC in the tml_open() Driver

In general writing to the CGC registers is a careful process. The right bitfield of the System Protect Register (R_SYSTEM->PRCR) must be set to even begin writing to the CGC registers. The bitfield should be cleared after writing to the CGC registers to protect the clock operation.

4.6 Add Interrupts Using NVIC

On the FPB-RA0E1, a CM23 device, the interrupt vector table begins with ARM’s 16 standard system exceptions followed by 39 MCU-specific application interrupts. The table below is an excerpt of Table 11.3 in the Interrupt Controller Unit (ICU) Chapter of the RA0E1 Hardware User’s Manual to illustrate the first 16 Arm standard exceptions.

Table 1. The ARM System Exceptions on the CM-23 RA0E1

Exception number	IRQ number	Vector offset	Source	Description
0	—	0x000	Arm	Initial stack pointer
1	—	0x004	Arm	Initial program counter (reset vector)
2	—	0x008	Arm	Non-Maskable Interrupt (IWDT Underflow/Refresh Error Interrupt, Voltage monitor 1 Interrupt, NMI Pin Interrupt, SRAM Parity Error Interrupt)
3	—	0x00C	Arm	Hard Fault
4	—	0x010	Arm	Reserved
5	—	0x014	Arm	Reserved
6	—	0x018	Arm	Reserved
7	—	0x01C	Arm	Reserved
8	—	0x020	Arm	Reserved
9	—	0x024	Arm	Reserved
10	—	0x028	Arm	Reserved
11	—	0x02C	Arm	Supervisor Call (SVCcall)
12	—	0x030	Arm	Reserved
13	—	0x034	Arm	Reserved

The BSP function `SystemInit()` in “`system.c`” is responsible for bringing the MCU out of reset and to the main application. Since CM23 devices have an undefined value in the NVIC’s VTOR register out of reset, `SystemInit()` sets the NVIC register `SCB->VTOR` to point to the address of the Arm standard system vector table defined in the BSP file “`startup.c`” and shown in the image below.

```

/* Vector table. */
BSP_DONT_REMOVE const exc_ptr_t __Vectors[BSP_CORTEX_VECTOR_TABLE_ENTRIES] BSP_PLACE_IN_SECTION(
    BSP_SECTION_FIXED_VECTORS) =
{
    (exc_ptr_t) (&g_main_stack[0] + BSP_CFG_STACK_MAIN_BYTES), /* Initial Stack Pointer */
    Reset_Handler, /* Reset Handler */
    NMI_Handler, /* NMI Handler */
    HardFault_Handler, /* Hard Fault Handler */
    MemManage_Handler, /* MPU Fault Handler */
    BusFault_Handler, /* Bus Fault Handler */
    UsageFault_Handler, /* Usage Fault Handler */
    SecureFault_Handler, /* Secure Fault Handler */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler, /* SVC Call Handler */
    DebugMon_Handler, /* Debug Monitor Handler */
    0, /* Reserved */
    PendSV_Handler, /* PendSV Handler */
    SysTick_Handler, /* SysTick Handler */
};

```

Figure 42. RA0E1 Arm Standard System Vector Table in “`startup.c`”

In the low power application, the 16 system exceptions vector table is copied from the `SCB->VTOR` into the variable array `app_vector_table`. The table is then appended with the application interrupts by calling `NVIC_SetVector()` for each application ISR.

```

/* Routine to provide the NVIC a pointer to the ISR Vector Table. */
void isr_vector_setup(void)
{
    /* Copy the vector table. */
    memcpy( (void *) app_vector_table, (void *) SCB->VTOR, sizeof(app_vector_table) );
    SCB->VTOR = (uint32_t) &app_vector_table[0];

    /* UARTA interrupts. */
    NVIC_SetVector(UARTA0_RXI_IRQn, (uint32_t) uarta_rxi_isr);
    NVIC_SetVector(UARTA0_TXI_IRQn, (uint32_t) uarta_txi_isr);
    #if(UARTA0_INT_RXI == 0)
    NVIC_SetVector(UARTA0_ERI_IRQn, (uint32_t) uarta_eri_isr);
    #endif

    /* IRQ SW1 interrupt. */
    NVIC_SetVector(ICU_IRQ0_IRQn, (uint32_t) sw1_lpm_cancel_callback);

    /* TML32 interrupt. */
    NVIC_SetVector(CYCLE_END_IRQ, (uint32_t) tml_itl_or_isr);
}

```

Figure 43. Adding the Application Interrupts to the Custom Low Power Application

The `isr_vector_setup()` routine is called during the Warm Start Post-C event defined in “`hal_entry.c`”.

```

if (BSP_WARM_START_POST_C == event)
{
    /* C runtime environment and system clocks are setup. */

    /* Configure pins. */
    R_IOPORT_Open (&IOPORT_CFG_CTRL, &IOPORT_CFG_NAME);

    /* Initialize NVIC vectors. */
    isr_vector_setup();
}

```

Figure 44. The Application Vectors are Initialized in the Warm Start Post-C Callback

Visit table 11.3 in the Interrupt Control Unit (ICU) chapter of the RA0E1 Hardware User's Manual to see the definitions for the 16 reserved CMSIS system interrupts and the 39 device ICU application interrupts.

The NVIC functions for the RA0E1 are defined in the “`core_cm23.h`” file. Along with appending vectors they also enable runtime control of interrupts through priority setting and toggling IRQs. The following functions were used:

- **NVIC_SetPriority(IRQn, priority)**: Set the priority of the device ICU application interrupt IRQn.
- **NVIC_ClearPendingIRQ(IRQn)**: Clear any pending requests for IRQn.
- **NVIC_EnableIRQ(IRQn)**: Enable interrupt requests for IRQn.
- **NVIC_DisableIRQ(IRQn)**: Disable interrupt requests for IRQn.

As an example, the `uarta_open()` custom API ends with calling NVIC functions to set priorities for and enable the TX and RX interrupts:

```

/* Configure and enable the TX, RX, and ERR interrupts. */
#if(UARTA0_INT_RXI == 0)
    NVIC_SetPriority(UARTA0_ERI_IRQn, UARTA0_ERI_IPL);
    NVIC_EnableIRQ(UARTA0_ERI_IRQn);
#endif
NVIC_SetPriority(UARTA0_RXI_IRQn, UARTA0_RXI_IPL);
NVIC_EnableIRQ(UARTA0_RXI_IRQn);

/* Wait for the period of at least one cycle of the UARTA operation clock according to "Note" in section 23.2.3
 * "ASIMA00 : Operation Mode Setting Register 00" in the RA0E1 Hardware User's Manual r01uh1040ej0110-ra0e1.pdf .*/
R_BSP_SoftwareDelay((uint16_t) UARTA_CLK_PER, BSP_DELAY_UNITS_MICROSECONDS);

NVIC_SetPriority(UARTA0_TXI_IRQn, UARTA0_TXI_IPL);
NVIC_EnableIRQ(UARTA0_TXI_IRQn);

```

Figure 45. Using NVIC Functions to Set Priorities and Unmask IRQs

5. Running the Low Power Mode Application Projects

There are two versions of the application project located in the folder *Low Level Register Programming*: the FSP reference project `LLRP_fbp_ra0e1_fsp` and the custom driver version `LLRP_fbp_ra0e1_custom`.

The application's description is detailed in 4.2 Overview of Low Power Mode Project.

5.1 Script for Low Power Mode Debugging

When debugging an application where the MCU enters a low power mode, you need to change the debug settings. When using the Segger J-Link emulator to debug, the default configuration will not operate properly when the MCU enters a low power mode state. The connection to the CPU and the IDE can be disconnected and the IDE debug session may be terminated. Even if the debug session appears to stay connected and working, undefined behavior may occur.

The debug script “`CM_low_power_debug.JLinkScript`” is included with each application project. This script corrects the issue to allow for low power handling when the application project enters Software Standby or Snooze Mode. This debug script is only meant for developing applications, not for actual low power evaluation.

Ensure that the script and debug settings are set up properly for each project for low power mode operation by verifying the settings of the project's Debug Configurations. To open the debug settings, select the project of interest in the workspace and **right click > Debug As > Debug Configurations...** Make sure the details match those shown in the Figure 46. Adding the Low Power Mode Debug Script to the Debug Configurations.

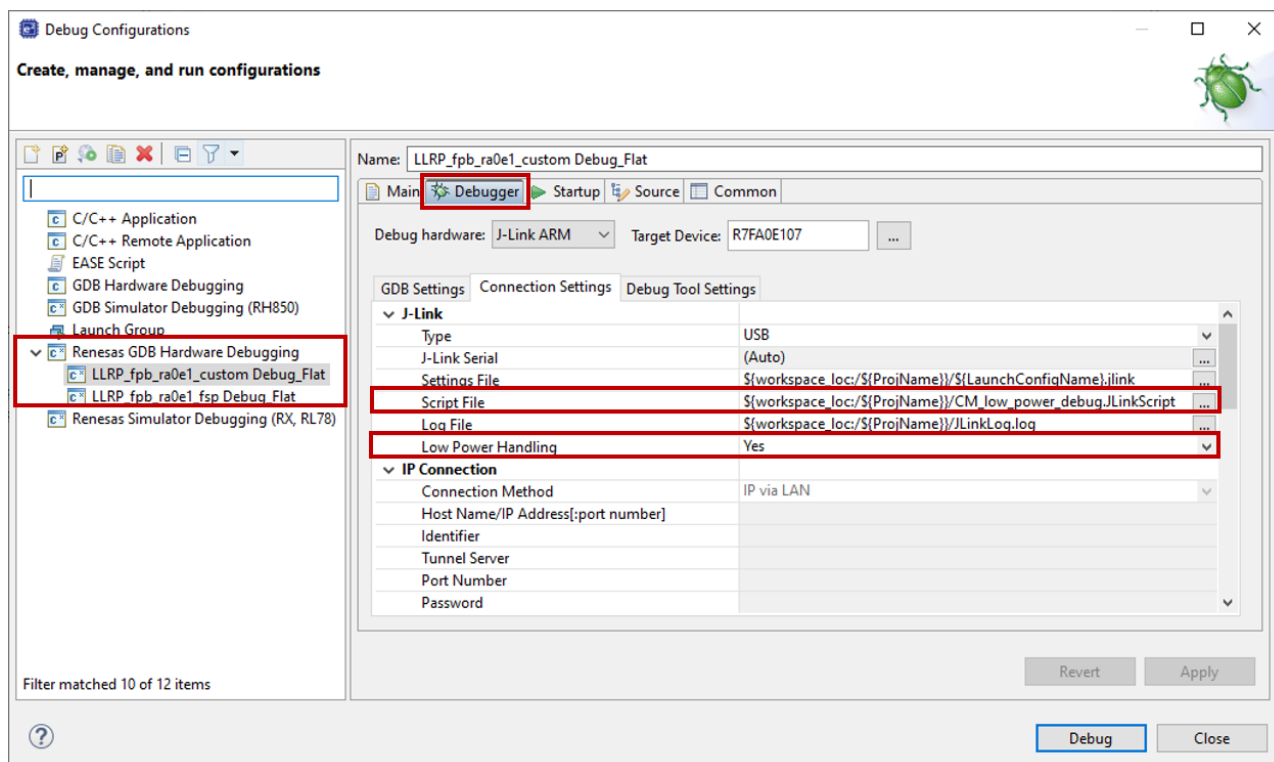


Figure 46. Adding the Low Power Mode Debug Script to the Debug Configurations

Note that accurate I_{cc} values cannot be measured when measuring the current consumption I_{cc} when these settings are enabled. Disable OCD (Debug function) and execute a power on reset when measuring the current.

5.2 Project Requirements

The software and hardware requirements for both versions of the project are as follows:

(1) Software Requirements

- e²studio v2024-10 with FSP v5.7.0
- Tera Term or similar serial terminal application

(2) Hardware Requirements

- FPB-RA0E1 with J-Link VCOM enabled:
 - Short the E4 trace (back of the board)
 - Add 150ohm resistor to R10 (front of the board)
- Variable voltage generator with 0-3.3V range as input

5.3 Build, Download, and Run the Project

1. Make sure that the FPB-RA0E1 has hardware connections to enable J-Link VCOM. See section 5.2 Project Requirements for details.
2. Connect the FPB-RA0E1 and the PC using the USB C cable.
3. Connect the variable voltage generator to P011, corresponding to the ADC Channel 1 input AN001.
4. Launch the terminal software on the PC and choose "COMxx: JLink CDC UART Port". Change the serial port settings to:
 - Baud rate: 115,200
 - Data length: 8 bits
 - Parity: None

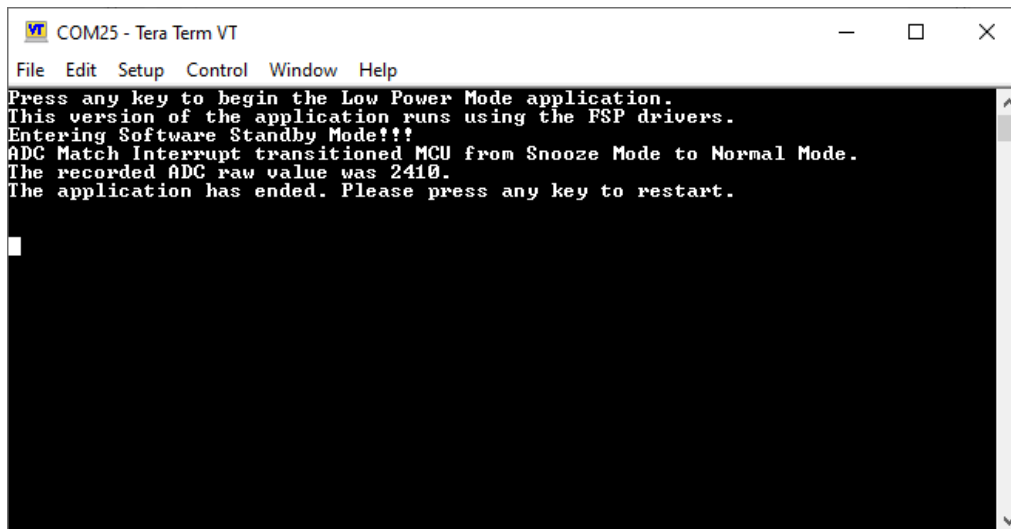
- Stop-bit length: 1 bit
 - Flow control: None
5. Launch e²studio, Import the projects in the folder **Low_Level_Register_Programming**.
 6. Check that the Debug Configurations for each project are configured for the MCU to enter low power modes. See section 5.1 Script for Low Power Mode Debugging for details
 7. Select the project you want to run in the workspace. Press Build.
 8. Open the Debug Perspective and press Run twice.
 9. Follow the prompts in the serial terminal to run the application.

5.4 Verifying the Low Power Mode Application

When the application triggers the MCU to enter Software Standby Mode, there are two ways that you can wake the MCU to Normal Mode.

(1) ADC Match Compare (Snooze Mode -> Normal Mode)

One way is to set the voltage generator to trigger the ADC match compare after 5 seconds elapses on the TML, by setting a voltage in the range of [1.65V, 3,3V). The ADC match compare interrupt will transition the MCU from Snooze Mode to Normal Mode and the value in the ADC register is printed in the serial console [2048, 4095].

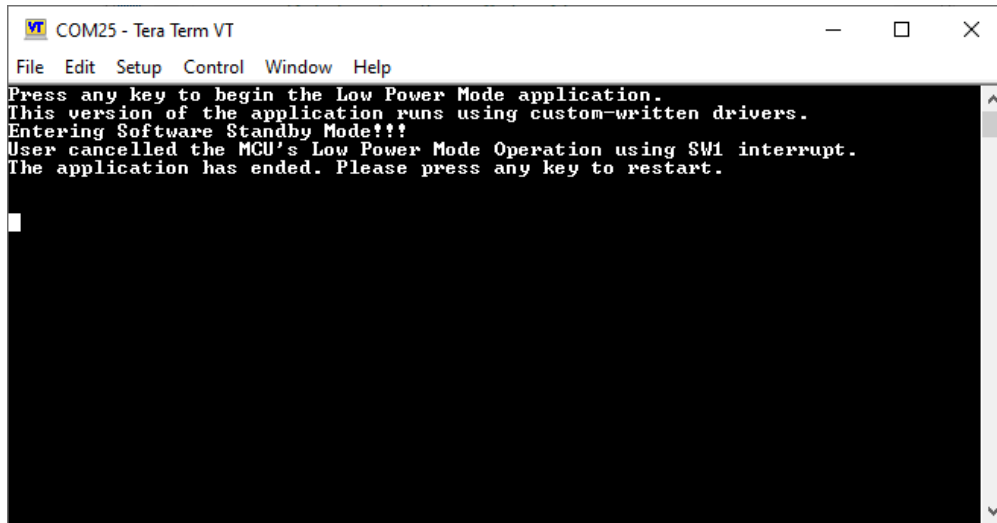


```
COM25 - Tera Term VT
File Edit Setup Control Window Help
Press any key to begin the Low Power Mode application.
This version of the application runs using the FSP drivers.
Entering Software Standby Mode!!!
ADC Match Interrupt transitioned MCU from Snooze Mode to Normal Mode.
The recorded ADC raw value was 2410.
The application has ended. Please press any key to restart.
```

Figure 47. Serial Terminal Output When Triggering the ADC Match Interrupt to Wake the MCU

(2) Switch 1 Interrupt (Software Standby -> Normal Mode)

If the voltage is outside the window bounds, in the range of (0V, 1.65V), the ADC match compare is not triggered. You can wake the MCU from Software Standby Mode to Normal Mode by pressing SW1 on the FPB-RA0E1.



```
COM25 - Tera Term VT
File Edit Setup Control Window Help
Press any key to begin the Low Power Mode application.
This version of the application runs using custom-written drivers.
Entering Software Standby Mode!!!
User cancelled the MCU's Low Power Mode Operation using SW1 interrupt.
The application has ended. Please press any key to restart.
```

Figure 48. Serial Terminal Output When Triggering the SW1 to Wake the MCU

For technicality's sake, there is an extremely small chance pressing SW1 will transition the MCU from Snooze Mode to Normal Mode if the SW1 interrupt occurs at the exact moment the ADC is converting the value in the ADCR register. However this is very small, so the SW1 interrupt typically transitions the MCU from Software Standby Mode to Normal Mode.

6. References

Software:

Renesas Flexible Software Package v5.7.0 UM (r11um0155)

[Renesas e²studio v2023-10 or Higher User's Manual](#) (r20ut5210)

Hardware:

ARM Cortex M-23 Doc Set - <https://developer.arm.com/Processors/Cortex-M23>

RA0E1 Group User's Manual: Hardware (r01uh1040)

FPB-RA0E1 User's Manual (r20ut5378)

FPB-RA0E1 Schematic (r12tu0275)

Website and Support

Visit the following vanity URLs to learn about key elements of the RA family, download components and related documentation, and get support.

RA Product Information	www.renesas.com/ra
RA Product Support Forum	www.renesas.com/ra/forum
RA Flexible Software Package	www.renesas.com/FSP
Renesas Support	www.renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.26.25	—	First release document

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.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

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/.